VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

MODUL PRO TRANSFORMACI DAT PRO DIGITÁLNÍ ÚŘEDNÍ DESKY

DATA TRANSFORM MODULE FOR DIGITAL OFFICIAL BOARDS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **MARTIN LEONOV**
AUTHOR

VEDOUCÍ PRÁCE — **RNDr. MAREK RYCHLÝ, Ph.D.**
SUPERVISOR

BRNO 2019

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav informačních systémů (UIFS) Akademický rok 2018/2019

Zadání bakalářské práce

Student: **Leonov Martin**
Program: Informační technologie
Název: **Modul pro transformaci dat pro digitální úřední desky**
Data Transform Module for Digital Official Boards
Kategorie: Informační systémy

Zadání:

1. Nastudujte a porovnejte programová rozhraní (API) nejpoužívanějších spisových služeb a dalších zdrojů dat pro digitální úřední desku včetně strategií pro získávání takových dat.
2. Navrhněte univerzálně použitelný strukturovaný formát dat pro použití v aplikaci pro digitální úřední desky (např. v XML, JSON, YAML) včetně příslušného schéma pro validaci dat v daném formátu. Navrhněte způsob transformace dat z vybraných spisových služeb do Vámi navrženého formátu včetně možnosti rozšíření o další spisové služby a jiné zdroje dat.
3. Po konzultaci s vedoucím implementujte navržený formát a transformace ve vhodné technologii. Ověřte řešení pomocí automatizovaných testů a poskytněte programovou dokumentaci.
4. Navrhněte a realizujte vhodný způsob průběžné integrace a nasazení (CI/CD) řešení pomocí moderních nástrojů (např. Docker).
5. Výsledky zdokumentujte, zhodnoťte a navrhněte možná rozšíření.

Literatura:

- Kathy Andersen, Mitchell Barry, Dave Burchell, Lora Mae Frecks, Jay Hannah, Noah Koch, Ryan Walker, and Cody Winchester. 2014. Local boards web service API: accessible local and regional boards data. In Proceedings of the 15th Annual International Conference on Digital Government Research (dg.o '14). ACM, New York, NY, USA, 343-344. [https://doi.org/10.1145/2612733.2612793]
- Irena Holubová, Jaroslav Pokorný. XML technologie: principy a aplikace v praxi. Praha: Grada, 2008. Průvodce (Grada). ISBN 978-80-247-2725-7.
- MV ČR. Metodický návod pro vedení elektronické úřední desky v územních samosprávných celcích. [http://www.mvcr.cz/clanek/metodicke-pomucky-ke-spravnimu-radu.aspx]
- Robert C. Martin. Clean code: a handbook of agile software craftsmanship. Pearson Education, 2009. ISBN 978-0-13-235088-4.

Podrobné závazné pokyny pro vypracování práce viz http://www.fit.vutbr.cz/info/szz/

Vedoucí práce: **Rychlý Marek, RNDr., Ph.D.**
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2018
Datum odevzdání: 15. května 2019
Datum schválení: 16. října 2018

Abstrakt

Cílem této práce bylo navrhnout a implementovat aplikaci umožňující univerzální přístup k datům ze systémů elektronické spisové služby a jejich následnou transformaci do jednotného formátu. Ze získaných poznatků o těchto systémech byly navrženy rozhraní pro stahování, ukládání a transformaci dat. Výsledky této práce poskytují uživatelskou konfiguraci a jednoduchou integraci nových zdrojů dat. Funkčnost byla ověřována nástroji pro průběžnou integraci a průběžné nasazení.

Abstract

The goal of this thesis was creation of application that will allow a universal way of getting data from electronic document and record management systems and their transformation to unified designed format. Interfaces for downloading, storing and transforming data were designed from knowledge of these systems. Created solution offers user configuration and easy way to add new data sources. Functionality was verified with using tools for continuous integration and continuous delivery.

Klíčová slova

Digitální úřední deska, elektronicky systém spisové služby, go, golang, zerolog, sentry, CI/CD, průběžná integrace, průběžné nasazení, průběžná integrace a průběžné nasazení

Keywords

Digital Official Boards, document Record Management System, go, golang, zerolog, sentry, CI/CD, continuos integration, continuos delivery, continuous integration and continuous delivery

Citace

LEONOV, Martin. *Modul pro transformaci dat pro digitální úřední desky.* Brno, 2019. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce RNDr. Marek Rychlý, Ph.D.

Modul pro transformaci dat pro digitální úřední desky

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana RNDr. Marka Rychlého, Ph.D. Další informace mi poskytli Ing. Oldřich Ešner, DiS. a Ing. Petr Mikušek. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Martin Leonov
14. května 2019

Poděkování

Tímto bych chtěl poděkovat vedoucímu práce RNDr. Markovi Rychlému, Ph.D. a externímu zadavateli Ki-Wi Digital s.r.o. Dále bych chtěl poděkovat Ing. Oldřichovi Ešnerovi, DiS za odborné vedení práce a Ing. Petrovi Mikuškovi za poskytnutí odborných znalosti týkající se programovacího jazyka Golang a napojení na API poskytovatelů spisových služeb.

Obsah

Kapitola 1

Úvod

V současné době je ve státní správě kladen stále větší důraz na digitalizaci a automatizaci rutinních činností. Proto města či obce upouštějí od klasických (papírových) úředních desek, kde je časově náročné vyvěšení nových a svěšení starých dokumentů. Pro tyto účely začaly vznikat digitální úřední desky, které bývají součástí webových stránek města či obce nebo jako samostatně stojící dotykové zařízení před budovou obecního úřadu. Digitalizaci úřadu zajišťuje informační systém tzv. elektronický systém spisové služby, který nabízí různé moduly pro obvyklou práci s dokumenty. K vyvěšení nových a svěšení starých dokumentů z úřední desky dochází automaticky. V této práci bude využíván modul úřední desky. Přístup k dokumentům je realizován pomocí aplikačního rozhraní tzv. API. To umožňuje napojení aplikacím třetích stran, které následně mohou, např. vyvěšené dokumenty ve spisové službě, zobrazit občanům pomocí dotykového zařízení.

Motivací vzniku této práce, ve spolupráci s firmou Ki-Wi Digital s.r.o., je vytvoření aplikace sjednocující různé přístupy a filozofie k získávání dokumentům z úředních desek prostřednictvím poskytovaného API. Díky využití tohoto řešení bude snadnější vyhovět požadavkům libovolného zákazníka bez ohledu na používanou úřední desku nebo jiný zdroj dat.

Cílem této práce je vyvinout jednorázově spouštěnou aplikaci, která zajistí synchronizaci aktuálně vyvěšených dokumentů a umožňující uživatelskou konfiguraci přes parametry příkazové řádky anebo proměnné prostředí. Pro snadnou možnost rozšiřování o napojení na nové poskytovatele spisových služeb nebo jiné zdroje dat, musí být rozhraní navržena dostatečně obecně tak, aby toto umožňovala. Získané dokumenty aplikace transformuje do navrženého strukturovaného formátu, který poslouží dalším aplikacím jako zdroj dat. Při nasazení v produkčním prostředí je kladen důraz na logování a automatické hlášení vzniklých chyb. Správnost řešení je ověřována automaticky spouštěnými testy, které budou integrovány pomocí nástrojů průběžné integrace a nasazení.

Práce bude členěna do několika kapitol. Druhá kapitola 2 popíše a porovná systémy elektronické spisové služby a jimi využívané komunikační protokoly pro získávání dat pomocí aplikačnímu rozhraní. Kapitola 3 bude věnována návrhu architektury aplikace, synchronizačního algoritmu a strukturovaného výstupního formátu. V kapitole 4 bude popsána implementace navrženého řešení a k tomu využité technologie. Předmětem kapitoly 5 bude použití nástrojů pro průběžnou integraci a průběžné nasazení, integrace logování a automatického hlášení chyb.

Kapitola 2

Systémy elektronické spisové služby

Tato kapitola popíše systémy elektronické spisové služby a vzájemné rozdíly mezi poskytovateli. Na konci budou zmíněny komunikační protokoly využívané těmito systémy.

2.1 Elektronický systém spisové služby

Elektronický systém spisové služby (ESSL) je informační systém pro odbornou správu dokumentů a pro elektronické vedení spisové služby. Takový program musí zajišťovat evidenci elektronických i listinných dokumentů. Pojem odpovídá anglickému označení "Electronic Document and Record Management System", EDRMS.

ESSL musí zejména:

- *evidovat příjem, zařazení, označení, rozdělení, předání a projednání dokumentů,*
- *zaznamenávat informace o všech operacích s dokumentem (transakční záznam),*
- *spravovat metadata o dokumentech.*

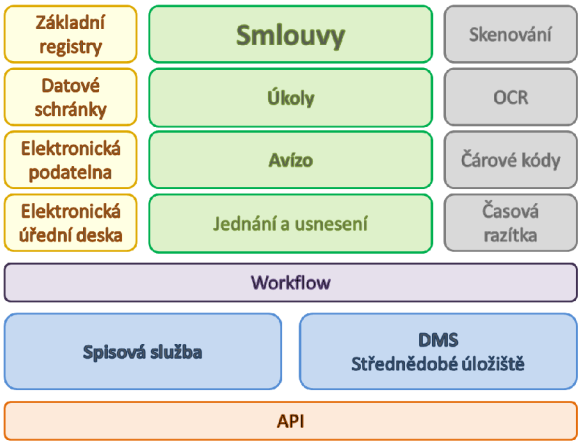
Podrobnosti funkcí elektronického systému spisové služby stanovuje zejména:

- *Zákon č. 499/2004 Sb., o archivnictví a spisové službě, ve znění zákona č. 190/2009 Sb.,*
- *Vyhláška č. 259/2012 Sb., o podrobnostech výkonu spisové služby,*
- *Národní standard pro elektronické systémy spisové služby (NSESSS), vyhlášený Ministerstvem vnitra dle zmocňovacího ustanovení Zákona o archivnictví a spisové službě – jedná se o adaptovaný a přeložený mezinárodní standard MoReq2.*

Tato podkapitola byla převzata z [34].

2.2 Poskytovatelé

Na trhu existuje velké množství poskytovatelů systémů elektronické spisové služby. Někteří z nich poskytují vlastní řešení, jiní nabízejí aplikaci agregující různé poskytovatele do jedné. Produkt spisové služby je běžně tvořen z více modulů, které nabízejí specifické funkce.



Obrázek 2.1: Struktura produktu Spisová služba od firmy GEOVAP, spol. s r.o. Obrázek v převzat z [7].

Provozovatel	**Produkt**	**Komunikační protokol**
GEOVAP, spol. s r.o.	Spisová služba	SOAP
GORDIC spol. s. r.o.	GINIS	SOAP
Galileo Corporation s.r.o.	-	JSON-RPC 2.0
ICZ a.s.	e-spis	SOAP
SoftHouse, s. r. o.	EZOP	SOAP
MARBES CONSULTING s. r. o.	XSpis	SOAP

Tabulka 2.1: V tabulce je seznam firem poskytujících, jako jeden ze svých produktů, systémy elektronické spisové služby. V posledním sloupci jsou uvedeny jimi využívané komunikační protokoly.

Například na obrázku 2.1 je struktura produktu Spisová služby od firmy GEOVAP s.r.o. Mezi běžně poskytované moduly patří:

- základní registry,
- elektronická podatelna,
- datové schránky,
- elektronická úřední deska.

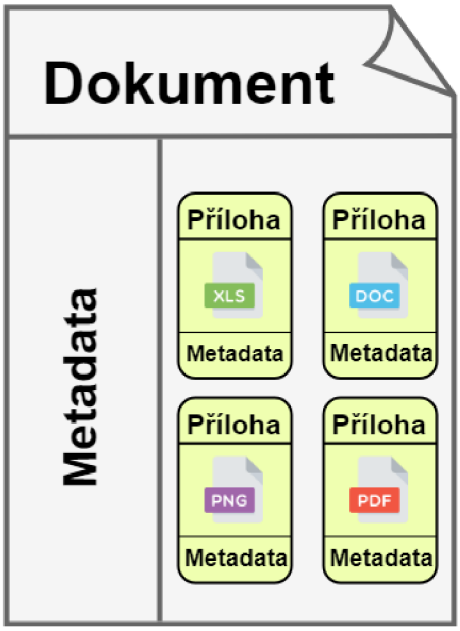
V tabulce 2.1 jsou zmíněni někteří poskytovatelé spisových služeb .

2.3 Elektronická úřední deska

V této práci se používá rozhraní tohoto modulu pro přístup k aktuálně vyvěšeným dokumentům a jich přílohám. Z úřední desky je možné získat 0–n dokumentů, kde n je celé nezáporné číslo. Struktura dokumentu je graficky znázorněna na obrázku 2.2 a skládá se z:

- **Metadata** v sobě zahrnují informace o dokumentu, jakými jsou např. název, kategorie, datum vyvěšení/svěšení, autor...

- **Přílohy** v sobě zahrnují své metadata, mezi které patří např. název souboru, název na úřední desce a binárních data většinou v kódování base64. Příloh u dokumentu může být 1–n, kde n je celé kladné číslo.



Obrázek 2.2: Struktura dokumentu se dělí na metadata dokumentu a na přílohy. Přílohy mají také vlastní metadata. Ikona XLS, DOC, PNG a PDF převzaty z [22], [19], [21], [20].

2.4 Komunikační protokoly

Komunikační protokol je soubor syntaktických a sémantických pravidel, který umožňuje komunikaci mezi dvěma nebo více zařízeními. Doporučená implementace protokolů je popsána dokumenty RFC.

2.4.1 HTTP

Jedním z nepoužívanějších protokolů při komunikaci v internetu je HTTP. Ten je určen pro komunikaci na principu dotaz-odpověď. Jedná se o textový protokol, který komunikuje pomocí TCP na portu 80. Tento protokol je definován dokumentem RFC 2616 [6].

Dotaz (Request)

Příklad odesílaného dotazu je na výpisu 2.1. Dotaz je tvořen následující strukturou:

- Na prvním řádku je uvedena použitá metoda dotazu, která je oddělená mezerou od lokalizace zdroje na serveru. Mezerou je také oddělená použitá verze protokolu HTTP. Metody, které budou v této práci práci používané jsou následující:
  - **GET** metoda slouží k získávání dat ze serveru na základě zadané URI. Tato metoda žádným způsobem neovlivňuje získávaná data.

  - **POST** metoda slouží k odesílání dat na server. Odesílaná data jsou uváděny v těle požadavku. Typ a délka odesílaných dat jsou v hlavičce uvedeny v položkách Content-Type a Content-Length.

- Další řádky jsou určeny pro zadání položek hlaviček dotazu. Obsahem hlaviček může být např.:
  - **Host** určuje adresu serveru, na kterou byl dotaz odeslán.
  - **Content-Type** určuje typ obsahu, který je posílán metodou POST v těle zprávy.
  - **Content-Length** určuje velikost (v bajtech) odesílaných dat v těle požadavku.
- Prázdný řádek od sebe odděluje hlavičku a tělo zprávy.
- Tělo zprávy nemusí být zadáno, slouží pro zadání zprávy, která má být odeslána na server. Používá se hlavně s použitím metody POST.

```
POST /api/official_boards/ HTTP/1.1
Content-Length: 93
Host: localhost:8080
Content-Type: text/json

{"jsonrpc":"2.0","id":2,"method":"getCategories","params":{boardId":4}}
```

Výpis 2.1: Ukázka HTTP POST dotazu na zdroj /api/official_boards/ odeslaného na server http://localhost:8080/.

Odpověď (Response)

Příklad odesílané odpovědi ze serveru je na výpisu 2.1. Odpověď má následující strukturu:

- Na prvním řádku jsou obsaženy informace o stavu odeslané odpovědi. Jako první se uvádí verze použitého protokolu HTTP, za kterou následuje stavový kód se slovním popisem typu odpovědi, např. HTTP/1.1 201 Created.
- Další řádky slouží, stejně jako HTTP dotazu, pro uvedení položek v hlavičce. Položky které mohou být vyplněné jsou např. následující:
  - **Date** určuje datum kdy byla odpověď vytvořena.
  - **Last-Modified** určuje datum poslední modifikace odesílaného obsahu.
  - **Content-Type** určuje typ přenášených dat v těle zprávy.
- Prázdný řádek od sebe odděluje hlavičku a tělo zprávy.
- Tělo odpovědi, ve kterém server posílá zprávu klientovi. Tělo není povinné a může být prázdné.

```
HTTP/1.1 200 OK
Date: Wed, 1 May 2019 15:22:23 GMT
Content-Type: text/json
Content-Length: 117
Last-Modified: Wed, 1 May 2019 15:20:00 GMT

{
  "jsonrpc": "2.0",
  "id": 2,
  "result": [
    {"category_id": 1,"board_id": 4,"name": "název","document_count": 2}
  ]
}
```

Výpis 2.2: Příklad serverem odeslané HTTP odpovědi na dotaz z příkladu 2.1. Návratový kód 200 označuje úspěch.

2.4.2 SOAP

SOAP (Simple Object Access Protocol) je protokol pro výměnu strukturovaných informací přes protokol HTTP. Informace jsou strukturovány pomocí formátu XML. Rozhraní webové služby, která tento protokol využívá je popsán jazykem WSDL[1]. Tento protokol je definován v dokumentu RFC 4227 [24].

Struktura zprávy

Příklad SOAP zprávy je na výpisu 2.3. Jejím kořenovým elementem je `<Envelope>`, který identifikuje XML dokument jako SOAP zprávu. Jeho potomky mohou být elementy:

- **<Header>** element není povinný, jeho obsahem mohou být např. autentizační údaje nebo jiné informace vyžadované webovou službou.
- **<Body>** element je povinný, jeho obsahem jsou informace odesílané z nebo na server. Při odesílání na server může obsahovat např. operace, které se mají provést, při odesílání zprávy ze serveru obsahuje výsledek provedené operace.
- **<Fault>** element není povinný, jeho obsahem jsou podrobnosti o vzniklé chybě, která nastala při provádění operace.

2.4.3 JSON-RPC

JSON-RPC (JavaScript Object Notation - Remote Procedure Call) je protokol pro volání vzdálených procedur ve formátu JSON. Jeho struktura je to velmi jednoduchá, protože definuje pouze několik základních příkazů a datových typů. Použití není závislé na zvoleném komunikačním protokolu, proto jeho zpracování může být stejné přes sokety, HTTP nebo jiný princip zasílání zpráv. Klient při komunikaci vytváří objekty požadavků, které definují vykonávanou akci. Na tyto požadavky server reaguje tak, že zaslaný příkaz zpracuje, vytvoří objekt odpovědi a ten odešle zpět. Tento protokol podporuje odesílání hromadných

[1] Web Services Description Language https://www.w3.org/TR/2001/NOTE-wsdl-20010315

```
<soap:Envelope>
  <soap:Header>
    <Authentication>
      <Username>name</Username>
      <Password>password123</Password>
    </Authentication>
  </soap:Header>
  <soap:Body>
    <getDocuments>
      <desk>MBBVYZWOOA021</desk>
    </getDocuments>
  </soap:Body>
</soap:Envelope>
```

Výpis 2.3: Příklad struktury SOAP dotazu. V hlavičce jsou vyplněny autentizační údaje. Tělo pak obsahuje operaci, která se má provést.

požadavků. Rozdíl oproti běžným příkazům je ten, že je odesíláno pole objektů požadavků, které by byly odesílány jednotlivě. Na hromadné požadavky server odpovídá polem, jehož obsahem jsou objekty odpovědí. [12]

Objekt požadavku

Příklad objektu požadavku, který klient odesílá na server 2.4. Objekt musí mít následující položky:

- **jsonrpc** označuje verzi JSON-RPC protokolu.
- **method** určuje název příkazu, který má být proveden.
- **params** obsahuje strukturovaný objekt, který nese hodnoty, které budou použity při provádění příkazu
- **id** je klientem vytvořený identifikátor požadavku, na který musí server odpovědět stejnou hodnotou. Pokud není hodnota uvedena, jedná se o notifikaci na kterou server neodesílá odpověď.

```
{
  "jsonrpc":"2.0",
  "method":"getDocuments",
  "params":{
    "clientId":"testest"
  }
  "id":4,
}
```

Výpis 2.4: Příklad objektu požadavku pomocí protokolu JSON-RPC.

Objekt odpovědi

Příklad odpovědi vygenerované serverem je na výpisu 2.5. V odpovědi musí být obsaženy následující položky:

- **jsonrpc** označuje verzi JSON-RPC protokolu.
- **result** je struktura ve které se nachází data při úspěšně vykonaném příkazu. Tato položka není povinná, pokud nastala chyba při zpracovávání požadavku.
- **error** je struktura ve které se nachází informace o vzniklé chybě. Tato položka je vyžadována pouze, pokud se vyskytla chyba. Struktura je popsána ve specifikaci [12].
- **id** musí být stejné jako klient uvedl v odesílaném požadavku.

```
{
  "jsonrpc": "2.0",
  "result": [
    {
      "category_id": 1,
      "board_id": 4,
      "name": "ztráty a nálezy",
      "document_count": 2
    }
  ],
  "id": 2,
}
```

Výpis 2.5: Příklad objektu odpovědi odeslané serverm při použití protokolu JSON-RPC.

Kapitola 3

Návrh řešení

Tato kapitola definuje požadavky potřebné pro návrh a následnou implementaci aplikace. Další část představí architekturu aplikace 3.2, která definuje samostatné logické celky. Následně každý z těchto celků bude popsán samostatně. Předposlední podkapitola popíše synchronizační algoritmus, který zajišťuje správnou synchronizaci nových a odstranění starých dokumentů z úložiště. Samotný konec kapitoly popíše požadavky a návrh výstupního strukturovaného formátu 3.4.

3.1 Požadavky na aplikaci

Požadavky na funkcionalitu aplikace popíšeme formou tzv. user stories[1]. Pro jejich vytvoření je potřeba definovat role a jejich požadavky na funkcionalitu, která něco vylepší. [17]

Při návrhu aplikace byly definovány čtyři základní role a jejich požadavky:

1. **Elektronický systém spisové služby**
    - úsporné volání API, aby nedocházelo ke zbytečnému vytěžování serveru
2. **Integrátor**
    - jazyk používající statickou typovou kontrolu, aby nedocházelo k běhovým chybám spojených s přístupem k neexistujícím vlastnostem nebo funkcím
    - jednoduché a dokumentované API, aby implementace nového zdroje dat nebyla delší než je nezbytně nutná
    - použití CI/CD, aby došlo k automatizaci procesů při testování, sestavení a vystavení na aktualizační server
3. **Zadavatel**
    - multiplatformní jazyk, aby bylo možné spustit aplikace na Windows (koncové zařízení) i Linuxu (CI/CD)
    - jednoduchá rozšiřitelnost o další zdroje dat tak, aby byl pokryt co největší počet zákazníků
    - integrace původního výstupního formátu, aby bylo možné napojení na starší verze webové aplikace pro digitální úřední desky

[1] Je způsob zápisu funkcionality vyvíjené aplikace v přirozeném jazyku [35]

- integrace spisových služeb Galileo, Ginis, Geovap, ICZ, aby bylo možné provést migraci starších zařízení na nový způsob integrace
- pojmenování spuštěné instance programu, aby bylo možné přesně lokalizovat nasazení
- vytváření logů, aby byly jasné kroky, které vedly k vyvolané chybě
- hlášení vzniklých chyb, abychom jsme se o chybách dozvěděli dřív než zákazník

4. **Zákazník**

   - dostupnost protokolu (log) událostí o vyvěšených a svěšených dokumentech, abych měl přehled o aktuálně vyvěšených dokumentech

3.2 Architektura aplikace

Architektura aplikace je logicky rozdělena do několika základních částí. Prvním částí je konfigurace, která může být nastavena přes parametry příkazové řádky nebo proměnné prostředí a následně je předána vstupnímu bodu. Ten zahájí synchronizační algoritmus 3.3, při kterém je využíváno rozhraní Fetch pro získání dat a rozhraní Store pro jejich uložení. Z uložených dat jsou rozhraním Output vygenerovány požadované výstupní formáty. Celý tento proces je graficky znázorněn na obrázku 3.1.

3.2.1 Objektový návrh

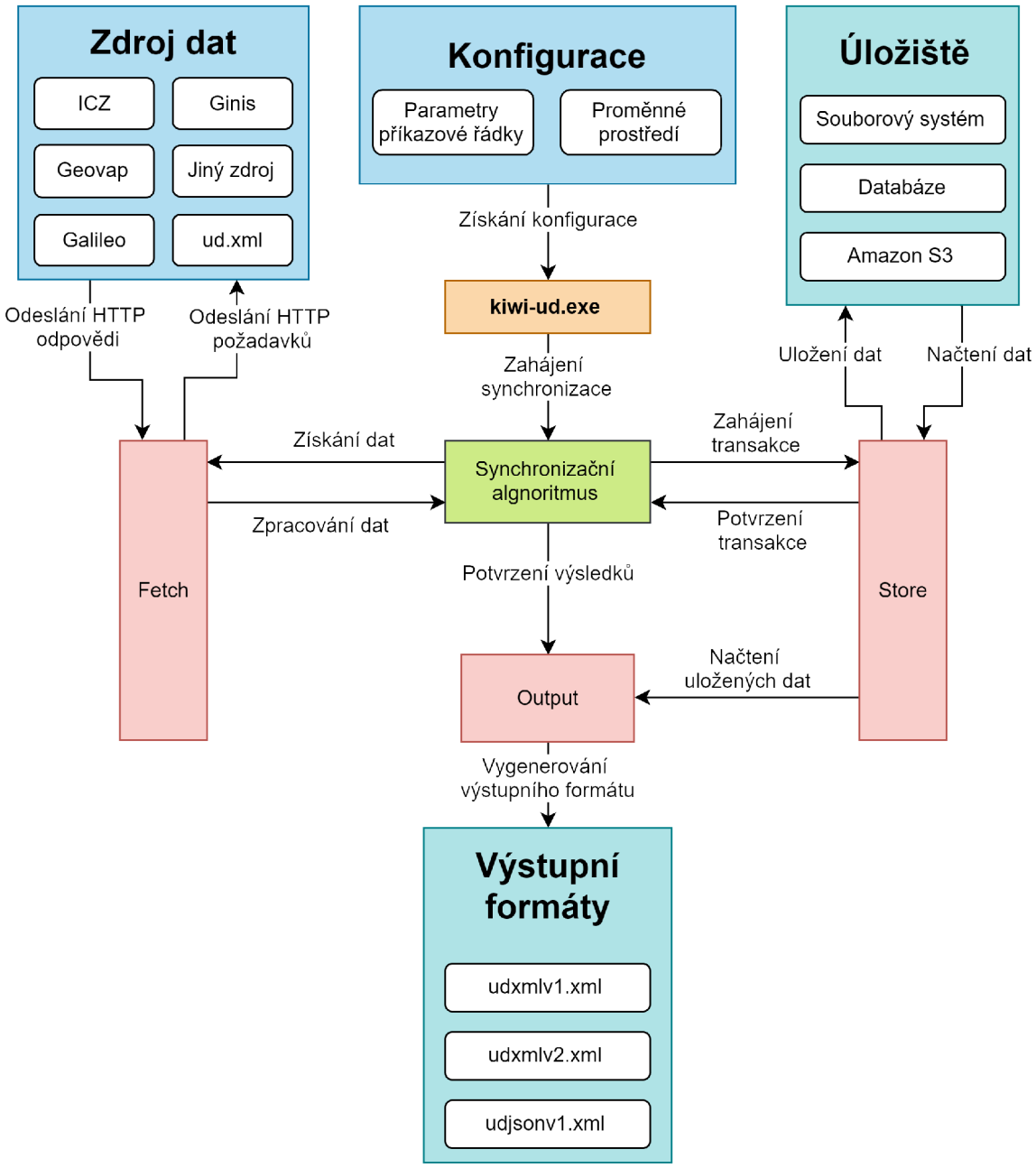
Součástí architektury je objektový návrh, který rozšiřuje pohled na zmíněná rozhraní Fetch, Store a Output a jejich závislosti. Při návrhu bylo vycházeno z definovaných požadavků na funkčnost a rozšiřitelnost. Pro vytvoření objektového návrhu bylo použito UML syntaxe. Z návrhu na obrázku 3.2 je patrné, že rozhraní:

- **Fetch** umožňuje načítání informací o dokumentech a jejich přílohách bez ohledu na zdroj dat.
- **Store** umožňuje transakční ukládání a díky rozšíření chování z rozhraní Fetch také načítání dokumentů a příloh z úložiště.
- **Output** umožňuje na základě rozhraní Fetch pro načítání vygenerovat výstupní formát.

3.2.2 Rozhraní Fetch

Prvním hlavním rozhraním je `Fetch`, jehož hlavním úkolem je načítání metadat a příloh dokumentu. Toto rozhraní je využito jak pro načítání dat ze spisových služeb, tak i z úložiště. Proto, aby bylo možné získat zmíněná data, je potřeba definovat další rozhraní, které budou toto umožňovat.

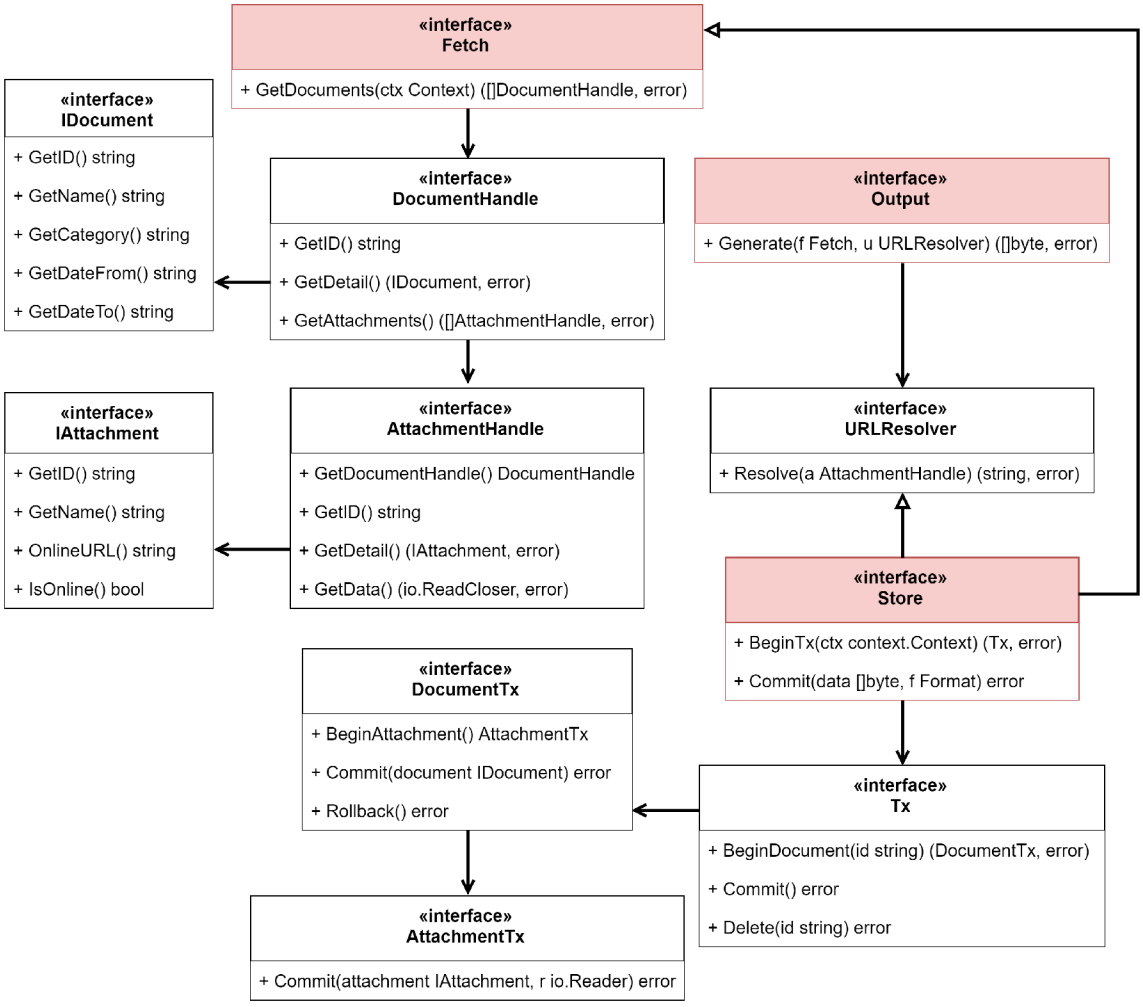
Jedinou dostupnou metodou v tomto rozhraní je `GetDocuments()`. Jejím hlavním úkolem je získání seznamu ukazatelů na dokumenty. Každý dokument je popsán rozhraním `DocumentHandle`. Konkrétní způsob získání seznamu ukazatelů ze spisové služby nebo jiného zdroje dat je popsán v implementaci rozhraní Fetch 4.3.



Obrázek 3.1: Znázorňuje architekturu aplikace a závislosti jednotlivých části mezi sebou. Oranžově je označen vstupní bod aplikace, který používá další barevně označené části. Zeleně je označen synchronizační algoritmus, pro synchronizaci nových dokumentů. Červeně jsou označeny základní rozhraní, která jsou detailněji popsané na objektovém návrhu na 3.2. Modře jsou označeny vstupní a tyrkysově výstupní body aplikace.

DocumentHandle

Toto rozhraní reprezentuje jeden konkrétní dokument a je vstupním bodem pro získávání detailnějších informací. V rozhraní jsou definovány metody pro získání identifikátoru, metadat a seznamu ukazatelů na přílohy dokumentu.



Obrázek 3.2: Popisuje objektový návrh základních tří rozhraní (Fetch, Store, Output) a jejich podřízená rozhraní. Červeně jsou označeny rozhraní, která odpovídají červeným částem z architektury aplikace na obrázku 3.1.

Identifikátor dokumentu lze získat z metody `GetID()`, která nese libovolný textový řetězec. Detaily dokumentu, nebo-li metadata dokumentu, kterými jsou např. název, kategorie..., jsou dostupné při použití metody `GetDetail()`, jejíž výsledkem je rozhraní `IDocument`, které popisuje nesená metadata. Každý dokument obsahuje 1–n příloh, kde n je celé číslo větší nebo rovno 1. Dostupnost příloh zajišťuje metoda `GetAttachments()`, která vrací seznam ukazatelů na přílohy. Každá příloha je popsána rozhraním `AttachmentHandle`.

IDocument

Tímto rozhraním jsou definovány metadata, která budou pro daný dokument dostupná. Přístup k nim je přes následující metody:

- GetID() - identifikátor dokumentu
- GetName() - název dokumentu
- GetCategory() - název kategorie dokumentu

- GetDateFrom() - datum vyvěšení dokumentu
- GetDateTo() - datum svěšení dokumentu

AttachmentHandle

Rozhraní `AttachmentHandle` reprezentuje jednu přílohu. Proto, aby bylo možné získat ukazatel na rodičovský dokument přílohy, je nutné definovat metodu `GetDocumentHandle()`, která toto zajistí. Identifikátor a metadata jsou, stejně jako v rozhraní DocumentHandle, přístupné metodami `GetID()` a `GetDetail()`. Metadata přílohy jsou definované rozhraním `IAttachment`.

Pokud se nejedná o online přílohu a její binární data mají být staženy, pak pro její získání je definována metoda `GetData()`, která vrací rozhraní `io.ReadCloser`[2], díky němuž umožňuje streamované načítání, tzn. soubor nemusí být celý načten do operační paměti, ale je z něj načítáno průběžně.

IAttachment

Metadata příloh jsou popsána tímto rozhraním. Následující metody definují, která metadata budou přílohy zpřístupňovat:

- GetID() - identifikátor přílohy
- GetName() - název přílohy
- IsOnline() - příznak určující, zda-li se jedná o online přílohu
- OnlineURL() - pokud příznak `IsOnline` je nastaven na hodnotu `true`, pak výsledkem bude online URL

3.2.3 Rozhraní Store

Druhým hlavním rozhraním je `Store`, které charakterizuje vlastnosti úložiště dokumentů a příloh. Z objektového návrhu 3.2 je patrné, že je rozšířeno o funkcionalitu z rozhraní `Fetch` a `URLResolver`. Díky prvně zmíněnému rozhraní lze, kromě ukládání dat, také uložená data načítat. Rozhraní `URLResolver` zde slouží jako překladač, který určuje, jak budou vytvářeny URL v rámci úložiště. Implementace rozhraní `Store`, jako souborový systém, je popsána v podkapitole 4.4.

Rozhraní nabízí metody pro vytváření transakcí `BeginTx()` a jejich úspěšné potvrzení `Commit()`. Každá transakce je popsána rozhraním `Tx`. Při potvrzování vytvořených transakcí metodou `Commit()` je vyžadováno předání pole bajtů, které popisuje úspěšně dokončenou transakci.

- **Tx** - tímto rozhraním je popsána jedna transakce, která reprezentuje seznam dokumentů. V rozhraní je definována metoda `BeginDocument()` k vytváření transakcí dokumentů. Dokument, který má být odstraněn musí být označen metodou `Delete()` k odstranění. Pro potvrzení všech provedených změn je možné metodou `Commit()`.
- **DocumentTx** - toto rozhraní popisuje metody transakce dokumentu. Tyto metody umožňují vytváření nových transakcí příloh `AttachmentTx`. K tomu je určená metoda

[2] io.ReadCloser je součástí standardní knihovny jazyka Go https://golang.org/pkg/io/#ReadCloser.

`BeginAttachment()`. Úspěšné dokončení transakcí lze potvrdit metodou `Commit()`, které jsou předány metadata dokumentu `IDocument`. V případě, že nastane neočekávaný stav, lze provedené změny vrátit zpět metodou `Rollback()`.

- **AttachmentTx** - posledním rozhraním v celé hierarchii je `AttachmentTx`. Jeho jediným úkolem je ukládání příloh do úložiště metodou Commit(). Ta pro uložení přílohy vyžaduje metadata přílohy `IAttachment` a stream dat jako `io.ReadCloser`.

3.2.4 Rozhraní Output

Posledním z logických celků je rozhraní `Output`. To obsahuje jedinou metodou `Generate()`, jejímž úkolem je, z předané implementace rozhraní `Fetch` a `URLResolver`, vygenerovat navržený výstupní formát 3.4. Rozhraní `URLResolver` je vyžadováno kvůli možnosti získání URL adresy lokalizující přílohu v úložišti. Přesná podoba URL je dána na konkrétní implementaci tohoto rozhraní. Implementace rozhraní `Output` je popsána v podkapitole 4.5.

3.3 Synchronizační algoritmus

Synchronizační algoritmus slouží k rozhodování o tom, které dokumenty byly nově vyvěšeny nebo svěšeny. Nově vyvěšené dokumenty budou uloženy a ty svěšené odstraněny. Algoritmus vyžaduje instance implementující rozhraní `Fetch` a `Store`. První instancí poskytuje napojení na API spisové služby nebo jiný zdroj dat 4.3. Druhá instance poskytuje načítání lokálně uložených dat, ale také zprostředkovává ukládání těch nových. Fungování algoritmu je popsána pseudokódem na výpisu 3.1.

Pseudokód

```
// Získej všechny lokálně uložené dokumenty - []DocumentHandle
localDocs := Store.GetDocuments()
// Získej dokumenty ze spisové služby - []DocumentHandle
remoteDocs := Fetch.GetDocuments()
// Proveď rozdíl množin dokumentů ze spisové služby a lokálně uložených.
// Výsledkem budou dokumenty, které byly nově přidané
newDocs := Sets.Diff(remoteDocs, localDocs)
// Proveď rozdíl dokumentů lokálně uložených a vrácených spisovou službou
// Výsledkem budou dokumenty, které byly spisovou službou svěšeny
oldDocs := Sets.Diff(localDocs, remoteDocs)

// Pro každý starý dokument proveď
foreach doc in oldDocs {
  // Získej identifikátor odstraňovaného dokumentu
  id := doc.GetID()
  // Označ dokument k~odstranění
  Tx.Delete(id)
}

// Pro každý nový dokument proveď následující
foreach doc in newDocs {
```

```
  // Zahaj transakci dokumentu
  Tx.BeginDocument()
  // Načti seznam ukazatelů na přílohy - []AttachmentHandle
  attachments := doc.GetAttachments()
  // Pro každou přílohu proveď následující
  for a in attachments {
    // Zahaj transakci přílohy
    DocumentTx.BeginAttachment()
    // Získej stream dat přílohy - io.ReadCloser
    data := a.GetData()
    // Získej metadata přílohy - IAttachment
    detail := a.GetDetail()
    // Potvrď potvrzení uložení přílohy
    AttachmentTx.Commit(data, detail)
  }
  // Pokud alespoň jedna transakce přílohy neuspěla
  if not attachmentsSucceed {
    // Proveď navrácení provedených změn
    doc.Rollback()
    // Pokračuj na další dokumentu
    continue
  }
  // Získej metadata dokumentu - IDocument
  detail := doc.GetDetail()
  // Potvrď potvrzení uložení dokumentu
  DocumentTx.Commit(detail)
}
// Potvrď úspěšné uložení dokumentů
Tx.Commit()
```

Výpis 3.1: Synchronizační algoritmus popsaný pseudokódem. Tyrkysovou barvou jsou označeny názvy použitých rozhraní a klíčová slova

3.4 Výstupní strukturovaný formát

Navrhnout výstupní formát je potřeba tak, aby splňoval všechny požadavky na obsahované informace a zároveň práce s ním byla co nejpříjemnější. Navržený formát bude sloužit jako rozhraní mezi touto aplikací a aplikací načítající informace o dokumentech. Více informací o použití tohoto formátu je popsáno v podkapitole 4.8. Informace o návrhu struktury ve formátu XML a její ověření pomocí XSD byly čerpány z [14].

Požadavky na návrh výstupního formátu

- Struktura musí umožňovat uložení 0–n dokumentů, kde n je číslo větší nebo rovno 0.
- Každý dokument musí umožňovat uložení následujících metadat:
  - Identifikátor

  - Název
  - Kategorie
  - Datum vyvěšení
  - Datum svěšení
  - Seznam příloh
- Datum vyvěšení/svěšení musí být možné vložit formátované podle definice v dokumentu RFC3339 [16].
- Datum svěšení není povinné a nemusí být zadáno.
- Každý dokument v seznamu příloh může obsahovat 1–n příloh, kde n je číslo větší nebo rovno 1.
- Každá příloha musí umožňovat uložení následujících metadat:
  - Identifikátor
  - Název
  - Příznak, detekující zda-li se jedná o online přílohu
  - URL adresa umístění přílohy

Pro vytvoření požadovaného formátu byl použit značkovací jazyk XML. Navržený formát je validován pomocí schéma XSD[3] na výpisu 3.2.

Kořenovým elementem XML struktury je element `<documents>`, jehož obsahem může být 0–n elementů s dokumenty `<document>`. To je zajištěno nastavením XSD atributů `minOccurs` na hodnotu 0 a `maxOccurs` na hodnotu `unbounded`, která znamená neomezený počet. Každý dokument je tvořen elementem `<document>`. Metadata jsou vkládány jako jeho potomci tohoto elementu. V tabulce 3.1 je mapování metadat dokumentu na XML elementy.

Metadata	**XML element**	**XSD datový typ**
Identifikátor	<id>	string
Název	<name>	string
Kategorie	<category>	string
Datum vyvěšení	<dateFrom>	dateTime
Datum svěšení	<dateTo>	dateTime
Seznam příloh	<attachments>	attachment 3.2

Tabulka 3.1: Tabulka mapující metadata dokumentů na XML elementy.

Kořenovým elementem pro vkládání příloh je `<attachments>`, který je potomkem elementu `<document>`. Vkládání příloh je omezeno vložením 1–n příloh, kde n je číslo větší nebo rovno 1. Tato povinnost je zajištěna nastavením XSD atributů `minOccurs` na hodnotu 1 a `maxOccurs` na hodnotu `unbounded`, která definuje neomezený počet elementů. Každá příloha je tvořena elementem `<attachment>`. Metadata jsou vkládány jako jeho potomci tohoto elementu. V tabulce 3.2 je mapování metadat přílohy na XML elementy.

[3]XSD definuje strukturu XML https://www.w3schools.com/xml/schema_intro.asp

Metadata	XML element	XSD datový typ
Identifikátor	<id>	string
Název	<name>	string
URL adresa	<url>	string
Příznak online přílohy	<isOnline>	boolean

Tabulka 3.2: Tabulka mapující metadata příloh na XML elementy.

```
<xs:schema xmlns="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified"
  targetNamespace="http://ki-wi.cz/kiwi-ud/1.0">
<xs:element name="documents">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="document" minOccurs="0" maxOccurs="unbounded">
        <xs:complexType>
          <xs:sequence>
            <xs:element type="xs:string" name="id" />
            <xs:element type="xs:string" name="name" />
            <xs:element type="xs:string" name="category" />
            <xs:element type="xs:dateTime" name="dateFrom" />
            <xs:element type="xs:dateTime" name="dateTo" minOccurs="0" />
            <xs:element name="attachments">
              <xs:complexType>
                <xs:sequence>
                  <xs:element name="attachment" maxOccurs="unbounded">
                    <xs:complexType>
                      <xs:sequence>
                        <xs:element type="xs:string" name="id" />
                        <xs:element type="xs:string" name="name" />
                        <xs:element type="xs:string" name="url" />
                        <xs:element type="xs:boolean" name="isOnline" />
                      </xs:sequence>
                    </xs:complexType>
                  </xs:element>
                </xs:sequence>
              </xs:complexType>
            </xs:element>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
</xs:schema>
```

Výpis 3.2: XSD pro validaci navrženého výstupního formátu. Pokud není u elementu uveden atribut `minOccurs` je automaticky použita výchozí hodnota, kterou je 1.

Kapitola 4

Popis implementace

Tato kapitola se zabývá implementací navrženého řešení z kapitoly 3. V 4.1 jsou popsány použité implementační technologie. Podkapitola 4.2 popíše strukturu aplikace. V dalších třech podkapitolách je představena implementace napojení na API konkrétní spisové služby 4.3, proces ukládání dokumentů a jejich příloh 4.4, a generování výstupního formátu 4.5. V podkapitole 4.6 jsou uvedeny možnosti konfigurace aplikace. Předposlední kapitola 4.7 je věnována ukázce použití aplikace jako více spolupracujících instancí. Na závěr bude shrnuto využití implementovaného řešení při realizaci zařízení pro digitální úřední desky 4.8.

4.1 Implementační technologie

K implementaci jsem se rozhodl použít jazyk Go. Následující dvě podkapitoly jsou věnovány zvolenému jazyk a výběru požitých knihoven. Při implementaci byly dodržovány zásady čistého kódu viz [18].

4.1.1 Programovací jazyk Go

Jazyk Go byl vytvořen ve firmě Google Inc. autory Robert Griesemer, Rob Pike, a Ken Thompson. První verze byla vydána v roce 2009, následně jeho popularita v poslední několika letech začala růst, protože mnoho firem a vývojářů, kteří doposud psali aplikace v Node.js, Python nebo Java, zjistili že použitím Go může dojít řádově vyššímu výkonu. Go je staticky typovaný jazyk překládaný do strojového kódu, jehož syntaxe vychází z jazyka C. Velmi dobře hodí pro psaní síťových nástrojů nebo serverových aplikací. [32]

Mezi populární nástroje napsané v tomto jazyce patří Kubernets[1] nebo Docker[2]. Za zmínku určitě stojí i firmy, které začali používat jazyk Go:

- Netflix
- Twitter
- Facebook
- Amazon

Celý oficiální seznam firem využívající jazyk Go je na [11].

[1] Více informací o nástroji Kubernetes na https://kubernetes.io/
[2] Více informací o nástroji Docker na https://www.docker.com/

Jazyk Go jsem zvolil, protože mně zaujala rostoucí popularita, možnosti použití a následující vlastnosti jazyka:

- statická typová kontrola,
- překlad do strojového kódu,
- multiplatformní použití,
- garbage collection,
- tzv. gorutiny[3].

4.1.2 Použité knihovny

Pro práci s knihovnami Go od verze 1.11 poskytuje nástroj Go Modules[4], který umožňuje jejich snadnější instalaci a správu. Knihovny, které byly použity při vývoji jsou následující:

- github.com/mitchellh/gox - jedná se o jednoduchý nástroj pro multiplatformní sestavení Go aplikací, který přebírá část funkcionality ze standardního sestavovacího nástroje `go build`. Při multiplatformním sestavování lze využít paralelizaci této činnosti. [13]
- github.com/jessevdk/go-flags - tato knihovna nabízí pokročilejší parser parametrů příkazové řádky, než je knihovna `flags`, která je součástí standardní knihovny jazyka Go. Díky tomu nabízí řadu užitečných funkcí jako je použití krátkého názvu parametru (-v), dlouhého názvu parametru (--verbose), generovanou nápovědu programu, použití proměnných prostředí jako výchozí hodnoty parametrů a mnoho dalších[5]. [15]
- github.com/rs/zerolog - knihovna Zero Allocation JSON Logger poskytuje rychlý a jednoduchý způsob vytváření logů ve formátu JSON [26]. Vytváření logů aplikacemi je popsané v podkapitole 5.2.
- github.com/getsentry/raven-go - jedná se o oficiální Go SDK pro odesílání událostí nebo chybových stavů na službu Sentry 5.2.2. [5]
- github.com/google/renameio - nabízí způsob, jak atomicky vytvořit, přepsat soubor nebo symbolický odkaz. [31]
- github.com/h2non/filetype - dokáže určit typ souboru z prvních 262 bajtů. Využívá se pro správnou detekci stahovaného souboru. [33]
- github.com/pkg/errors - oproti knihovně `errors` ze standardní knihovny jazyka Go, rozšiřuje práci s chybovými stavy. Umožňuje např. přidání dalšího kontextu k vzniklé chybě. [4]
- github.com/deckarep/golang-set - přidává do jazyka Go datový typ množina. Množinové operace se používají v synchronizačním algoritmu 3.3 pro filtrování dokumentů. [3]

[3] Gorutiny jsou méně paměťově náročná vlákna, které jsou řízeny běhovým prostředím Go ale méně paměťově náročné [28].

[4] Více informací o Go Modules na https://github.com/golang/go/wiki/Modules.

[5] Seznam všech podporovaných funkcí je na https://godoc.org/github.com/jessevdk/go-flags

4.2 Strukturování projektu

Při vytváření adresářové struktury této aplikace bylo využito způsobů popsaných v projektu Standard Go Project Layout[6]. Nejedná se o oficiální standard vytvoření vývojářským týmem Go, nicméně obsahuje strukturu projektu typickou pro jazyk Go [27].

```
cmd
└── kiwi-ud
    └── main.go
internal
└── kiwi-ud
test
pkg
├── fetch
├── model
├── output
└── store
```

Hlavním adresářem je adresář `/cmd`, v jehož podadresářích jsou definovány vstupní body různých variant aplikace. Každý z těch vstupních bodů lze sestavit jako samostatnou aplikaci. Dalším adresářem je `/internal`, jehož podadresáře jsou stejné jako u `/cmd`. Jejich obsahem je definice a implementace metod, používaných pouze daným vstupním bodem. Předposledním adresářem je /test obsahující obsahuje testovací data používaná spouštěnými testy. Posledním adresářem je `/pkg`, kde jsou veřejné definice modelů, rozhraní a jejich implementací, které mohou být importovány jinými projekty.

4.3 Napojení na API spisové služby

V rámci práce došlo k implementaci následujících poskytovatelů:

- Galileo
- Geovap
- Ginis
- ICZ

Kromě výše zmíněných poskytovatelů, byl implementován původní i nově navržený výstupní formát 3.4, jako zdroj dat. Implementace výstupních formátů, jako zdroje dat, je zvlášť užitečná pro použití, které je na příkladu 4.7, na kterém je ukázka více aplikací stahujících data z centrálního serveru na lokální síti.

Pro ukázku implementace rozhraní `Fetch` byla zvolena spisová služba Galileo. Postup, při implementaci jiných poskytovatelů spisových služeb je pouze o rozdílném adaptování poskytovaného API na informace vyžadované implementovanou metodou.

Implementovaná spisová služba využívá ke komunikaci protokol JSON-RPC. Každá zpráva na server je odesílána jako POST požadavek, v jehož těle je struktura JSON popisující příkaz, který má být proveden. Každý odesílaný požadavek ve struktuře JSON vyžaduje

[6] Více o projektu na https://github.com/golang-standards/project-layout#standard-go-project-layout

vyplnění klíčů `clientId` pro autentizaci uživatele, `boardId` pro identifikaci úřední desku a `id`, které identifikuje odesílaný požadavek, pokud není tento klíč vyplněn, server na něj neodpovídá. Ukázka odesílané struktury JSON je na příkladu 4.1.

Struktura Fetch

Rozhraní `Fetch` je implementováno strukturou `Fetch` z balíčku `/pkg/fetch/galileo`. Díky rozdílnému názvu balíčku pro strukturu (galileo) a rozhraní (fetch) nedochází ke kolizi názvů. Rozhraní definuje pouze metodu `GetDocuments()`, která je pro získání seznamu ukazatelů na dokumenty `DocumentHandle`.

Aby při implementaci metody `GetDocuments()` mohly být získány všechny aktuálně vyvěšené dokumenty, je nejdříve potřeba získat seznam kategorií a následně všechny dokumenty spadající do získané kategorie. Spojením těchto výsledků budou získány všechny dokumenty. K získání kategorií slouží příkaz `getCategories`, jehož ukázka je na příkladu 4.1. Pokud server úspěšně provede zaslaný příkaz, pak získaná odpověď je na příkladu 4.2.

```
{
  "jsonrpc":"2.0",
  "id":2,
  "method":"getCategories",
  "params":{
    "clientId":"testest",
    "boardId":4
  }
}
```

Výpis 4.1: Příklad příkazu pro získání všech kategorií s použití protokolu JSON-RPC. Odpověď na tento příkaz je na výpisu 4.2.

```
{
  "jsonrpc": "2.0",
  "id": 2,
  "result":
    [{
      "category_id": 1,
      "board_id": 4,
      "name": "ztráty a nálezy",
      "document_count": 2
    }]
}
```

Výpis 4.2: Příklad odpovědi na příkaz `getCategories` z příkladu 4.1.

Pro získání dokumentů dané kategorie se používá příkaz `getDocuments`. Aby nedocházelo ke zbytečnému odesílání požadavků na server, lze agregovat více příkazů do jednoho HTTP požadavku. To je užitečné při získávání všech dokumentů pro každou kategorii. Příklad takového požadavku je na 4.3. Odpovědí na odeslaný požadavek je struktura na výpisu 4.4. Z této odpovědi jsou dostupné všechny informace pro vytvoření seznamu ukazatelů na

dokumenty, který je návratovou hodnotou této funkce. Protože kromě identifikátoru byly získány i metadata dokumentu, tak je možné jejich uložení do vnitřního stavu, aby dále nedocházelo ke zbytečnému načítání informací, které jsou již k dispozici.

```
[{
  "jsonrpc":"2.0",
  "id":4,
  "method":"getDocuments",
  "params":{
    "clientId":"testest",
    "boardId":4,
    "categoryId":1
  }
}]
```

Výpis 4.3: Příklad příkazu JSON-RPC pro získání všech dokumentů v kategoii s identifikátorem 1. Odpověď na tento příkaz je na výpisu 4.4.

```
[{
  "jsonrpc": "2.0",
  "id": 4,
  "result":
    [{
      "document_id": 1,
      "board_id": 4,
      "category_id": 1,
      "title": "testovací dokument",
      "post_date": "2016-07-19",
      "remove_date": "2016-08-04",
      "number": "13131245123",
      "only_electronic": false,
      "perex": "ttesttetet",
      "author": "Super Admin",
      "text": "<p>obsah dokumentu test</p>"
    }]
}]
```

Výpis 4.4: Příklad odpovědi odeslané serverem na příkaz `getDocuments` z příkladu 4.3.

Struktura DocumentHandle

Struktura DocumentHandle implementuje stejnojmenné rozhraní. Stejně jako u struktury `Fetch` nemůže dojít ke kolizi názvů, protože rozhraní a struktura se nacházejí v odlišných balíčcích. Aby bylo rozhraní splňováno, je potřeba implementovat tyto metody:

- `GetID()`
- `GetDetail()`

- `GetAttachments()`

Implementace metody `GetID()` vyžaduje konverzi identifikátoru, který je celočíselného datového typu, na řetězec. Převedená hodnota je návratovou hodnotou této funkce. Díky tomu, že metadata byly získány již z předchozího požadavku odeslaného na server, bude při implementaci metody `GetDetail()` stačit provést mapování vlastnosti uložených ve vnitřním stavu struktury na vlastnosti rozhraní `IDocument`, které je návratovou hodnotou této funkce.

Získat přílohy dokumentu lze po implementaci metody `GetAttachments()`. Protože informace nejsou dostupné, z předchozího požadavku odeslaného na server, je nutné je získat odesláním příkazu `getAttachments`. Pro získání příloh je přidán identifikátor dokumentu jako další parametr odesílaného příkazu. Příklad takového požadavku je na výpisu 4.5. Po úspěšném zpracování požadavku serverem je přijata odpověď na výpisu 4.6. Z informací v odpovědi je vytvořen seznam instancí ukazatelů na přílohy `AttachmentHandle`, který je návratovou hodnotou implementované funkce. Protože mezi informacemi o příloze nejsou uvedené identifikátory přílohy, je každé příloze přiřazen identifikátor podle aktuálního indexu v poli.

```
{
  "jsonrpc":"2.0",
  "method":"getAttachements",
  "params":{
    "clientId":"testest",
    "boardId":4,
    "categoryId":1,
    "documentId":3
  },
  "id":5
}
```

Výpis 4.5: Příklad příkazu pro získání všech příloh pro dokument s identifikátorem 3 při použití protokolu JSON-RPC. Odpověď na tento příkazu je na výpisu 4.6.

Struktura AttachmentHandle

Poslední strukturou je `AttachmentHandle`, která reprezentuje jednu přílohu a stejně jako v případě obou výše používaných struktur, implementuje stejnojmenné rozhraní. Rozhraní nabízí k implementaci následující metody:

- `GetDocumentHandle()`
- `GetID()`
- `GetDetail()`
- `GetData()`

Implementace metody `GetDocumetHandle()` vrací, z vnitřního stavu této struktury, ukazatel na rodičovský dokument, který byl nastaven v metodě `GetAttachments()` při vytváření nové instance struktury `AttachmentHandle`. Při implementaci metody `GetID()`

```
{
  "jsonrpc": "2.0",
  "id": 5,
  "result":
    [{
        "text": "vyhláška",
        "link": "http://localhost/index.php?file=obsah.jpg"
      },
      {
        "text": "oznámení",
        "link": "http://localhost/index.php?file=scan.pdf"
      }]
}
```

Výpis 4.6: Příklad odpovědi odeslané serverem na příkaz `getDocuments` z příkladu 4.3. Odpověď je tvořena polem o délce 1, protože při odesílání bylo použito, k odeslání příkazu, pole o délce 1.

se pouze konvertuje identifikátor přílohy, který celočíselného datového typu, na řetězec, převedená hodnota je vrácena. Získání metadat přílohy zajišťuje implementace metody `GetDetail()`, to probíhá namapování vlastností z vnitřního stavu této struktury na vlastnosti rozhraní `IAttachment`, které je návratovou hodnotou implementované funkce.

Poslední metodou k implementaci je `GetData()`, ve které dochází k získání binárních dat přílohy. Příloha je přístupná z URL uvedené v odpovědi 4.6 pod klíčem `link`. Pro získání přílohy je poslán GET požadavek na uvedenou URL přílohy, a protože tělo HTTP odpovědi splňuje rozhraní io.ReadCloser, je přímo předáno jako návratová hodnota této funkce.

4.4 Uložení dokumentů a příloh

Pro ukládání dokumentů do úložiště je potřeba implementace rozhraní `Store`, které díky rozšíření o chování rozhraní `Fetch`, umožňuje i načítání dokumentů. Jedinou implementací rozhraní Store bude v této je práci úložiště implementované jako souborový systém. Protože implementace rozhraní `Fetch` je, oproti implementaci rozhraní `Fetch` jako napojení na API spisové služby 4.3, rozdílná pouze ve způsobu získávání metadat, kde místo odesílání požadavků na server, jsou metadata načítány ze souborů `.meta.json`, nebude tato část zmíněna. Detailnější popis struktury souboru `.meta.json` bude popsán v průběhu implementace ukládání. Implementace se nachází v balíčku `pkg/store/filesystem`.

Kódování indentifikátorů do `base32`

Kódování identifikátorů dokumentů a příloh, se používá hlavně z toho důvodu možnosti obsahu nepovolených znaků pro výskyt v URL. Při kódování bylo záměrně použito `base32`, protože výsledný kód může obsahovat pouze velká písmena a číslice, zatímco kód vytvořený pomocí `base64` může obsahovat i malé písmena nebo znak /. To by byl problém při provozování aplikace na operačních systémech Windows, kde není v názvech souborů a adresářů rozlišováno mezi velkými a malými písmeny. Proto by mohla vzniknout situace, kdy

rozdílné řetězce budou mít po zakódování stejný název, který se bude lišit pouze velikostí písmen. To by vyvolalo chybu, že daný soubor již existuje.

Struktura Filesystem

Filesystem je strukturou, která implementuje metody z rozhraní `URLResolver` a `Store`.

Při implementaci rozhraní `URLResolver` je potřeba implementovat metodu `Resolve`. Ta z informací v příloze `AttachmentHandle`, složí URL pro lokalizování přílohy v úložišti. URL adresáře přílohy vznikne spojením kořenového adresáře, který byl nastaven uživatelem, s identifikátorem dokumentu převedeného do kódování `base32`. Název přílohy je složen z identifikátoru přílohy, převedeného do kódování `base32` a přípony, která byla detekována, podle prvních 262 bajtů, knihovou `filetype` 4.1.2. Návratová hodnota je tvořena spojením adresáře příloh s název přílohy.

Pro splnění rozhraní `Store` je potřeba implementovat tyto metody:

- `BeginTx()`
- `Commit()`

V implementaci metody `BeginTx()`, která zahájí transakci pro ukládání dokumentů, jsou na začátku načteny, ze souboru `.meta.json` v adresáři zadaném uživatelem, identifikátory aktuálně uložených dokumentů, které jsou následně uloženy do vnitřního stavu struktury. Pokud je aplikace spuštěna poprvé a uživatelem zadaný adresář neexistuje, musí prvně vytvořen. Po provedení těchto kroků je vytvořena nová instance struktury `StoreTx`, která je vrácena.

Druhou metodou k implementaci je `Commit()`, která slouží k závěrečnému potvrzení uložených dokumentů. Tato metoda je volaná až poté, co byly úspěšně dokončeny všechny vytvořené transakce. Na vstupu je očekáváno pole bajtů, vygenerované metodou `Generate()` z rozhraní `Output`, které bude uloženo do úložiště na kořenovou úroveň uživatelem zadaného adresáře.

Struktura StoreTx

Struktura StoreTx je strukturou, která umožňuje vytváření transakcí pro zahájení ukládání nového dokumentu a mazání starých dokumentů dokumentů. Provedené změny pak dokáže potvrdit. K tomu aby tyto funkce byly dostupné, musí být splňováno rozhraní `Tx`, které obsahuje následující metody:

- `BeginDocument()`
- `Commit()`
- `Delete()`

Zahájení transakce dokumentu vyžaduje implementace metody `BeginDocument()`. Při implementaci prvně dochází k vytvoření dočasného adresáře pro uložení dokumentu a jeho příloh. Pro vytvoření dočasného adresáře je využita metoda `ioutil.TempDir()`[7], jenž je součástí standardní knihovny jazyk Go. Název dočasného adresáře je tvořen prefixem `kiwi-ud_`, identifikátorem dokumentu, v kódování `base32`, a náhodným sufixem, který je

[7] Více informací na https://golang.org/pkg/io/ioutil/

přidán použitou metodou. Po dokončení zmíněných kroků je vrácena nová instance struktury `DocumentHandle`.

Aby mohlo, při potvrzování správnosti dokumentu, dojít k atomickému přejmenování (přesunutí) dočasného adresáře dokumentu na skutečný adresář dokumentu, musí být jeho umístění na stejném diskovém oddílu jako cílový adresář. Jinak by mohlo, při zavolání metody `Os.Rename()`, dojít ke kopírování adresáře, které není atomickou operací a může v průběhu provádění selhat.

Dokumenty, které jsou považované za svěšené, musí být odstraněny z úložiště. K tomuto musí existovat implementace metody `Delete()`, která prvně dokumenty, podle zadaného identifikátoru, označí k odstranění. A následně je v metodě `Commit()` odstraní. Tyto změny se promítnou i do vnitřního stavu uložených dokumentů.

Po dokončení vytvořených transakcí dokumentů jsou změny, z vnitřního stavu struktury, potvrzeny v implementaci metody `Commit()`. Ve které probíhá namapování identifikátorů dokumentů na vlastnosti struktury `Meta`, jejíž definice je na výpisu 4.7. Následně je naplněná struktura převedena na formát `JSON`, jenž je atomicky zapsán do souboru `.meta.json`. K atomickému vytváření nebo přepisování souborů je použita knihovna `renameio` 4.1.2.

```
type Meta struct {
  Version string `json:"version"`
  IDs []string `json:"ids"`
}
```

Výpis 4.7: Definice struktury Meta, která je obsahem souboru `.meta.json` na kořenové úrovni. Hlavním výnamem, je uchování aktuálně uložených dokumentů. Tato struktura je načítání v metodě `BeginTx` ve struktuře Filesystem 4.4. Řetězec za datovým type definuje název klíče, pokud bude struktura převedena na formát `JSON`.

Struktura DocumentHandle

Struktura DocumentHandle implementuje rozhraní `DocumentTx`, aby umožňovala vytváření nových transakcí příloh a následné potvrzení úplnosti dokumentu, a také při výskytu neočekávaného stavu umožňuje vrácení provedených změn. K tomu je nutné implementovat následující metody rozhraní:

- `BeginAttachment()`
- `Commit()`
- `Rollback()`

Pro zahájení transakce přílohy, není potřebné provést žádné operace, které by zahájení musely předcházet. Proto implementace metody `BeginAttachment()` pouze vytvoří novou instanci struktury `AttachmentHandle` a vrátí ji.

Pokud uspějí všechny vytvořené transakce příloh `AttachmentHandle`, pak implementaci metody `Commit()` jsou předány metadata dokumentu `IDocument` 3.2.2, která jsou namapovány na vlastnosti struktury `DocumentMeta`, která je definována na výpisu 4.8. Tato struktura je následně převedena do formátu `JSON`, který je atomicky zapsán do souboru `.meta.json` v dočasném adresáři dokumentu. Posledním krokem je atomické přejmenování (přesunutí) dočasného adresáře, metodou `os.Rename()`, která je součástí standardní

knihovny jazyka Go, na skutečné umístění adresáře s dokumentem, jehož názvem je tvořen pouze identifikátor dokumentu převedeného do kódování `base32`.

Pokud dojde k, alespoň jedné, neúspěšně dokončené transakci přílohy, je dokument nekompletní a musí být provedené změny vráceny. K návratu slouží implementace metody `Rollback()`, která dokument ve vnitřním stavu označí jako nedokončený a dočasně vytvořenou složku odstraní.

```
type DocumentMeta struct {
  Version string `json:"version"`
  ID string `json:"id"`
  Name string `json:"name"`
  Category string `json:"category"`
  DateFrom string `json:"dateFrom"`
  DateTo string `json:"dateTo"`
  Attachments []AttachmentMeta `json:"attachments"`
}
```

Výpis 4.8: Definice struktury DocumentMeta, která je obsahem souboru `.meta.json` v adresáři dokumentu. Hlavním výnamem, je uchování metadat jednoho dokumentu, včetně souvisejících příloh 4.9. Tato struktura je načítána při implementaci metody `GetDetail()` z rozhraní `DocumentHandle`.

Struktura AttachmentHandle

Poslední definovanou strukturou je `AttachmentHandle`, která díky implementaci metody `Commit()` z rozhraní `AttachmentTx`, umožňuje ukládání příloh.

Jako vstupní parametry jsou metodě `Commit()` předány metadata přílohy `IAttachment` a stream dat `io.ReadCloser`. Prvním krokem je namapování metadat na vlastnosti struktury `AttachmentMeta`, jejíž definice je na výpisu 4.9. Tato struktura není v této metodě ukládána, ale pouze předána jako vnitřní stav struktuře `DocumentHandle`, která tuto informaci připojí ke zbytku metadat dokumentu `DocumentMeta`. Při dalším kroku je zkontrolováno, zda-li se jedná o online přílohu s validní URL, pak je funkce úspěšně dokončena.

Pokud není příloha označena příznakem online, je potřeba provést uložení. Z implementované metody `Resolve()` 4.4, je zjištěna URL adresa, na kterou jsou data přílohy atomicky zapsány. K atomickému vytváření nebo přepisování souborů je použita knihovna `renameio` 4.1.2.

```
type AttachmentMeta struct {
  ID string `json:"id"`
  Name string `json:"name"`
  Filename string `json:"filename"`
  IsOnline bool `json:"isOnline"`
  OnlineURL string `json:"onlineUrl"`
}
```

Výpis 4.9: Struktura popisuje ukládané metadata jedné přílohy. Ta je vkládána do pole všech příloh dokumentu `DocumentMeta`.

Adresářová struktura v souborovém systému

Potom, co je dokončeno ukládání dokumentů a jejich příloh, vznikne v souborovém systému následující hierarchická adresářová struktura.

```
galileo
├── udxmlv2.xml
├── temp
│   └── kiwi-ud_GYYDENZTGI3Q_456465
│       └── kiwi-ud_GYYDENZTGI3S2MI_12312.pdf
└── data
    ├── .meta.json
    ├── GYYDENZTGI3Q
    │   ├── .meta.json
    │   ├── GYYDENZTGI3S2MI.pdf
    │   └── GYYDENZTGI3S2MJQ.pdf
    └── GYYDGMZQGUZA
        ├── .meta.json
        └── GYYDGMRXGE2S2MI.pdf
```

První úroveň je tvořena uživatelsky zadaným adresářem, jenž je nastavitelný pomocí konfigurační direktivy storageURL 4.6. Ten je určen jako kořenový adresář pro akce vykonané aplikací.

Na druhé úrovni se nachází vygenerované výstupní formáty, dočasná složka temp pro ukládání zatím nepotvrzených dokumentů a datová složka kam jsou po potvrzení tyto dokumenty přesunuty.

Třetí úroveň patří adresářům dokumentů, jejichž název je vytvořen převedením identifikátoru dokumentu do kódování `base32`. Dále jsou na této úrovni v souboru `.meta.json` uchovány informace o aktuálně uložených dokumentech. Informace jsou ve formátu `JSON` vygenerovaného ze struktury `Meta` 4.7.

Na poslední úrovni jsou v souboru `.meta.json` uložena metadata dokumentů a příloh, jež jsou vytvořeny z struktur `DocumentMeta` a `AttachmentMeta`, převedených do formátu `JSON`. Na této úrovni jsou ukládány soubory stažených příloh. Jejich název je tvořen identifikátorem převedeným do kódování `base32` a detekovanou příponou.

4.5 Generování výstupního formátu

Posledním implementovaným rozhraním je `Output`, jehož úkolem je z uložených metadat, vygenerovat výstupní formát, který byl navržen v podkapitole 3.4. Implementace je umístěna v balíčku `/pkg/output/udxmlv2`.

Základní strukturou při pro implementaci rozhraní `Output`, je `Documents`, jejíž definice je na výpisu 4.10. Tato struktura implementuje metodu `Generate()`, která jako vstupní parametry vyžaduje implementace rozhraní `Fetch` a `URLResolver`. Protože výstupní formát má být generován ze souborového systému, bude oběma parametrům předána implementace `Store` 4.4.

Při generování výstupního formátu jsou, metodou `GetDocuments()` z rozhraní `Fetch` získány z úložiště všechny ukazatele na dokumenty `DocumentHandle`. Pro každý z nich je načten metodou `GetDetail()` z rozhraní `DocumentHandle` jejich detail, jehož vlastnosti

```
type Documents struct {
  XMLName xml.Name `xml:"documents"`
  Documents []Document `xml:"document"`
}
```

Výpis 4.10: Definice struktury Documents popisuje kořenový element `<documents>`, který je popsán v návrhu výstupního formátu 3.2. Řetězec za datovým type definuje název XML elmentu.

jsou namapovány na vlastnosti struktury `Document` definované na výpisu 4.11. Vlastnosti `DateFrom` a `DateTo` nelze přímo namapovat, protože jsou odlišného datového typu. Proto musí být převedeny z typu `time.Time` na typ `string`. Aby převedené datum, odpovídalo standardu RFC3339 [16], je použit při převodu následující formátovací řetězec:

```
2006-01-02T15:04:05Z07:00
```

Formátovací řetězec je definován jako konstanta (`time.RFC3339`) ve standardní knihovně. Význam jednotlivých složek je popsán v článku [23].

Při definici struktury 4.11 je ve vlastnosti `DateTo` použito klíčové slovo `omitempty`, které zaručí to, aby daný element nebyl vygenerován, pokud jeho hodnota nebyla nastavena. U vlastnosti `Attachments` je použit symbol >, který vytvoří elementy přesně tak, aby elementy příloh `<attachment>` byly umístěny jako potomci elementu `<attachments>` 3.2.

```
type Document struct {
  ID string `xml:"id"`
  Name string `xml:"name"`
  Category string `xml:"category"`
  DateFrom string `xml:"dateFrom"`
  DateTo string `xml:"dateTo,omitempty"`
  Attachments []Attachment `xml:"attachments>attachment"`
}
```

Výpis 4.11: Struktura Document popisuje kořenový element dokumentu `<document>`, který je popsán v návrhu výstupního formátu 3.2. Řetězec za datovým type definuje název XML elementu.

Posledním krokem je získání ukazatelů na přílohy `AttachmentHandle`, ty jsou získány metodou `GetAttachments()` v rozhraní `DocumentHandle`. Z každého ukazatele je metodou `GetDetail()` načten detail přílohy `IAttachment`, jehož vlastnosti jsou namapovány na vlastnosti struktury `Attachment` definované výpisu 4.12. Protože v získaných metadatech není k dispozici URL adresa přílohy, která je vyžadována strukturou `Attachment` 4.12, musí být získána z metody `Resolve()`, která je součástí předané implementace rozhraní `URLResolver`.

4.6 Uživatelská konfigurace

Aplikace je konfigurovatelná pomocí proměnného prostředí a parametrů příkazové řádky. Pokud jsou použity při spuštění oba způsoby, pak jsou upřednostňovány parametry příkazové řádky. Tento seznam definuje povolené parametry příkazové řádky, v závorkách je

```
type Attachment struct {
  XMLName xml.Name `xml:"attachment"`
  ID string `xml:"id"`
  Name string `xml:"name"`
  URL string `xml:"url"`
  IsOnline bool `xml:"isOnline"`
}
```

Výpis 4.12: Struktura Attachment popisuje kořenový element přílohy `<attachment>`, který je popsán v návrhu výstupního formátu 3.2. Řetězec za datovým type definuje název XML elmentu.

odpovídající ekvivalent pro proměnnou prostředí. Parametry příkazové řádky musejí být zadávány s prefixem `--`.

- **service (SERVICE)** - specifikuje typ použité spisové služby nebo jiného zdroje dat. Mezi povolené hodnoty patří `galileo, geovap, ginis, icz, udxmlv1 a udxmlv2`, jiné než tyto hodnoty vyvolají chybu, stejně jako, když parametr není uveden.
- **serviceUrl (SERVICE_URL)** - nastavuje URL adresu serveru spisové služby nebo jiného zdroje dat. Adresa musí odpovídat formátu podle dokumentu RFC 3986 [2]:

  ```
  scheme://user:password@server:port/path?param=special#anchor
  ```

  Pokud je vyžadována autentizace pro přístup ke zdroji dat, mohou být tyto údaje zadány mezi protokol a doménové jméno. Uživatelské jméno a heslo jsou mezi sebou odděleny symbolem :, od doménového jména je odděluje symbol @. Autentizační údaje nejsou povinné. Specifické údaje pro konkrétní zdroj dat mohou být zadány jako parametry URL, které jsou za symbolem ?. Jednotlivé parametry jsou mezi sebou odděleny symbolem &.
- **storage (STORAGE)** - specifikuje typ použitého úložiště. Mezi povolené hodnoty patří `file`.
- **storage (STORAGE_URL)** - nastavuje URL adresu cílového úložiště, kde budou ukládány data. Formát URL adresy musí splňovat stejné náležitosti jako je v případě parametru `serviceUrl`.

  Toto nastavení není povinné. Jeho výchozí hodnotou cesta relativní k cestě spouštěné aplikace.
- **outFormat (OUT_FORMATS)** - definuje, které výstupní formáty budou vygenerovány. Mezi povolené hodnoty patří `udxmlv1` a `udxmlv2`. Je podporování zadání více výstupních formátů současně.

  Pro zadání více hodnot současně přes parametry příkazové řádky, musí být tento parametr pro každý formát uveden zopakován. Při nastavení přes proměnné prostředí jsou všechny formáty uvedeny jako seznam oddělených hodnot, jejíž oddělovačem je symbolem ;.

  Tento parametr není povinný, jeho výchozí hodnota je `udxmlv2`.

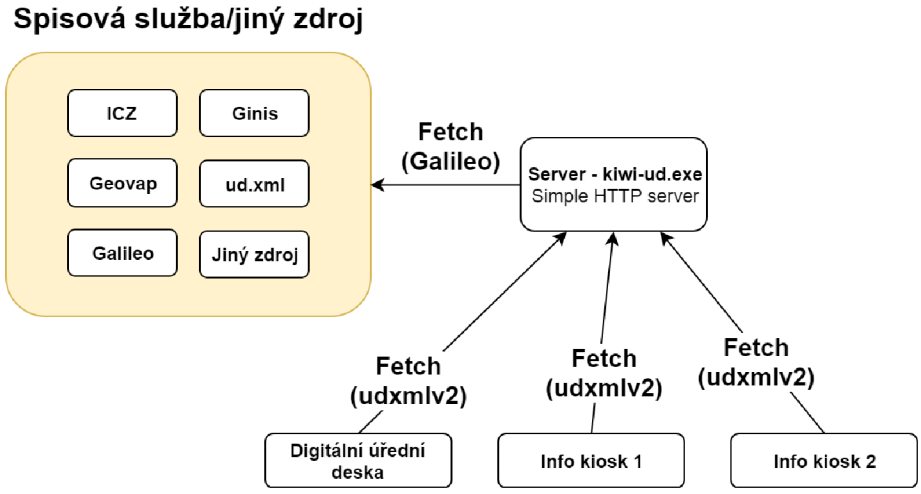
- **productionName (PRODUCTION_NAME)** - nastavuje název běžící instance aplikace, pod kterým se bude hlásit např. při odeslané chybě na Sentry 5.2.2.
- **SENTRY_DSN** - nastavuje URL adresu projektu na službě Sentry 5.2.2. Na zadanou adresu budou hlášeny vzniklé události. Tato hodnota je nastavitelná pouze přes proměnného prostředí.
- **debug** - zapne podrobné výpisy při provádění. Pro toto nastavení nelze použít proměnnou prostředí.
- **version** - slouží pro vypsání detailů o verzi aplikace. Tyto informace jsou nastavovány při sestavování aplikace v CI/CD 5.1.1. Tuto akci lze vyvolat pouze přes parametr příkazové řádky. Po vypsání verze je aplikace ukončena.

4.7 Příklad vzájemně spolupracujících aplikací

V případech, kdy tato aplikace bude u zákazníka provozována na více zařízeních současně, je dobré mít možnost použití centrálního serveru. Dokumenty budou stahovány pouze na jenom místě, ze kterého budou v lokální síti servírovány pro ostatní zařízení.

Díky navrženému výstupnímu formátu 3.4, který je zároveň implementován jako zdroj dat, lze realizovat výše popsaný způsob komunikace znázorněné na obrázku 4.1. Na centrálním serveru je v pravidelných intervalech spouštěna aplikace, která komunikuje se spisovou službou a tím stahuje dokumenty. Zároveň na stejném serveru běží jednoduchý `HTTP` server, který servíruje aplikací používaný adresář 4.4 s dokumenty a souborem výstupního formátu 3.4.

Např., pokud HTTP server běží na adrese `http://10.10.10.10`, pak zařízení Digitální úřední deska, Info kiosk 1 a 2 musí být spouštěny s parametry `--service` udxmlv2 a `--serviceUrl` http://10.10.10.10/UDxmlV2.xml, aby bylo dosaženo požadovaného stavu.



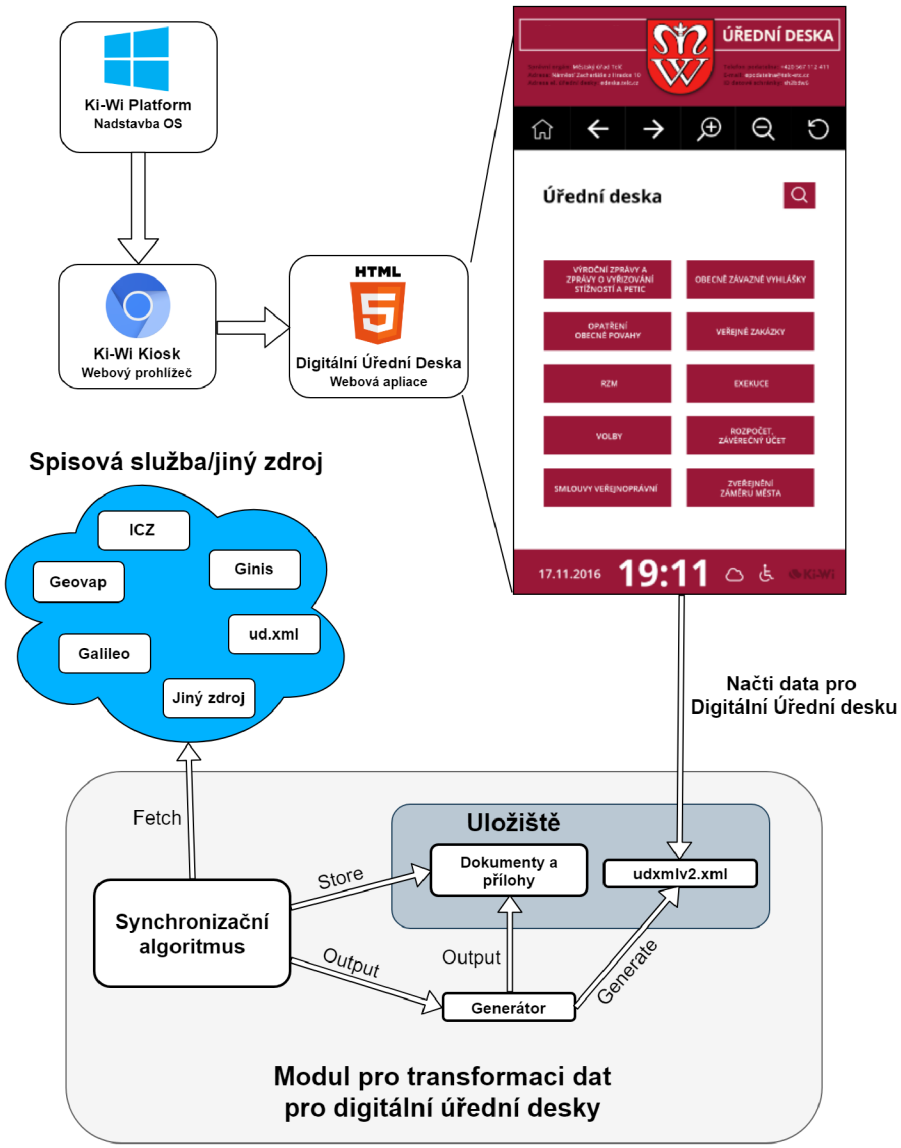
Obrázek 4.1: Optimalizace stahování dat ze spisové služby s použitím centrálního serveru a jednou aplikací komunikující. Z tohoto serveru budou data servírované ostatním zařízením v lokální síti.

4.8 Integrace s dalšími produkty Ki-Wi Digital s.r.o.

Na obrázku 4.2 je graficky znázorněna integrace výsledků této práce v celém produktu digitální úřední desky.

Ki-Wi Platform společně s Ki-Wi Kiosk tvoří základ pro automatické spouštění a zobrazování webových stránek na koncovém zařízení.

HTML aplikace Digitální Úřední Desky tvořena třemi hlavními stránkami. První z nich je zobrazení všech kategorií. Druhá stránka je tvořena detailem kategorie, jež obsahem jsou pouze dokumenty spadající do zvolené kategorie. Poslední obrazovka slouží jako detail dokumentu, odkud je možné otevírat lokálně uložené přílohy. Informace o tom, které kategorie a dokumenty v nich budou zobrazovány jsou dané z načteného výstupního formátu 3.4.



Obrázek 4.2: Schéma popisující integraci výsledků této práce do produktu digitální úřední desky. Logo Windows, Chromium a HTML5 byly převzaty z [38], [39], [37]

Kapitola 5

Nasazení

Předposlední kapitola uvede použití nástrojů pro průběžnou integraci a průběžné nasazení (CI/CD). Závěr kapitoly bude věnován realizaci logování a hlášení chyb.

5.1 CI/CD

CI/CD je zkratkou z anglického Continuous Integration/Continuous Delivery, což je překládáno jako průběžná integrace/průběžné nasazení. Jedná se o proces, při kterém dochází k automatizaci běžných procesů známých při vývoji softwaru, kterými jsou testování, sestavení, zabalení a nahrání na aktualizační server. Mezi oblíbené a používané CI/CD nástroje patří např. GitLab[1], TeamCity[2], Jenkins[3] nebo Bamboo[4] [25].

Pro tuto práci byl použit nástroj GitLab, který se ve firmě již používá při jiných projektech. Kromě použití GitLab jako pro verzování zdrojových kódů, nabízí v rámci svého ekosystému nástroje pro pokrytí celého životního cyklu DevOps[5] od plánování až po monitorování aplikace.

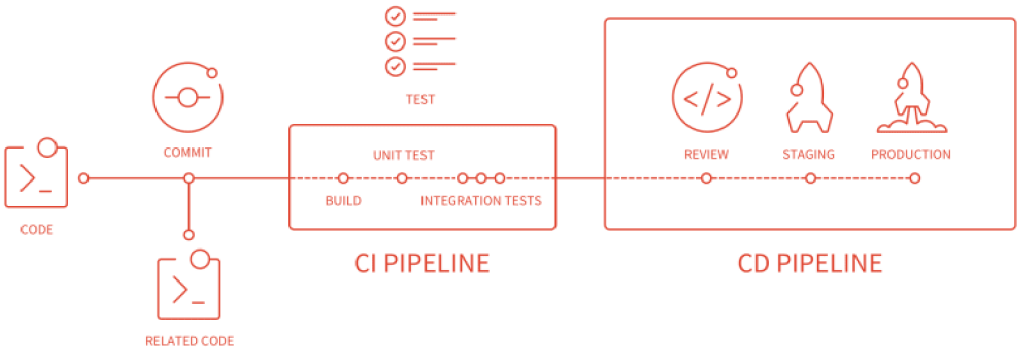
Na obrázku 5.1 je graficky znázorněn proces průběžné integrace a nasazení. Celý proces může sloužit jako manuál, ve kterém je popsáno, jak ručně spustit testy, sestavit a nahrát aplikaci na aktualizační server. Toto chování je popsáno souborem `.gitlab-ci.yml`[6], kde je každá fáze tvořena sérii příkazů popisující očekávané chování.

5.1.1 Continuous Integration

Pojmem Continuous Integration (CI), nebo-li průběžná integrace se rozumí systémová integrace (git commit) nové, oprava staré funkcionality v aplikaci. K integraci může docházet od různých vývojářů i několikrát za den. Proto je důležité, aby každá provedená změna žádným negativním způsobem neovlivnila chování aplikace. Tato fáze zahrnuje kroky:

- sestavení aplikace,
- spuštění jednotkových (unit) a integračních testů,

[1] Více informací o GitLab na https://about.gitlab.com/
[2] Více informací o TeamCity na https://www.jetbrains.com/teamcity/
[3] Více informací o Jenkins na https://jenkins.io/
[4] Více informací o Bamboo na https://www.atlassian.com/software/bamboo
[5] Fáze DevOps na GitLab https://about.gitlab.com/stages-devops-lifecycle/
[6] Vytvoření `.gitlab-ci.yml` je popsáno na https://docs.gitlab.com/ee/ci/quick_start/



Obrázek 5.1: Schéma popisuje proces průběžné integrace a průběžného nasazení. Obrázek byl převzat z [8].

- zabalení sestavené aplikace a vytvoření artefaktů[7] (artifacts). [36]

Výše popsané kroky jsou graficky znázorněny jako tzv. `CI PIPELINE`, na obrázku 5.1. Tato fáze začíná, když jsou provedené změny v kódu uloženy (git commit) a nahrány (git push) na server.

Testování

První fází je testování, pro kterou byly vytvořeny automatizované testy. V každém z balíčků je obsažen testový soubor, jehož název je tvořen názvem balíčku a sufixem `_test.go`, díky kterému Golang pozná, že se jedná o soubor s testy.

Implementace napojení na spisovou službu nebo jiný zdroj dat bylo testováno tak, že pro každý implementovaný zdroj byl vytvořen mock server, který simuluje chování reálného serveru. Na základě předem známých HTTP požadavků, zasílaných klientskými zařízeními, jsou odesílány odpovídající předem definované HTTP odpovědi. K vytvoření mock serveru byl použit balíček `httptest`[8] ze standardní knihovny Go. Otestování všech balíčků proběhne spuštěním příkazu:

```
go test ./... --cover
```

Celkové pokrytí aplikace testy je 68 %, protože né všechny části aplikace je nutné mít pokryty ze 100 % testy. Vytvoření některých testů může zabrat velké množství času a přitom celkový přínos nebude zdaleka odpovídat věnovanému času. Například vytvoření komplexního mock serveru pro každou implementovanou spisovou službu je velmi časově náročné.

Sestavení

Po úspěšně dokončené fázi testování, je aplikace sestavena pro platformy Windows a Linux. K tomuto se používá knihovna `gox` 4.1.2. Sestavené soubory jsou uloženy do adresáře,

[7] Seznam souborů, adresářů, které jsou spojeny s úspěšně dokončenou fází. Např. fáze sestavování může mít jako artefakty přeložené soubory. [9]

[8] Jedná se o balíček poskytující nástroje pro testování HTTP [10].

který je odvozen z názvu překládaného Go balíčku, v tomto případě je to `kiwi-ud`, a tento adresář je nastaven jako artefakt. Název sestavené aplikace pro konkrétní platformy je tvořen názvem překládaného balíčku, operačním systémem a architekturou. Tyto části jsou mezi sebou odděleny symbolem _.

Pro jednoznačnou identifikaci aplikace, po tom co byla nasazena, jsou při sestavování nastaveny vnitřní proměnné `Version` a `CommitSHA`, které mohou být vypsány při spuštění aplikace s parametrem `--version`. K zapsání těchto informací slouží parametr -ldflags, v jehož hodnotě musí být definován přepínač -X, který odkazuje na konkrétní balíček a název proměnné, která má být nastavena. K sestavení aplikace pro více platforem paralelně, včetně nastavení identifikace, je spouštěn příkaz z příkladu 5.1.

```
gox \
-ldflags="-X main.Version=$CI_COMMIT_TAG -X main.CommitSHA=$CI_COMMIT_SHA"\
-osarch="linux/amd64 windows/386" \
-output="$PACKAGE/{{.Dir}}_{{.OS}}_{{.Arch}}" ./cmd/$PACKAGE
```

Výpis 5.1: Ukazuje použití nástroje `gox` pro překlad aplikace na více platforem a současně nastavuje jednoznačnou identifikaci pro detekci nasazení.

Proměnná `Version` obsahuje proměnnou prostředí $CI_COMMIT_TAG, jejíž hodnota je nastavena při zahájení běhu CI. Jejím obsahem je informace uložená ve značce (git tag). Pokud je ručně spouštěna nebo testována verze, pro kterou nebyla vytvořena značka (git tag), pak k identifikaci slouží druhá proměnná, kterou je `CommitSHA`. Ta je nastavena vždy. Jejím obsahem je hodnota z proměnná prostředí $CI_COMMIT_SHA, ta má za úkol jednoznačně identifikovat revizi (commit), ze kterého byla aplikace sestavena.

5.1.2 Continuous Delivery

Hlavním úkolem ve fázi Continuous Delivery (CD), nebo-li průběžného nasazení, je připravení a nahrání sestavené aplikace na aktualizační server. Jako server, kde budou uchovávány vydané verze aplikace, jsem se rozhodl využít úložiště Amazon S3 [9]. Toto řešení bylo zvoleno na základě nabízených funkcí, jednoduchého API a snadné integraci v prostředí Docker[10].

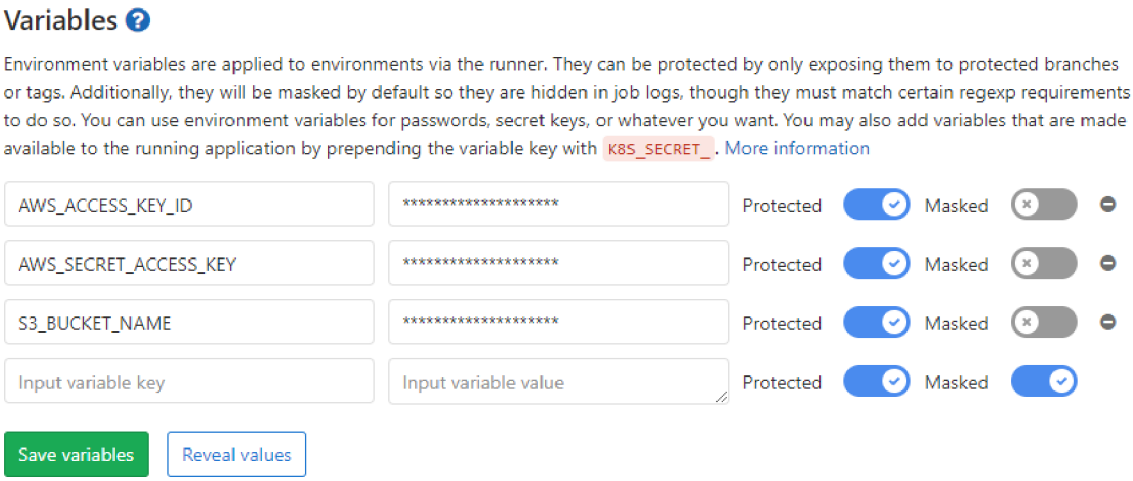
Tato fáze není spouštěna automaticky při každém nahrání nového revize (commit), ale jen u těch revizí, které byly schváleny a opatřeny značkou (git tag).

Autentizační údaje a adresa k AWS S3 úložišti jsou nastavovány přes proměnné prostředí v GitLabu, aby bylo zajištěno jejich bezpečné předání a také jejich snadnější změna. Proměnné prostředí se nastavuje nastavuje v nastavení projektu (Settings → CI/CD v části Variables 5.2).

Následné použití takto vytvořených proměnných je stejné, jako použití jakékoliv jiné proměnné v unixovém prostředí, tj. obsah proměnné je přístupný tak, že před nadefinovaný název je uveden symbol $, např. adresa S3 serveru je přístupná přes proměnnou $S3_BUCKET_NAME.

Při spuštění CD jsou automaticky staženy artefakty z fáze sestavení 5.1.1. Jako první je stažen, z úložiště Amazon S3, aktualizační manifest `update_manifest.xml`, který uchovává XML strukturu obsahující seznam vydaných verzí. Tato struktura je popsána pomocí XSD 5.3.

[9] Simple Storage Service https://aws.amazon.com/s3/
[10] Více informací o nástroji Docker na https://www.docker.com/



Obrázek 5.2: Nastavení proměnných prostředí používaných CI/CD skriptu. Nastavení je přístupné přes Settings → CI/CD v části Variables

```
<kiwi-ud pubDate="2019-03-18T18:51:25+0000">
  <version>0.2.7</version>
  <mandatory>false</mandatory>
  <commitSHA>285b660992abcc4d4b18d1f4bcf6746c60bf5559</commitSHA>
  <binary os="windows" arch="x86">
    <path>0.2.7/kiwi-ud_windows_386.exe</path>
    <checksum>bb4a0f141c3dfeed30a7b01aaf3d0526</checksum>
  </binary>
  <binary os="linux" arch="x64">
    <path>0.2.7/kiwi-ud_linux_amd64</path>
    <checksum>e488d9e92e1d516efcb0ea2ccd21b2b0</checksum>
  </binary>
</kiwi-ud>
```

Výpis 5.2: Příklad záznamu vloženého do aktualizačního manifestu (update_manifest.xml), který popisuje jednu konkrétní vydanou verzi aplikace.

Příklad XML struktury popisující vydanou verzi je na příkladu 5.2. Kořenovým elementem je `<kiwi-ud>`, jehož potomci uchovávají informace o verzi aplikace, identifikaci nahrané revize (commit SHA) a seznam sestavených platforem.

Každá platforma je reprezentována elementem `<binary>`, v jehož atributech jsou uloženy informace o operačním systému a použité architektuře. Dále je v elementu `<path>` obsažena cesta k sestavené aplikaci v úložišti, která je relativní k aktualizačnímu manifestu. V elementu `<checksum>` je kontrolní součet sestavené aplikace, který slouží pro ověření zdali kontrolní součet staženého souboru je stejný jako uvedený kontrolní součet. Pro vytvoření kontrolního součtu je použit algoritmus MD5.

Záznam s nově přidávanou verzí je prvně aktualizován pouze lokálně ve staženém aktualizačním manifestu. Poté, co úspěšně proběhne nahrání sestavených aplikací, tak se provede nahrání nové revize aktualizačního manifestu (`update_manifest.xml`). Toto pořadí kroků zaručí konzistenci mezi informacemi uloženými v aktualizačním manifestu a nahranými aplikacemi. Nebude tak moci dojít k situaci, kdy bude existovat nový aktualizační mani-

fest odkazující na novou neexistující verzi aplikace, protože např. její nahrávání v průběhu selhalo.

```
<xs:schema attributeFormDefault="unqualified"
  elementFormDefault="qualified"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://ki-wi.cz/kiwi-ud/update/1.0">
<xs:element name="updates">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="kiwi-ud" maxOccurs="unbounded" minOccurs="0">
        <xs:complexType>
          <xs:sequence>
            <xs:element type="xs:string" name="version"/>
            <xs:element type="xs:string" name="commitSHA"/>
            <xs:element name="binary" maxOccurs="unbounded" minOccurs="1">
              <xs:complexType>
                <xs:sequence>
                  <xs:element type="xs:string" name="path"/>
                  <xs:element type="xs:string" name="checksum"/>
                </xs:sequence>
                <xs:attribute type="xs:string" name="os" use="required"/>
                <xs:attribute type="xs:string" name="arch" use="required"/>
              </xs:complexType>
            </xs:element>
          </xs:sequence>
          <xs:attribute type="xs:string" name="pubDate" use="required"/>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
</xs:schema>
```

Výpis 5.3: XSD popisující strukturu pro ukládání vystavených verzí v aktualizačním manifestu (update_manifest.xml).

5.2 Logování

Logávání je nedílnou součástí každé aplikace, obzvlášť v produkčním prostředí. Užitečné je v případech, kdy se vyskytla chyba, nebo došlo k neočekávanému chování aplikace. Umožní jednodušší zjištění kroků, které předcházely neočekávanému stavu. Pokud se chyba vyskytne, je důležité umět na ni co nejrychleji reagovat analýzou kódu. K tomu běžně slouží různé webové služby, které umožňují odesílání chyb.

5.2.1 Strukturované logy

K logování se používají tzv. strukturované logy, které jsou jednoduše strojově zpracovatelné a díky tomu jejich jejich vizuální styl může být libovolně upraven. K logování je použí-

vána knihovna `zerolog`[11]. To umožňuje vytváření zmíněné strukturované logy, které jsou vytvářeny ve formátu `JSON`. V aplikaci se rozlišují dva typy logů.

- **Programové logy**, které monitorují chování programu. Jsou určeny především pro vývojáře. Ty se dělí na dvě základní úrovně:
  - **Informační logy** monitorují pouze základní informace o běhu programu, jakými jsou verze aplikace, počet stahovaných dokumentů, počet odstraňovaných dokumentů. V této úrovni jsou vypisovány i vzniklé chyby.
  - **Podrobné logy** v sobě zahrnují výše zmíněné informační logy a jsou rozšířeny o podrobnější informace o chování programu, kdy jsou monitorovány vnitřní stavy důležitých objektů, informace o tom, kdy začalo a kdy skončilo stahování dokumentu nebo přílohy. Díky tomu jsou dostupné detailní informace o tom, jaký byl vnitřní stav aplikace před neočekávaným chováním, aniž by bylo nutné aplikaci znovu spouštět. Tato úroveň se zapíná přepínačem `--debug`.
- **Audit logy** jsou určeny především pro zákazníky, kteří takovéto informace vyžadují. V takovém případě jsou zachytávány události, jako např. vyvěšení/svěšení dokumentů včetně názvů a data, kdy k události došlo. V tabulce 5.1 je ukázán příklad, jak mohou logy pro zákazníka vypadat.

Díky použití strukturovaných logů, lze jednoduchým způsobem tyto kategorie rozlišit. K tomu lze využít řada nástrojů, které podporují filtrování z formátu `JSON`. Programové logy jsou označovány hodnotou `program` v klíči `event`, audit logy mají v klíči `event` hodnotu `audit`. Ukázka strukturovaných logů je na výpisu 5.4.


```
[{"level":"info","event":"program","time":1555937945,"message":"4 documents
    to download"},
{"level":"debug","event":"program","time":1555937945,"message":"Downloading
    new documents"},
{"level":"debug","event":"program","tempDir":"C:/data/kiwi-
   ud_GYYDENZTGI3Q_359630631","time":1555937945,"message":"documentTempDir
   "},
{"level":"debug","event":"program","time":1555937945,"message":"Successful
   GET attachment with ID 60-1"},
{"level":"info","event":"debug","ID":"60-1","Název":"138_2018.pdf","time":
   1555937945,"message":"Příloha dokumentu byla stažena!"},
{"level":"debug","event":"program","time":1555937945,"message":"Successful
   GET attachment with ID 60-2"},
{"level":"info","event":"audit","ID":"60","Název":"Výzva podle zákona","
   time":1555937946,"message":"Novy dokument byl stažen!", "attachments":"
   [{"id": "60-1", "name": "138_2018.pdf"}]"}]
```


Výpis 5.4: Ukázka automaticky vygenerovaných strukturovaných logů ve formátu JSON. Tyto logy byly vygenerovány knihovnou zerolog 4.1.2. Aplikace byla spuštěna s parametrem `--debug` pro zahrnutí podrobných výpisů.

[11] zerolog poskytuje rychlý a jednoduchý způsob vytváření logů ve formátu JSON [26]

Datum	ID	Název dokumentu	Stažené přílohy		Událost
22. 2. 2019 v 15:21	60	Výzva podle zákona	ID	Název	Stažení
			60-1	138_2018.pdf	

Tabulka 5.1: Zobrazuje možnost, jak mohou vypadat převedené vyfiltrované audit logy na tabulku. Tabulka je vygenerovaná z vyfiltrovaných strukturovaných logů 5.2.1

.

5.2.2 Hlášení chyb

Pro hlášení chyb existuje řada webových služeb, které nabízejí i jiné funkce. Mezi oblíbené nástroje patří `Sentry`, `LogRocket`, `Rollbar` nebo `Raygun` [1].

V této práci jsem se rozhodl použít službu `Sentry` z následujících důvodů:

- nabízí oficiální SDK pro jazyk Go,
- v základní verzi, která obsahuje až 5000 nahlášených chyb za měsíc, je používání zcela zdarma,
- jedná se open-source projekt,
- webové rozhraní je jednoduché a přehledné,
- řada aplikací ve firmě již využívá této služby.

Mezi další užitečné funkce patří integrace s komunikátorem Slack, kdy při výskytu chyby se ihned v určeném kanálu objeví podrobnosti. Sentry je využíváno i velkými firmami jakými jsou Dropbox, Autodesk, Microsoft, DigitalOcean, Atlassian a mnoho dalších [30].

Integrace služby

Integrace Sentry do Golang aplikace je velmi jednoduchá. Aby byly chyby odesílány, musí být nastavena proměnná prostředí `SENTRY_DSN` 4.6. Odesílání je otázkou pár řádku, jako jsou na příkladu 5.5, které jsou umístěny na místě, kde probublají chyby, které mají být odesílány.

```
trace := raven.NewStacktrace(0, 2, nil)
packet := raven.NewPacketWithExtra(err.Error(), raven.Extra{
  "runtime.OS": runtime.GOOS,
  "runtime.ARCH": runtime.GOARCH,
  "ProductionName": ProductionName,
  "Version": Version,
  "CommitSHA": CommitSHA,
  }, raven.NewException(err, trace))
_, ch := raven.Capture(packet, nil)
```

Výpis 5.5: Ukázkový kód pro integraci služby Sentry do Golang aplikace. Pokud je nastavena proměnná prostředí `SENTRY_DSN`, pak není potřeba adresu pro odesílání zadávat. V ostatních případech je potřeba adresu zadat programově, jak je ukázáno na příkladu [29].

Odesílané informace

Kromě popisu chyby a tzv. stack trace[12], jsou odesílány i další informace:

- Název a architektura operačního systému kde byla aplikace spuštěna.
- Název spuštěné instance aplikace které je nastavena při spouštění aplikace parametrem `--productionName` 4.6. Tímto lze jednoznačně identifikovat konkrétní zařízení.
- Verze spuštěné aplikace, která byla použita z vytvořené značky (git tag) dané revize (git commit).
- Poslední informací je kontrolní součet dané revize, ze které aplikace byla sestavena (commit SHA).

[12] Více informace o stack strace na https://en.wikipedia.org/wiki/Stack_trace

Kapitola 6

Závěr

Cílem této práce bylo vytvořit aplikaci pro stahování dokumentů a jejich přílohy z modulu úřední desky, které bude transformovat do navrženého strukturovaného formátu.

Prvním úkolem bylo prostudovat a porovnat rozhraní (API) poskytovatelů elektronických systémů spisové služby. Ze získaných znalostí jsem navrhl jednotné rozhraní pro stahování, ukládání dat a generování výstupního formátu. Následně jsem pro získaná data navrhl jednotný strukturovaný formát, k jehož vytvoření jsem použil značkovací jazyk XML. Správnost řešení jsem v průběhu vývoje ověřoval vytvořenými automatickými testy. Při vývoji jsem používal nástroje pro průběžnou integraci a průběžné nasazení (CI/CD). Během tohoto procesu byla aplikace otestována, sestavena pro platformy Windows a Linux, zabalena a následně vytvořený balíček publikován ke stažení na cloudové úložiště Amazon S3. Jako implementační technologie pro vývoj aplikace jsem použil programovací jazyk Golang a službu GitLab pro integraci Continuous Integration/Continuous Delivery.

Vytvořená aplikace bude použita jako jedna z části při realizaci řešení pro digitální úřední desky. Při vývoje aplikace bylo dbáno na kvalitu a udržitelnost řešení, tak bylo možné bezproblémové nasazení v produkčním prostředí na desítkách zařízení po celé České republice. Během pilotního nasazení budou odstraňovány případně vzniklé chyby, které nebyly odhaleny během vývoje. Po této fázi bude probíhat průběžné nahrazování stávajícího řešení. Aplikace se v celém řešení bude starat o stahování a následnou transformaci dokumentů do navrženého výstupního formátu. Ten bude načítán a vizualizován webovou aplikací pro Digitální Úřední Desky.

Při pokračování v práci se jako první z možností nabízí integrace nových poskytovatelů spisových služeb. Druhou možností je rozšíření funkcionality o podporu příloh typu archív (ZIP...), které by byly rozbalovány a jejich struktura přímo promítnuta do výstupního formátu. To by usnadnilo činnost dalších aplikací, které by již s archívem nepracovaly. Tento typ přílohy však není příliš obvyklý. Dlouhodobějším cílem je snaha o tzv. dockerizaci, která celkově usnadní nasazení na koncových zařízeních. S tím je spojeno navržení a implementace nového úložiště, protože Docker ve výchozím stavu neposkytuje perzistentní úložiště, tzn. po vypnutí kontejneru jsou zahozeny všechny uvnitř provedené změny.

Díky této práci jsem měl možnosti vyzkoušet a naučit se moderní technologie používané při vývoji softwaru. První z nich je jazyk Go, který je jedním z nejvhodnějších jazyků pro implementaci tohoto druhu aplikací, protože kombinuje výhody kompilovaných a dynamicky typovaných jazyků. Další technologií je nástroj GitLab v kombinaci s cloudovým úložištěm Amazon S3 pro realizace procesů Continuous Integration/Continuous Delivery.

Literatura

[1] Arsenault, C.: Error Tracking - Top Suggestions and Tools. Září 2018, [Online; navštíveno 16.4.2019].
URL https://www.keycdn.com/blog/error-tracking

[2] Berners-Lee, T.; Fielding, R. T.; Masinter, L.: Uniform Resource Identifier (URI): Generic Syntax. STD 66, RFC Editor, January 2005, [Online; navštíveno 7.4.2019].
URL http://www.rfc-editor.org/rfc/rfc3986.txt

[3] Caraveo, R.; rcaraveo; Jones, T.; aj.: golang-set. GitHub, [Online; navštíveno 13.1.2019].
URL https://github.com/deckarep/golang-set

[4] Cheney, D.; Sweeney, T.; Mudrik, S.; aj.: errors. GitHub, [Online; navštíveno 22.1.2019].
URL https://github.com/h2non/filetype

[5] Cramer, D.; Rudenberg, J.; Robenolt, M.: raven. GitHub, [Online; navštíveno 5.2.2019].
URL https://github.com/getsentry/raven-go

[6] Fielding, R. T.; Gettys, J.; Mogul, J. C.; aj.: Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, RFC Editor, Červen 1999, [Online; navštíveno 2.11.2018].
URL http://www.rfc-editor.org/rfc/rfc2616.txt

[7] Geovap: Geovap - Spisová služba. Únor 2016, [Online; navštíveno 1.11.2018].
URL https://dms.geovap.cz/

[8] GitLab: GitLab Continuous Integration & Delivery. online, [Online; navštíveno 15.2.2019].
URL https://about.gitlab.com/product/continuous-integration/

[9] GitLab: Introduction to job artifacts. GitLab Documentation, [Online; navštíveno 15.2.2019].
URL https://docs.gitlab.com/ee/user/project/pipelines/job_artifacts.html

[10] Golang: httptest. online, [Online; navštíveno 20.01.2019].
URL https://golang.org/pkg/net/http/httptest/

[11] Golang: Companies currently using Go throughout the world. Březen 2019, [Online; navštíveno 8.4.2019].
URL https://github.com/golang/go/wiki/GoUsers

[12] Group, J.-R. W.: JSON-RPC 2.0 Specification. Leden 2013, [Online; navštíveno 4.11.2018].
URL https://www.jsonrpc.org/specification

[13] Hashimoto, M.: Gox - Simple Go Cross Compilation. GitHub, [Online; navštíveno 19.2.2019].
URL https://github.com/mitchellh/gox

[14] Holubová, I.; Pokorný, J.: *XML technologie: principy a aplikace v praxi*. Grada, 2008, ISBN 978-80-247-2725-7.

[15] Kieboom, J.: Gox - Simple Go Cross Compilation. GitHub, [Online; navštíveno 11.1.2019].
URL https://github.com/jessevdk/go-flags

[16] Klyne, G.; Newman, C.: Date and Time on the Internet: Timestamps. RFC 3339, RFC Editor, July 2002.

[17] Knesl, J.: Návod na User Stories. Listopad 2014, [Online; navštíveno 12.12.2018].
URL http://www.knesl.com/navod-na-user-stories

[18] Martin, R. C.: *Clean code a handbook of agile software craftmanship*. Prentice Hall, 2010, ISBN 978-0-13-235088-4.

[19] Miroliubov, D.: Doc free icon. [Online; navštíveno 10.4.2019].
URL https://www.flaticon.com/free-icon/doc_337932

[20] Miroliubov, D.: Pdf free icon. [Online; navštíveno 10.4.2019].
URL https://www.flaticon.com/free-icon/pdf_337946

[21] Miroliubov, D.: Png free icon. [Online; navštíveno 10.4.2019].
URL https://www.flaticon.com/free-icon/png_337948

[22] Miroliubov, D.: Xls free icon. [Online; navštíveno 10.4.2019].
URL https://www.flaticon.com/free-icon/xls_337958

[23] Nilsson, S.: Format and parse a time or date [complete guide]. online, [Online; navštíveno 9.2.2019].
URL https://yourbasic.org/golang/format-parse-string-time-date-example/#all-layout-options

[24] O'Tuathail, E.; Rose, M.: Using the Simple Object Access Protocol (SOAP) in Blocks Extensible Exchange Protocol (BEEP). RFC 4227, RFC Editor, Leden 2006, [Online; navštíveno 3.11.2018].
URL https://tools.ietf.org/html/rfc4227

[25] Pecanac, V.: Top 8 Continuous Integration Tools. online, 2 2016, [Online; navštíveno 15.2.2019].
URL https://code-maze.com/top-8-continuous-integration-tools/

[26] Poitrey, O.: Zero Allocation JSON Logger. GitHub, [Online; navštíveno 5.2.2019].
URL https://github.com/rs/zerolog

[27] Quest, K.: Standard Go Project Layout. GitHub, [Online; navštíveno 7.1.2019].
URL https://github.com/golang-standards/project-layout

[28] Ramanathan, N.: Part 21: Goroutines. Červenec 2017, [Online; navštíveno 8.4.2019].
URL https://golangbot.com/goroutines/

[29] Sentry: package raven. GoDoc online, [Online; navštíveno 15.04.2019].
URL https://godoc.org/github.com/getsentry/raven-go#example-package

[30] Sentry: Powering the world's best apps. online, [Online; navštíveno 16.4.2019].
URL https://sentry.io/customers/

[31] Stapelberg, M.: renameio. GitHub, [Online; navštíveno 7.3.2019].
URL https://github.com/google/renameio

[32] Tišnovský, P.: Go: minimalistický a překvapivě výkonný programovací jazyk. Listopad 2018, [Online; navštíveno 9.1.2019].
URL https://www.root.cz/clanky/go-minimalisticky-a-prekvapive-vykonny-programovaci-jazyk/

[33] Tomás: renameio. GitHub, [Online; navštíveno 22.1.2019].
URL https://github.com/h2non/filetype

[34] Wikipedia: Elektronický systém spisové služby. Srpen 2016, [Online; navštíveno 16.10.2018].
URL https://cs.wikipedia.org/wiki/Elektronick%C3%BD_syst%C3%A9m_spisov%C3%A9_slu%C5%BEby

[35] Wikipedia: User story. Listopad 2018, [Online; navštíveno 12.12.2018].
URL
http://en.wikipedia.org/w/index.php?title=User%20story&oldid=891480381

[36] Wikipedia: Průběžná integrace. 2019, [Online; navštíveno 14.2.2019].
URL https://cs.wikipedia.org/wiki/Průběžná_integrace

[37] Wikipedie: Logo HTML5 od W3C. 1 2011, [Online; navštíveno 12.4.2019].
URL https://cs.wikipedia.org/wiki/HTML5#/media/File:HTML5_logo_and_wordmark.svg

[38] Wikipedie: This is images the Windows logo used in Windows Server 2012 and Windows 8. 8 2012, [Online; navštíveno 12.4.2019].
URL https://cs.wikipedia.org/wiki/Microsoft_Windows#/media/File:Windows_logo_-_2012.svg

[39] Wikipedie: Main logo and icon for the open source internet browser project, Chromium. 7 2015, [Online; navštíveno 12.4.2019].
URL https://en.wikipedia.org/wiki/Chromium_(web_browser)#/media/File:Chromium_Material_Icon.png

Příloha A

Obsah přiložené SD karty

Adresářová struktura na SD kartě přiložené u technické zprávy.

A.1 Adresářová struktura

- **build** obsahuje sestavenou aplikaci a technickou zprávu
  - **app** obsahuje přeložené zdrojové kódy *.go pro platformy Windows x86 a Linux x64.
  - **doc** obsahuje přeložené zdrojové kódy technické zprávy do formátu PDF
- **doc** obsahuje zdrojové kódy a použité obrázky pro překlad technické zprávy
- **app** obsahuje zdrojové kódy pro sestavení vytvářené aplikace

Příloha B

Manuál

B.1 Instalace nástrojů Go

Pro bezproblémový překlad a instalaci závislostí je potřeba nainstalovat golang minimálně ve verzi 1.11, protože až od této verze jsou podporované Go Modules. Aktuální verze golang je 1.12.5. Stažení instalačního balíčků pro konkrétní platformu lze z oficiálních webových stránek Go[1].

B.1.1 Windows

- Instalátor na Windows je distribuován formou MSI, který, ve výchozím stavu nainstaluje golang do adresáře c:/Go.
- Prvním krokem je spuštění instalátoru, který nastaví i proměnnou `PATH` v proměnném prostředí.
- Ověření správnosti instalace je takové, že z libovolného pracovního adresáře lze spustit příkaz:

  ```
  go
  ```

- Používanou verzi golang lze ověřit příkazem:

  ```
  go version
  ```

B.1.2 Linux

- Golang je pro Linux distribuován pomocí archívu *.tar.gz.
- Prvním krokem je rozbalení do staženého archívu do cílového adresáře, to lze provést příkazem:

  ```
  tar -C /usr/local -xzf go1.12.5.linux-amd64.tar.gz
  ```

- Dalším krokem je přidání cílového adresáře do proměnné `PATH` proměnného prostředí následujícím příkazem:

[1] https://golang.org/dl/

```
export PATH=$PATH:/usr/local/go/bin
```

- Ověření správnosti instalace je takové, že z libovolného pracovního adresáře lze spustit příkaz:

  ```
  go
  ```

- Používanou verzi golang lze ověřit příkazem:

  ```
  go version
  ```

B.2 Přeložení zdrojových kódu

Překlad na platformách Windows a Linux probíhá stejně. Ve výchozím stavu je aplikace sestavena pro tu platformu, kde byl spuštěn překlad. Toto chování lze změnit nastavením proměnných `GOOS` a `GOARCH` v proměnném prostředí. Validní hodnoty pro tyto proměnné jsou popsáné na stránce https://golang.org/doc/install/source#environment.

1. Stažení knihoven do lokální cache příkazem:

   ```
   go mod download
   ```

2. Přeložení zdrojových kódu lze příkazem:

   ```
   go build -o build/kiwi-ud.exe cmd/kiwi-ud
   ```

   Parametr -o specifikuje, kde bude přeložená aplikace uložena a pod jakým názvem. Adresář `cmd/kiwi-ud` říká, který vstupní bod má být přeložen. Pokud je jako adresář zadáno `./...`, pak se přeloží všechny vstupní body současně.

B.3 Spuštění aplikace

Ukázka spuštění aplikace s konfigurací přes:

- **parametry příkazové řádky**

  ```
  $ kiwi-ud.exe --service=geovap\
          --serviceURL="https://name:pass@10.10.10.24:1790/spsws.asmx?"\
          --storage=file --storageURL="file:///c:/ud-data/mesto"
  ```

- **proměnné prostředí**

  ```
  $ STORAGE=file SERVICE=udxmlv2\
  STORAGE_FILE="file:///c:/ud-data/mesto"\
  SERVICE_URL="https://10.10.10.24:8080/source.xml"\
      kiwi-ud.exe
  ```

B.4 Programová nápověda

Nápovědu programu lze zobrazit spuštěním aplikace bez parametrů nebo s parametrem `--help`. V hranatých závorkách za popisem parametru je název proměnné v proměnném prostředí pro alternativní nastavení.

```
Usage:
  kiwi-ud.exe [OPTIONS]

Application Options:
  --service=           Name of EDRMS provider [%SERVICE%]
  --serviceUrl=        URL of EDRMS web service or another
                      source [%SERVICE_URL%]
  --storage=           Name of target storage [%STORAGE%]
  --storageUrl=        File storage URL [%STORAGE_URL%]
  --outFormat=         Output format [%OUT_FORMATS%]
  --productionName=    Name for identify deployment [%PRODUCTION_NAME%]
  --timezone=          Settings of service timezone, if not available e.g.
                      Europe/Prague [%TIMEZONE%]
  --debug              Debug mode
  --version            Print program build info
```