FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání diplomové práce

Název:	Python SDK pro Data Stewardship Wizard
Student:	Bc. Jakub Drahoš
Vedoucí:	Ing. Marek Suchánek
Studijní program:	Informatika
Obor / specializace:	Webové a softwarové inženýrství, zaměření Webové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2021/2022

Pokyny pro vypracování

Data Stewardship Wizard (DSW) je univerzální nástroj pro generování dokumentů z dotazníků využívaný především pro tvorbu plánu správy dat. DSW poskytuje API formou webových služeb primárně své klientské aplikaci ale vznikají i různé další skripty a aplikace, které tyto služby využívají různými způsoby. Cílem této práce je zefektivnit vývoj těchto aplikací vytvořením kvalitního SDK:

- Seznamte se s DSW a popište skutečnosti důležité pro vývoj SDK.
- Proveďte stručnou rešerši rozšířených a moderních Python SDK, zaměřte se především na obecný návrh, dokumentaci a testování.
- Navrhněte Python SDK pro práci s DSW dle analýzy a rešerše. Návrh musí umožňovat snadné rozšiřování a návaznost na dané verze API.
- Implementujte SDK dle návrhu, zdokumentujte a připravte pro distribuci prostřednictvím PyPI.
- Připravte testy pro ověřování funkčnosti SDK a usnadnění aktualizací dle změn API v souladu s návrhem.
- Zhodnoťte přínosy SDK, zejména použitelnost, udržitelnost a budoucí rozvoj.

Elektronicky schválil/a Ing. Jaroslav Kuchař, Ph.D. dne 29. října 2020 v Praze.

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Diplomová práce

Python SDK pro Data Stewardship Wizard

Bc. Jakub Drahoš

Katedra softwarového inženýrství
Vedoucí práce: Ing. Marek Suchánek

6. května 2021

Poděkování

Tímto bych chtěl poděkovat vedoucímu své diplomové práce, Ing. Markovi Suchánkovi, za velmi vstřícné a odborné vedení. Také děkuji své přítelkyni a rodičům za bezmeznou podporu v průběhu celého studia.

Prohlášení

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

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 2373 odst. 2 zákona č. 89/2012 Sb., občanský zákoník, ve znění pozdějších předpisů, tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu) licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 6. května 2021 . . . . . .

Odkaz na tuto práci

Drahoš, Jakub. *Python SDK pro Data Stewardship Wizard*. Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2021. Dostupný také z WWW: ⟨`https://github.com/ds-wizard/dsw-sdk`⟩.

Abstrakt

Tato práce se zabývá tvorbou SDK v jazyce Python pro webovou aplikaci Data Stewardship Wizard. Architektura a návrh knihovny nachází inspiraci v nejrozšířenějších Pythonních SDK současnosti, jejichž rešerše pomohla stanovit řešení určitých problémů. Zároveň zohledňuje zavedené postupy a aspekty pro posuzování kvality softwaru. Je tak kladen důraz na vysokou kvalitu kódu, rozšiřitelnost s ohledem na měnící se API a snadnou použitelnost. SDK bylo úspěšně navrženo, implementováno, otestováno a distribuováno na softwarový repozitář PyPI. V rámci popisu implementace se nachází také jednoduché představení klíčové funkcionality. Práce zároveň poskytuje čtenáři letmý úvod k nástroji Data Stewardship Wizard.

Klíčová slova Data Stewardship Wizard, DSW, Software Development Kit, SDK, Python, knihovna

Abstract

This thesis focuses on the creation of a Python SDK for the Data Stewardship Wizard web application. The architecture and design of the software library are inspired by the most widespread Python SDKs of the present,

whose research has helped to identify solutions to certain obstacles. Established procedures and aspects for software quality assessment are also taken into account. Therefore the emphasis is on high-quality code, extensibility considering the changing API, and ease of use. The SDK has been successfully designed, implemented, tested, and distributed on the PyPI software repository. There is also a simple introduction of the key functionality within the description of the implementation. The thesis also provides the reader with a casual introduction to the Data Stewardship Wizard tool.

Keywords Data Stewardship Wizard, DSW, Software Development Kit, SDK, Python, library

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Data Stewardship Wizard (dále jen DSW) je nástroj spojující dohromady výzkumníky z nejrůznějších oborů a tzv. *data stewards*, jenž poskytuje možnost jednoduché a efektivní tvorby *data management plánů* (dále jen DMP). Tito data stewards zprostředkovávají své znalosti a zkušenosti z oblasti data managementu skrze DSW výzkumníkům, a de facto je tak navigují celým procesem tvorby DMP. [1]

Nespornou výhodou tohoto systému (mimo fakt, že DMP je v dnešní době velmi často vyžadován pro financování výzkumů) je pak především předání znalostí o správném zacházení s daty – jak splňovat principy *FAIR*, jak data udržovat během celého výzkumu a jak je spravovat z dlouhodobého hlediska. [2]

Motivace

DSW posyktuje API primárně vlastní klientské aplikace, nicméně v rámci projektu existuje několik podpůrných nástrojů napsaných v jazyce Python (*Document Worker* pro generování dokumentů – samotných data management plánů; *Template Development Kit* umožňující tvorbu a úpravu šablon z nichž se dokumenty generují), které s API komunikují vlastním způsobem. Během workshopů si organizátoři často vytváří testovací data (např. uživatele) ručně nebo pomocí vlastních API skriptů. Správci DSW instancí potřebují tvořit skripty pro hromadné aktualizace či ke správě konfigurace.

Sdílená a snadno udržitelná knihovna, jež poskytne objektový přístup k API, by mohla velmi usnadnit vývoj výše zmíněných aplikací a zároveň nabídnout jednotný způsob, jak implementovat vlastní skripty pro interakci s DSW.

Cíle práce

Hlavním cílem této práce je tedy tvorba SDK pro jazyk Python, která usnadní libovolnou programatickou práci s DSW. Bude dodržen klasický softwarový cyklus – analýza, design, implementace a testování. Návrh musí zohlednit snadnou rozšiřitelnost s ohledem na měnící se API a návaznost na dané verze. Implementace by měla klást důraz na vysokou kvalitu kódu a jeho udržitelnost z pohledu budoucího rozvoje. Důležitým prvkem je také kvalitní a srozumitelná dokumentace usnadňující korektní použití.

Knihovna bude distribuována prostřednictvím repozitáře PyPI pro zajištění snadné dostupnosti.

Vedlejším cílem práce je seznámení čtenáře s nástrojem Data Stewardship Wizard, rešerše nejrozšířenějších SDK pro jazyk Python a shrnutí obecných *best practices* při návrhu a implementaci softwarové knihovny.

Struktura práce

Práce je členěna do celkem 4 kapitol. První se zabývá představením nástroje Data Stewardship Wizard, jeho architekturou a zhodnocením oblíbených Pythonní SDK, zejména z hlediska jejich návrhu a použitelnosti. V závěru této kapitoly se pokusím analyzovat důležité aspekty kvalitního a udržitelného kódu. Následuje shrnutí případů užití, vymezení funkčních a nefunkčních požadavků a z nich plynoucí návrh architektury celé knihovny. Ve třetí kapitole se věnuji popisu implementace, designových rozhodnutí a stručně představím použití SDK. Poslední kapitola je zaměřena na způsoby testování a distribuci knihovny. Ke konci také shrnu naplnění vytyčených požadavků a následující kroky v budoucím vývoji. V samotném závěru pak vyhodnotím cíle zmíněné výše a zhodnotím přínosy SDK.

Kapitola 1

Analýza

Následující kapitolu lze rozdělit do 3 samostatných celků:

1. Seznámení s nástrojem Data Stewardship Wizard. Začnu s účelem celého projektu, přejdu k funkcionalitě a odtud k jeho softwarové architektuře, API a datovému modelu.
2. Průzkum a rešerše stávajících Python SDK, která fungují a jsou prověřená časem.
3. V poslední části se zaměřím na obecné aspekty při psaní softwarové knihovny, nezávislé na konkrétním programovacím jazyku – snadnou rozšiřitelnost, jednoduché použití, udržitelnost či důležitost dokumentace.

1.1 Data Stewardship Wizard

Jak bylo již uvedeno v úvodu, Data Stewardship Wizard je nástroj umožňující snadnou tvorbu data management plánů. Jedná se však o univerzální nástroj pro generování dokumentů z dotazníků, kde jsou otázky hierarchické (teoreticky neomezeně „hluboko“). Existují různé možnosti užití tohoto nástroje, například generování právních dokumentů a smluv či sběr *Case Report Forms* (jedná se o dotazníky specificky využívané při klinických studiích) o pacientech v nemocnicích (tento způsob užití aplikace byl skutečně realizován). Nicméně jeho momentálně hlavním využitím je právě data stewardship a tvorba plánů správy dat. Těmi se tedy budu zabývat i v rámci této části, popisující funkcionalitu a architekturu DSW. [3]

1.1.1 Vysvětlení hlavních pojmů

Data management plan neboli plán, jak se bude nakládat s datovými zdroji a potenciálními produkty. Popisuje data, která budou získána nebo

produkována během výzkumu, jak tato data budou uložena, popsána, v jakém formátu nebo co se s daty bude dít po skončení výzkumu a zda (a kde) budou dostupná. [4]

Data steward je expert se znalostí data managementu, jenž zároveň velmi dobře zná konkrétní doménu, ve které se data nachází. Ví, jak se o data starat, jak je spravovat a jak je správně poskytovat ostatním a vytěžit z nich tak jejich maximální potenciál. [5]

FAIR je sada obecných principů pro management vědeckých dat. Kladou důraz především na tzv. *machine-actionability*, neboli schopnost výpočetních systémů hledat, získávat a používat data bez žádné nebo minimální lidské intervence. [6]

Jejich detailnějšímu popisu se bude věnovat následující sekce.

1.1.1.1 FAIR data

FAIR je akronym pro 4 základní principy. Pod každým konceptem se ukrývá několik dílčích pravidel, která se týkají 3 základních entit: dat, metadat (informace o datech) a infrastuktury. Pokud naše data tyto zásady budou dodržovat, jejich konzumace, správa a následné znovupoužití bude o mnoho jednodušší. Navíc budou lehce strojově zpracovatelná a jejich využití tak bude ještě širší. [6]

Následující seznam pravidel je čerpán z [7]:

Findability Prvním krokem k znovupoužití dat je jejich nalezení. Proto by měla data (spolu s metadaty) být lehce dohledatelná jak lidmi, tak stroji.

- **F1** (Meta)data mají přiřazen globálně unikátní a perzistentní identifikátor.
- **F2** Data jsou popsána metadaty.
- **F3** Metadata jasně a explicitně zahrnují identifikátor dat, jež popisují.
- **F4** (Meta)data jsou registrována nebo indexována v nějaké službě, která zprostředkovává jejich dohledatelnost (např. indexována Googlem nebo registrována v datovém katalogu)

Accessibility Jakmile jsou data nalezena (ať už člověkem či strojem), jejich konzument musí znát způsob, jak k datům přistupovat, případně jak se autentizovat a autorizovat.

- **A1**. (Meta)data jsou dostupná pod jejich identifikátorem a lze k nim přistoupit pomocí standardizovaných komunikačních protokolů (např. HTTP).

  A1.1 Výše zmíněný protokol je otevřený, zdarma a kýmkoli implementovatelný.

  A1.2 Protokol umožňuje autentizaci a autorizaci, pokud je to vyžadováno.

- **A2**. Metadata jsou dostupná i přesto, že samotná data (která tato metadata popisují) již dostupná nejsou.

Interoperability Data by měla být propojena mezi sebou a navzájem spolu interagovat.

- **I1**. (Meta)data používají formální, dostupný a obecně známý jazyk pro svou reprezentaci (skvělým příkladem jsou RDF nebo JSON-LD).
- **I2**. (Meta)data používají slovníky, které také dodržují FAIR principy.
- **I3**. (Meta)data obsahují smysluplné (a co nejvíce popisné) odkazy na další (meta)data.

Reusability Hlavním cílem FAIR principů je optimalizovat znovupoužívání a recyklaci dat. Abychom toho docílili, měla by data (a metadata) být velmi dobře popsána, aby bylo možné je replikovat a/nebo kombinovat.

- **R1** (Meta)data jsou popsána množstvím přesných a relevantních atributů.

  R1.1 (Meta)data jsou vydávána pod jasnou a dostupnou licencí o použití dat.

  R1.2 U (meta)dat je uveden detailní původ.

  R1.3 (Meta)data dodržují doménově závislé standardy.

Tyto principy nám tedy mohou velmi napovědět, jak s daty zacházet a jak vytvářet případný *data management plan*.

1.1.1.2 FAIR vs. Open data

Pozornému čtenáři jistě neunikne jistá podobnost FAIR principů s konceptem Linked Open Data (neboli LOD, více info např. [8]). A vskutku, tyto dva přístupy jsou v některých ohledech podobné, avšak spíš se doplňují, než že by se navzájem duplikovaly či se snažily spolu soupeřit.

Zatímco LOD lze chápat jako určitý protokol či standard, FAIR je spíše soubor charakteristik, které by každá datová sada (nebo obecně libovolný předmět výzkumu) měla mít, pokud má být někomu prospěšná. [9]

Hlavním rozdílem, který je třeba rozlišovat, je pravděpodobně přístup k oné „otevřenosti“ dat. Jelikož se FAIR principy vztahují hlavně k oblasti vědeckých dat, je zcela pochopitelné, že ne všechna data mohou být volně publikována (ať už třeba z etických důvodů nebo z důvodů ochrany osobních údajů). FAIR tedy neklade žádné nároky na otevřenost dat a záměrně neadresuje etické a morální problémy týkající se této otevřenosti. [10]

Naproti tomu LOD (ačkoli nemají nijak omezenou doménu své působnosti) se uplatňují především v oblasti státní moci a veřejné správy, z čehož pramení i jejich důraz na celkovou transparentnost a otevřenost dat. Každý by měl mít právo otevřená data zkoumat, užívat je, kombinovat a sdílet. [11]

1.1.2 Funkcionalita DSW

Popis funkcionality DSW vychází z prezentace představující nástroj [12], oficiálního webu [1] a oficiální dokumentace [14]; jelikož se však zdroje prolínají, pro přehlednost je uvádím souhrnně zde. Některé informace mohou být zároveň založené na mé osobní zkušenosti s DSW.

Hlavním cílem celého projektu je jakési mentorování a vzdělávání výzkumníků v oblasti data managementu.

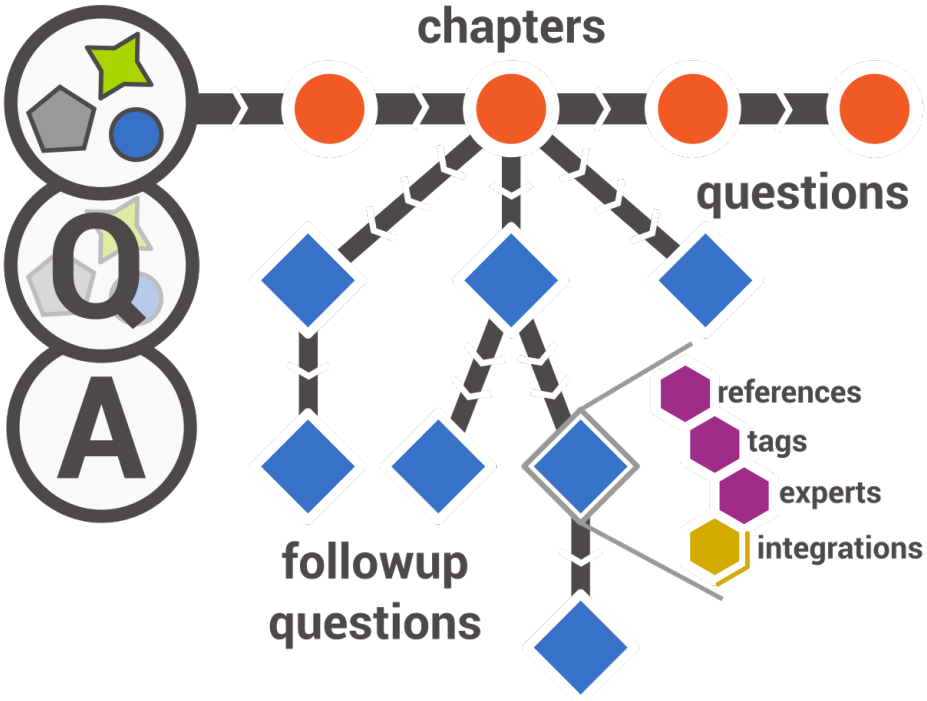
Co se praktické stránky týče, celá funkcionalita DSW se točí kolem tvorby *data management plánů* na základě vyplňování interaktivních dotazníků(uživateli se zobrazují jen ty otázky, které jsou pro něj relevantní na základě předchozích odpovědí). Každý dotazník vychází z tzv. *knowledge modelu*, který je v kompetenci data stewardů.

Následuje stručný popis jednotlivých entit, které spolu dohromady tvoří jádro celého systému:

Knowledge model v podstatě reprezentuje znalosti, vědomosti a zkušenosti potřebné ke správě dat v dané doméně. Tyto informace pocházejí od data stewardů, kteří daný knowledge model vytvářejí. Fakticky se jedná o stromovou strukturu základních knowledge model elementů, vizte 1.1.

Knowledge model element je esenciální stavební prvek knowledge modelu. Rozděluje se na:

- *Kapitoly* – logické celky shromažďující dohromady otázky, které k sobě patří.
- *Otázky* – základ dotazníku; každá otázka obsahuje stručný popis a ve které fázi projektu je dobré ji mít zodpovězenou; dále může obsahovat různé reference (odkazy na detailnější informace, pasáže z knihy *Data Stewardship for Open Science* od Barenda Monse [13]) nebo kontakty na experty v dané problematice.



Obrázek 1.1: Struktura knowledge modelu [15]

- *Odpovědi* jsou různého typu; DSW zároveň k některým odpovědím nabízí doporučení (např. pokud je uživatelem vybraná možnost velmi nepravděpodobná, vybízí k přehodnocení).
- *Integrace* – jedná se o speciální druh odpovědi, který po vzoru Linked data propojuje DSW s cizími službami a dokáže tak poskytnout více relevantních informací; zároveň ve finálním DMP vytváří odkazy na uživatelem uvedenou hodnotu (např. zadávání institucí může být zpracováno formou integrace).
- *Štítky*, pomocí kterých můžeme rozdělit otázky do různých podmnožin (ku příkladu označit všechny otázky, které jsou povinné pro Horizon 2020 DMP).

Dotazník je reprezentace vybraného knowledge modelu v podobě interaktivního formuláře, který vyplňují výzkumníci.

Šablona je předpis, jak má konkrétní DMP vypadat. Výsledkem kombinace dat z vyplněného dotazníku a určité šablony je pak samotný DMP. Jelikož

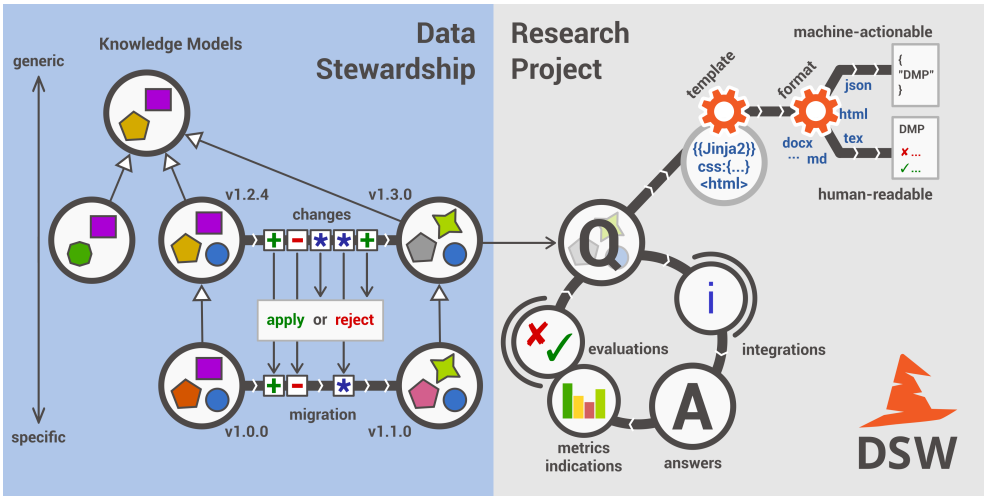
existuje mnoho různých forem DMP (každý vyžaduje jinou strukturu, jiné informace), lze si šablony přizpůsobovat a každá organizace tak může mít své vlastní.

Dokument je již samotný data management plán dle konkrétní šablony a v konkrétním datovém formátu.

Samotná funkcionalita je pak rozdělena podle uživatelských rolí. DSW pracuje se 3 druhy autorizovaných uživatelů (výzkumník, data steward, administrátor) a s jedním neautorizovaným. Vyšší role mohou dělat to stejné co nižší role, avšak vždy mají zpřístupněnou určitou funkcionalitu navíc.

1. *Anonymní uživatel* neboli uživatel, který není přihlášený. Může se zaregistrovat, přihlásit nebo požádat o obnovu hesla. Díky systému sdílení se může také plně spolupodílet na vyplňování dotazníků (pokud to má daný dotazník/projekt povolené).
2. *Výzkumník* je uživatel, který pracuje na určitém vědeckém projektu a disponuje znalostmi z dané domény výzkumu. Jeho hlavním cílem je vytvoření kvalitního DMP naplňující FAIR metriky a nabytí znalostí, jak správně zacházet s daty během výzkumu. Vyplňuje tedy dotazníky, ze kterých pak dle šablon tvoří samotné DMP.
3. *Data steward* je typ uživatele, který má dobrou znalost data stewardship domény (ví, jak správně zacházet s daty). Tyto znalosti transformuje do knowledge modelů; má tedy přístup do knowledge model editoru, kde může jednotlivé modely mazat, upravovat, exportovat nebo vytvářet.
4. *Administrátor* má na starosti celkové nastavení DSW instance a jako takový má tu nejvyšší úroveň oprávnění. Spravuje nastavení organizace, jednotlivé uživatele a další konfigurace aplikace.

Obrázek 1.2 popisuje hlavní workflow nástroje DSW. Vidíme dva odlišné „světy“ – na levé straně doménu data stewardshipu, pod kterou spadá správa knowledge modelů. Ty ze sebe mohou navzájem vycházet (nahoře jsou obecnější zatímco dole jsou konkrétnější), jako je tomu například u objektového programovacího paradigmatu (dědičnost). Každá změna v modelu znamená novou verzi (jednotlivé modely jsou *immutable* – neměnné). Vpravo pak vidíme část DSW, která je přístupná výzkumníkům. Celý proces tvorby DMP je vlastně cyklus – výzkumník prochází jednotlivé otázky, k těmto otázkám získává informace (např. pomocí integrací nebo odkazů); na základě těchto informací dokáže poskytnout odpověď a dle svého rozhodnutí zhodnotit, jak si vede v různých metrikách (to za něj dělá samozřejmě wizard). Z vyplněného dotazníku se pak podle konkrétní šablony vytvoří DMP (v libovolném formátu; tedy pokud chceme strojově zpracovatelný plán, zvolíme třeba RDF, pro zpracování lidmi nejspíš PDF nebo Word formát).



Obrázek 1.2: DSW worklfow [14]

1.1.3 Architektura DSW

Celý DSW není jedna monolitická struktura, ale skládá se z několika oddělených komponent, které spolu vzájemně komunikují (vizte diagram 1.3). Hlavním prvkem je *Engine Wizard*, který vystavuje RESTové API. S tímto API pak komunikuje frontendová část Wizardu (uživatelské webové rozhraní) a dohromady tvoří základ celého nástroje. [16] [17]

Jako samostatná služba běží Data Stewardship Registry (opět backendová část vystavující REST API a k tomu frontendová část zprostředkovávající uživatelské rozhraní). Tato služba slouží jako sdílené úložiště pro knowledge modely a šablony DMP. [16] [17]

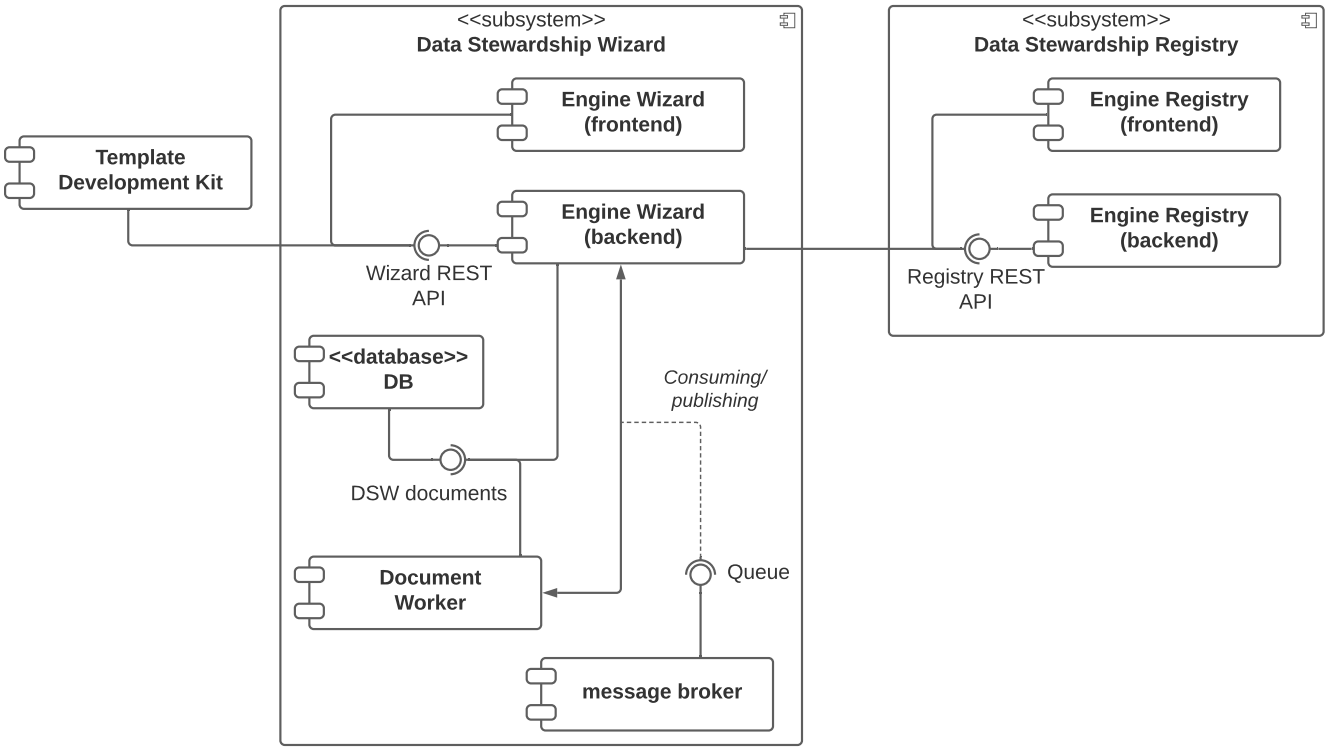
Další nedílnou součástí je tzv. *Document Worker*, jehož starostí je generování samotných DMP dokumentů z vyplněných dotazníků. Komunikuje s DSW backendem pomocí fronty (message broker), do které se zapisují žádosti o tvorbu dokumentu. *Document Worker* na tuto žádost získá z databáze detaily dokumentu, vygeneruje ho a uloží na sdílený file system, odkud ho DSW poskytuje uživatelům (ať už přes webovou aplikaci či přes API). [18]

Podpůrným nástrojem pro tvorbu nových šablon DMP je *Template Development Kit*. Ten komunikuje s API DSW, přes které publikuje vytvořené šablony. [19]

1.1.4 Popis API a datový model

Diagram na obrázku 1.4 zobrazuje jednotlivé datové entity domény DSW a jak jsou spolu navzájem provázány.

Můžeme si všimnout, že z pohledu datového modelu je knowledge model jakousi abstrakcí a reálně ho reprezentují 2 entity – *knowledge model pac-*



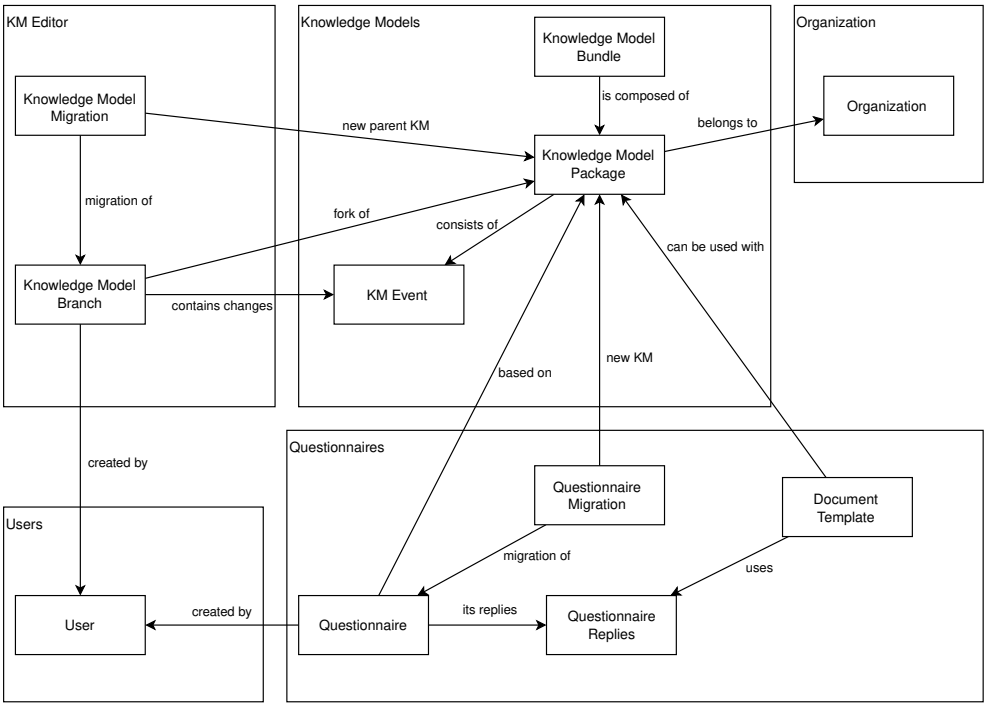
Obrázek 1.3: DSW diagram komponent [vytvořil autor]

kage a *knowledge model branch*. První je knowledge model publikovaný jako balíček na Data Stewardship Registry (tedy je veřejně dostupný i mimo danou instanci DSW) a mimo jiné obsahuje informace o instituci, jež ho spravuje. Jedná se o stabilní verzi knowledge modelu, ze kterého mohou výzkumníci tvořit dotazníky. *Knowledge model branch* je pak verze odvětvená od určitého knowledge model package a jedná se o rozpracovanou verzi knowledge modelu. Tato není dostupná jako základ pro tvorbu dotazníků (to nastane až po publikování, kdy se z branche stává package). [14] [16] [20]

Exportováním nějakého knowledge modelu dostaneme *Knowledge model bundle*.

Za povšimnutí také stojí, že interně jsou knowledge modely reprezentovány jako kolekce událostí – *KM eventů*. Každé přidání nebo odebrání otázky či odpovědi vlastně vytvoří novou událost. To má několik příjemných benefitů, například snadné verzování změn či redukci celkové paměťové náročnosti, neboť každý knowledge model může obsahovat pouze události, ve kterých se liší od svého předka (pokud nějakého má). [14] [20]

Co se týče RESTového API nástroje DSW, to poskytuje mnoho funkcionality; tabulka 1.1 se tedy zaměřuje na popis pouze těch částí, které jsou relevantní pro tuto práci. Zároveň lze rozhraní rozčlenit dle entit, se kterými



Obrázek 1.4: DSW doménový diagram [14]

se pracuje a lépe tak strukturovat charakteristiku jednotlivých endpointů. Kompletní Swagger dokumentaci lze nalézt na odkazu `https://api.demo.ds-wizard.org/swagger-ui/`. [20]

Tabulka 1.1: API endpointy

Entita	Funkce endpointů
Uživatelé	CRUD operace změna hesla
Autentizace/autorizace	získání přihlašovacího tokenu OAuth
KM packages	CRD operace import a export aktualizace KM z Data Stewardship Registry
KM branches	CRUD operace migrace větve na nový KM
Dokumenty (data management plány)	CRD operace stažení dokumentu
Dotazníky	CRUD operace reporty metrik vygenerované dokumenty pro daný dotazník migrace dotazníků na nový KM
Šablony	CRUD operace assety a šablony samotné import a export aktualizace šablony z Data Stewardship Registry

1.2 Rešerše stávajících SDK

Na úvod bych rád začal vysvětlením, co to vlastně SDK je. SDK neboli Software Development Kit je sada vývojářských nástrojů a knihoven specificky používaných k tvorbě softwaru pro danou cílovou platformu. Hlavním cílem je usnadnit programátorům celý vývojový cyklus (včetně například testování) a poskytnout jim kompletní sadu prostředků k realizaci jejich cíle. [21]

V kontextu webových služeb však může s pojmem SDK splývat výraz *knihovna* (anglicky *software library*), nicméně jsou zde lehce odlišné konotace. Knihovna by měla řešit jeden ucelený problém a nic víc (můžeme si představit knihovnu pro HTTP komunikaci). Naproti tomu SDK zprostředkovává práci s celou platformou (představme si operační systém, ale v rámci internetu je to spíš nějaká webová služba nebo nástroj). Dokonce ani nemusí obsahovat samotnou logiku (kód, pro nějž SDK vytváří rozhraní); ta se může nacházet na vzdáleném serveru a být přístupná skrze webové API, což je v dnešní době velmi oblíbený přístup, jak si ukážeme v následujících sekcích. Součástí SDK bývají často také různé pomocné nástroje (třeba CLI utility). [22, 23]

Na druhou stranu toto rozlišení není nijak striktní a pojmy SDK i knihovna se běžně používají zaměnitelně. Častým synonymem je také výraz *klient* nebo *klientská knihovna*. Dle [24] dokonce nic jako SDK pro webové API v dnešní

době neexistuje a uvádí, že jde spíš o slovíčkaření. S dovolením tak budu i já sám používat v tomto textu výrazy „SDK“ a „knihovna“ záměnně, vždy však budu mít na mysli jedno a to samé.

Nyní si můžeme představit některé známé SDK napsané pro programovací jazyk Python, zaměřit se na jejich funkcionalitu, architekturu, přístupy k různým problematikám (jako například k testování či dokumentaci) a obecně na jejich návrh. Výběr těchto knihoven nebyl náhodný, ale byl založen na určitých metrikách z GitHub repozitářů – počet uživatelů a stáří prvního commitu.

- Boto3 – 98 tisíc uživatelů, stáří cca 6 let.
- Docker SDK – 18 a půl tisíce uživatelů, stáří přes 7 let.
- Sentry SDK – 9 tisíc uživatelů, stáří stáří 3 roky.

1.2.1 Boto3

```
1   import boto3
2
3   s3 = boto3.resource('s3')
4   for bucket in s3.buckets.all():
5       print(bucket.name)
6
7   s3_low_level = boto3.client('s3')
8   for bucket in s3_low_level.list_buckets()['Buckets']:
9       print(bucket['name'])
```

Obrázek 1.5: Vysokoúrovňové a nízkoúrovňové rozhraní Boto3

Následující popis čerpá z oficiálního GitHub repozitáře [25] a příslušné dokumentace [26].

Boto3 je Software Development Kit pro platformu AWS (*Amazon Web Services*). Umožňuje přesně to, co bylo zmíněno v úvodu této sekce – AWS provozuje mnoho různých služeb a každá z těchto služeb vystavuje webové rozhraní (API). Boto3 tato API konzumuje a poskytuje jejich funkcionalitu nativně v Pythonním kódu.

Zajímavý je přístup, který vývojáři z Amazonu zvolili. Všechny třídy jsou generované až za běhu programu, tedy k nim nelze předem nijak přistupovat nebo z nich dědit, a rozšiřovat tak jejich funkcionalitu. Místo dědičnosti lze však využít tzv. *systém událostí*. Uživatelé (programátoři) mohou definovat funkce (callbacky), které se mají zavolat ve chvíli, kdy nastane nějaká událost. Díky tomu je pak možné libovolně rozšiřovat logiku tříd.

Výše zmíněný princip má velkou výhodu. Generování tříd je řízeno JSON modely, jež popisují AWS API. To dovoluje vývojovému týmu poskytovat

velmi rychlé aktualizace se silnou konzistencí napříč všemi nabízenými službami.

Mezi další body, které stojí za zmínku, patří:

Konfigurace je řešena na 3 úrovních: objekt předaný při inicializaci, environment variables a soubor s nastavením (jejich precedence odpovídá tomuto pořadí). Tedy obecné hodnoty uložíme do souboru (ty může využít vícero instancí) a nastavení například pro konkrétní prostředí (produkce vs. vývoj) můžeme specifikovat pomocí environment variables. Pro snadnou konfiguraci testů pak lze předávat objekt, který má nejvyšší prioritu.

Clients je low-level rozhraní, které 1:1 mapuje webové API. Umožňuje nízkoúrovňovou interakci s AWS službami, v podstatě jen obaluje HTTP komunikaci a poskytuje patřičné metody odpovídající vzdáleným endpointům. Data se vrací jako Pythonní slovníky (deserializovaná JSON odpověď ze serveru) a je na programátorovi, aby si zpracoval potřebné informace. Tento přístup se může hodit pro pokrytí veškeré (i nějaké obskurnější) funkcionality, neboť vždy zrcadlí komplet celé webové API, navíc díky dynamickému generování bez potřeby cokoli upravovat. Zároveň je toto rozhraní thread-safe.

Resources je rozhraní poskytující naopak vyšší úroveň abstrakce a objektově orientovaný přístup k AWS službám. Cílem je usnadnit běžné akce, oprostit programátora od různých implementačních detailů, a zároveň poskytnout optimalizovaný přístup k webovému rozhraní AWS (např. lazy-loaded atributy nebo transparentní iterace přes kolekce, která schovává logiku stránkování). Narozdíl od „clients“ rozhraní nejsou thread-safe (udržuje se zde sdílený stav).

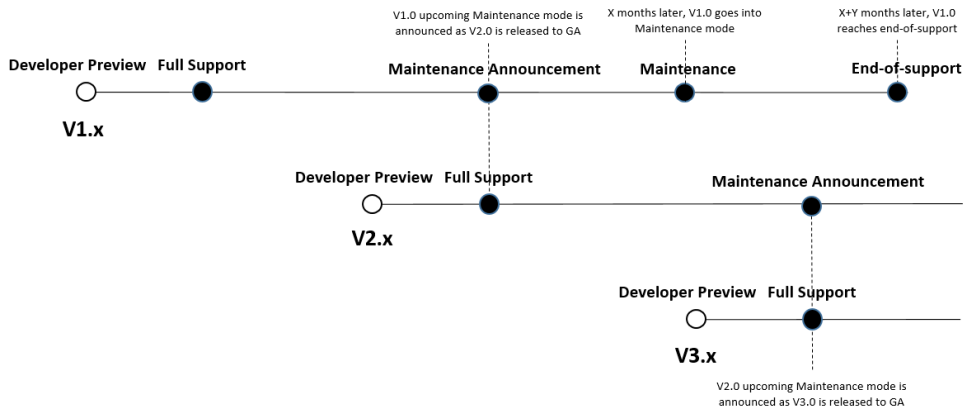
Verzování probíhá klasicky podle specifikace sémantického verzování[28]. Každá jejich major verze navíc podléhá jasně definovanému životnímu cyklu znázorněnému na obrázku 1.6.

Testování probíhá podle standardních zvyklostí ekosytsému jazyka Python. Testují se všechny podporované verze Pythonu (2.7, 3.6, 3.7, 3.8) nástrojem `tox`. Je využito jednotkových, integračních a funkčních testů.

1.2.2 Sentry SDK

Informace této sekce pramení z dokumentace [29] a oficiálního GitHub repozitáře [30].

Sentry je software pro monitorování aplikací (používá se hlavně k reportování chyb, ale také ke sledování a optimalizaci výkonu). Sentry SDK pak umožňuje integrovat zmíněnou funkcionalitu přímo do aplikace, přičemž se snaží cílit na velmi jednoduchou a transparentní použitelnost.



Obrázek 1.6: Životní cyklus major verze AWS SDK [27]

```
import sentry_sdk

def custom_hook(event, hint):
    # Modify event here or add hints
    return event

sentry_sdk.init(
    'https://mydsn@sentry.io/123',
    before_send=custom_hook,
)
raise ValueError()  # Will create event
```

Obrázek 1.7: Ukázka rozšíření funkcionality Sentry SDK o dodatečnou logiku

Konfigurace probíhá opět buď pomocí souboru, environment variables či při inicializaci celého SDK. Kromě základního nastavení je možné také ovlivnit transportní vrstvu (například pro komplexnější autentizaci či změnu HTTP proxy) nebo využít tzv. *hooks* – uživatel může poskytnout callback, který se má zavolat v určité situaci (jako příklad může sloužit funkce `before_send`, jež se volá před reportováním chyby, kde lze anonymizovat data nebo přidat dodatečné informace). Tyto funkce umožňující uživatelům rozšiřovat funkcionalitu hlavního workflow jsou velmi důležité, neboť významně přispívají k použitelnosti a hlavně rozšiřitelnosti celého nástroje.

Dokumentace klade důraz na používání jednoduchého jazyka, nezatíženého doménově specifickým žargonem. Jak sami autoři zmiňují: „předpokládejme vždy vývojáře neobeznámené s problematikou“. Dokumentace (API reference) je automaticky generována pomocí nástroje `Sphinx` – popis jednotlivých elementů se bere přímo ze zdrojových kódů (docstringů). Veřejné API SDK musí tedy obsahovat smysluplný popis u svých tříd/funkcí/metod.

Závislosti jsou brány velmi seriózně, obzvlášť pak v Pythonní implementaci SDK – zatímco ve frontendové JavaScriptové verzi jsou nové závislosti většinou vítané, na backendových platformách je na ně nahlíženo daleko konzervativněji. Libovolná knihovna může představovat potenciální bezpečnostní hrozbu, jelikož může na serveru spustit škodlivý kód a dostat se tak k citlivým údajům všech uživatelů. Zároveň je těžší udržovat více podporovaných verzí, čím více těchto závislostí je (nezřídka se stává, že se určitá knihovna přestane spravovat).

Testování je rozděleno na jednotkové a integrační. Za zmínku stojí, že integrační testy testují zároveň integraci s jinými knihovnami a frameworky (např. Flask, Django nebo Sqlalchemy). Mezi další podpůrné nástroje využité tímto projektem patří Flake8 (dodržování PEP8 coding stylu, statická analýza), Codecov (pokrytí testy) nebo MyPy (statické typování dle PEP484).

1.2.3 Docker SDK

```
1     def create_container(self, ...):
2     """
3     Creates a container. Parameters are ...
4
5     .. code-block:: python
6
7       container_id = cli.create_container(
8         'busybox', 'ls', ports=[1111, 2222],
9         host_config=cli.create_host_config(port_bindings={
10          1111: 4567,
11          2222: None
12        })
13      )
14    """
```

Obrázek 1.8: Ukázka příkladu použití přímo ve zdrojovém kódu Docker SDK

Následující informace jsou převzaty z dokumentace nástroje [31] a oficiálního GitHub repozitáře [32].

Docker pomáhá vývojářům vytvářet přenositelné aplikace pomocí tzv. kontejnerů. Toto SDK pak slouží pro nativní práci s Docker Engine API v jazyce Python (umožňuje tak dělat vše, co lze s CLI nástrojem `docker`).

Stejně jako Boto3, i Docker SDK obsahuje 2 úrovně rozhraní. Objektově orientovaný `DockerClient` a nízkoúrovňový `APIClient`, jenž tvoří základ prvního zmíněného. Z toho plynou stejné benefity (větší flexibilita pro uživatele, kteří požadují pokročilejší funkcionalitu; objektový přístup při používání vysokoúrovňového rozhraní), avšak i záporné stránky – v tomto případě nejsou třídy generovány za běhu programu, ale jsou statické, což ústí v křehkost

nízkoúrovňového rozhraní, které musí vždy 1:1 zrcadlit vzdálené API. Veškeré změny se pak musejí provádět ručně.

Dokumentace je opět generována pomocí nástroje `Sphinx`, navíc však obsahuje přímo v kódu (jako součást docstringu třídy/metody/funkce) ukázky použití, což velmi zvyšuje přidanou hodnotu.

1.3 Best practices při tvorbě SDK

Při tvorbě libovolného softwaru je třeba dbát na různé aspekty, kterými lze souhrnně posuzovat kvalitu kódu. Při psaní knihoven a SDK je však nutné upřednostnit některé principy před jinými, neboť i způsob využití je jiný. Softwarové knihovny jsou používány zpravidla jinými programátory, než kteří danou knihovnu/SDK vyvíjejí a spravují, což klade na výsledný produkt určité nároky. Následující popis se věnuje důležitým aspektům vývoje nastíněných v [33] a [34].

1.3.1 Jednoduchost

Jednoduchost užívání je pravděpodobně tím nejdůležitějším kritériem pro úspěch. Nástroj, který se používá těžkopádně a má dlouhou a strmou křivku učení, nebude nikým příliš oblíbený.

Existuje mnoho přístupů, jak tuto problematiku řešit – vystavovat jednoduché rozhraní, s minimálním počtem funkcí a metod (pečlivě zvážit, které jsou opravu potřeba), minimální počet vstupních parametrů, poskytovat rozumné a očekávané výchozí hodnoty (jež lze ale snadno přetížit/změnit). S tím nám mohou pomoci i známé návrhové vzory jako je *Facade*, *Bridge* nebo *Builder*. [35, 36]

1.3.2 Dokumentace

Dokumentace musí být popisná, jasná, ale zároveň výstižná a ne příliš dlouhá. Všechny veřejné části (ty, s nimiž budou ostatní interagovat) musí být důkladně zdokumentovány. V ideálním případě by neměly chybět příklady použití jednotlivých funkcí/tříd. Také krátký návod, jak začít a sestavit minimální funkční celek může velmi pomoci. Čím více dokážeme minimalizovat čas do prvního použití a nastartování aplikace, tím lépe. Užitečný může být i nějaký ukázkový mini projekt, jenž prezentuje nabízenou funkcionalitu. [36, 37]

1.3.3 Rozšiřitelnost

Tento aspekt je z pohledu SDK velmi důležitý, neboť každé SDK do určité míry zrcadlí rozhraní nějaké platformy. Málokterý systém je však neměnný, a tak je flexibilní architektura, jež dovoluje snadno a rychle rozšiřovat stávající funkcionalitu, nepostradatelná.

Na rozšiřitelnost bychom se však měli dívat i z jiného pohledu – a sice z pohledu uživatele knihovny. Náš software by měl svým uživatelům poskytovat prostředky pro jejich vlastní reimplementace – někdo se může rozhodnout, že chce například trochu jinak zpracovávat požadavky na síťové vrstvě; či používat odlišný nástroj pro logování, než který je výchozí. Obecně bychom se v tomto případě měli držet softwarového open-closed principu. [34]

1.3.4 Standardizace

Každý projekt by se měl řídit určitými standardy a dodržovat konvence typické pro danou doménu. Při takovémto postupu citelně snižujeme náročnost integrace s ostatními systémy, činíme náš produkt o poznání srozumitelnější pro ostatní a v neposlední řadě ulehčujeme implementaci budoucích modifikací [34]. Mezi příklady takových standardů a konvencí patří:

- Obecně známé a uznávané architektonické a návrhové vzory.
- Jasně definovaný proces verzování (ideálně semver [28]).
- Jmenné konvence (pojmenování tříd/metod/funkcí/proměnných).
- Konvence spojené s daným programovacím jazykem (v případě Pythonu např. PEP8 nebo *The Zen of Python*).
- V neposlední řadě také předem určený vývojový cyklus a licencování, čímž značně usnadníme ostatním vývojářům přispívání do našeho projektu.

1.3.5 Udržitelnost

Slovník pojmů terminologie softwarového inženýrství IEEE [38] definuje pojem *udržitelnost* (orig. *maintainability*) jakožto proces úpravy softwaru po jeho nasazení/distribuci. Podle účelu této úpravy pak zavádí 3 různé druhy udržitelnosti:

- Adaptivní údržba (*adaptive maintenance*) se zaměřuje na adaptaci softwaru, aby byl použitelný v měnících se prostředích nebo v nových business procesech (jedná se jak o implementaci nové funkcionality, tak o možnost znovupoužití produktu v jiných podmínkách).
- Opravná údržba (*corrective maintenance*) – oprava chyb.
- Zdokonalující údržba (*perfective maintenance*) se snaží zlepšit stávající funkcionalitu softwaru v nějakém ohledu (výkonově, v samotné udržitelnosti).

Dle standardu ISO/IEC 25010 [39], jenž definuje modely pro posuzování kvality softwaru, je udržitelnost míra efektivity, s níž lze modifikovat daný produkt nebo systém. Zároveň model kvality produktu zavádí pro udržitelnost několik podkategorií:

- Modularita neboli míra složení systému ze samostatných komponent takových, že změna jedné komponenty má minimální dopad na zbytek.
- Znovupoužitelnost (*reusability*) je schopnost využití komponenty na vícero místech.

- Analyzovatelnost (*analysability*) nám říká, do jaké míry jsme schopni určit dopad plánované změny produktu a také identifikovat části, jež bude nutné upravit.
- Přizpůsobitelnost (*modifiability*) určuje, jak snadné je modifikovat systém bez zanesení nových defektů či degradování kvality existující produktu.
- Testovatelnost (*testability*) posuzuje, s jakou obtížností mohou být pro daný systém stanovena testovací kritéria a jak mohou být tato kritéria ověřena.

Tyto kategorie spolu přirozeně souvisejí a jedna ovlivňuje druhou. V některých případech je třeba učinit určité ústupky a dát přednost jedné vlastnosti před druhou, často však lze dosáhnout rozumných kvalit ve většině z nich. [33]

Kapitola 2

Návrh

V rámci této kapitoly nastíním předpokládané příklady použití vytvořeného SDK. Z těchto odvodím specifikaci jak funkčních, tak nefunkčních požadavků. Na základě těchto nároků pak představím architektonický návrh celého SDK, případně omezení, která vzniknou a je třeba brát je v potaz.

2.1 Případy užití

Všechny následující případy užití vyplývají z reálného používání aplikace Data Stewardship Wizard a přidružených podpůrných nástrojů. Jsou tedy diskutovány s autory a správci DSW. Existence těchto případů je vlastně důvodem, proč vznikl celý nápad vytvořit pro nástroj DSW vlastní SDK.

Jako hlavní případ užití lze uvést smysl jakéhokoli SDK – tj. umožnit nativní práci s nástrojem DSW, objektový přístup k datům a odstínění vývojáře od určitých nízkoúrovňových záležitostí. Tento přístup je pak možné využít na rozdílných místech (v odlišných aplikacích/nástrojích) a není nutné kopírovat logiku, jež je pro přístup k platformě DSW společná.

Konkrétní případy využití tohoto SDK jsou pak následující:

1. Komponenta *Document Worker* momentálně nepracuje s DSW přímo, ale přes message brokera (představme si jako frontu na zprávy) a na základě těchto zpráv poté sama získává data z databáze. Nově by tato komponenta mohla komunikovat přímo s DSW (pomocí SDK) a získávat tak data z API.

2. Nástroj TDK (*Template Development Kit*) umožňuje uživatelům tvořit nové či upravovat stávající šablony. Pro komunikaci s DSW však musí implementovat vlastního API klienta a třídy pro datovou reprezentaci.

3. V rámci projektu DSW je nutné vykazovat počty uživatelů za různá období (aktivita/registrace) a různých druhů (rozlišování např. pomocí

domény emailových adres). Obdobně by bylo dobré moci dělat i statistiky, kolik je kde jakých dokumentů a dotazníků. Hlavní je tedy práce s entitou *User*, a sice, aby šlo lehce dohledat projekty, které danému uživateli patří.

4. SDK by šlo využít pro synchronizaci mailing listu a seznamu uživatelů z různých instancí DSW.

5. Správci DSW instancí by chtěli mít možnost snadno vytvářet skripty pro zjišťování aktuálních verzí šablon/knowledge-modelů, jejich hromadné aktualizace z Data Stewardship Registry, možnost promazávat cache či spravovat nastavení instancí.

6. Workshopům věnovaným nástroji DSW často předchází mnoho manuální práce ze strany organizátorů (např. tvorba testovacích uživatelů) či skrze vlastních API skriptů. Tento proces lze zjednodušit pomocí vytvořeného SDK.

2.2 Specifikace požadavků

V této sekci se nachází kompletní výčet všech požadavků kladených na vytvářené SDK. Požadavky jsou rozděleny do 3 kategorií – funkční (*co* má software dělat), nefunkční (*jak* má onu činnost software dělat) a nice-to-have (vlastnosti, jež nejsou kritické pro hlavní funkcionalitu).

2.2.1 Funkční požadavky

Většina funkčních požadavků přímo vyplývá z případů užití uvedených v předchozí sekci. Zároveň je zde uvedena i funkcionalita, jež byla kladně hodnocena v sekci 1.2.

F1) SDK poskytne nízkoúrovňové rozhraní, jež bude pouze zprostředkovávat HTTP komunikaci s DSW API.

   a) Nebude vytvářet žádné modely pro práci s daty, vstupní i výstupní parametry budou předány v nativních datových strukturách.

   b) Toto rozhraní 1:1 zrcadlí DSW API.

F2) K dispozici bude také vysokoúrovňové rozhraní, které abstrahuje určitou logiku vybraných operací a zajišťuje objektový přístup – jsou vytvořeny modely pro práci s daty. Zahrnuje operace zmíněné v následujících bodech.

F3) Interakce s dotazníky:

a) Získání dotazníků.
b) Získání přidružených dokumentů a šablon.

F4) Práce se šablonami:

a) Získání šablon a přidružených souborů.
b) Úprava existujících šablon.
c) Smazání existujících šablon.
d) Tvorba nových šablon.

F5) Práce s entitou Uživatel:

a) Získání uživatelů.
b) Úprava uživatelů.
c) Smazání uživatelů.
d) Vytvoření uživatelů.
e) Dohledání dotazníků a dokumentů, které patří vybraným uživatelům.

F6) Aktualizace vybraných/všech šablon nebo knowledge modelů.

F7) Správa nastavení instance DSW:

a) Získání konfigurace aplikace.
b) Úprava konfigurace aplikace.

2.2.2 Nefunkční požadavky

V sekci 1.2 věnované rešerši stávajících SDK se můžeme inspirovat při tvorbě požadavků. Z průzkumu vyplynula důležitost poskytnutí vývojářům možnosti jednoduché konfigurace skrze několik různých způsobů. Také podrobná a velmi kvalitní dokumentace je nezbytná. V neposlední řadě se ukázalo, že vystavit jak nízkoúrovňové, tak objektově orientované rozhraní je velkou výhodou a umožňuje pokrýt veškerou funkcionalitu, jež samotné API nabízí.

Sekce 1.3 nám napovídá, jaké aspekty bychom měli primárně zohlednit při vývoji SDK. Použitelnost (míra jednoduchosti, s níž lze nástroj používat) a dodržování (případně i tvorba nových) standardů by měly být brány velmi vážně. Zároveň jsou esenciálními kvalitami, které SDK nemůže postrádat, snadná rozšiřitelnost a udržitelnost z hlediska budoucího vývoje.

N1) Autentizace probíhá poskytnutím přihlašovacích údajů (e-mailová adresa a heslo k účtu na dané instanci DSW) při inicializaci SDK.

N2) Veškerá komunikace s API je ve výchozím stavu šifrovaná; toto chování lze však uživatelem změnit.

N3) Je kladen velký důraz na jednoduchost používání SDK – snadná inicializace a konfigurace; jsou poskytnuty rozumné výchozí hodnoty, jež lze pro konkrétní případy užití snadno přetížit.

N4) Konfigurace celého SDK probíhá 3 způsoby:

   a) Předáním objektu s hodnotami nastavení při inicializaci.
   b) Nastavení hodnot pomocí proměnných prostředí.
   c) Předání cesty ke konfiguračnímu souboru (ve formátu YAML).

N5) Dokumentace:

   a) Všechny veřejné funkce SDK budou zdokumentovány (v kódu, pomocí docstringů).
   b) Z těchto dílčích komentářů bude nástrojem Sphinx vygenerována celková dokumentace.
   c) Dokumentace bude doplněna o krátké příklady použití jednotlivých komponent a funkcí.
   d) Tyto příklady použití budou testované (pomocí nástroje *doctest*).

N6) Rozšiřitelnost:

   a) Do SDK lze snadno přidat novou funkcionalitu. Toto přidání vyžaduje minimální modifikaci stávajícího kódu.
   b) Uživatelé mohou měnit chování klíčových procesů z vnějšku.

N7) Verzování:

   a) Řídí se principy sémantického verzování.
   b) Vždy odráží konkrétní verzi API; díky tomu lze lehce dosáhnout kompatibility.
   c) SDK je vyvíjeno a testováno pro verzi *wizard-serveru 2.13.0*. Tedy první stabilní verze této knihovny by měla být také 2.13.0.

N8) SDK je napsané v jazyce Python; podporované jsou verze 3.7 a vyšší.

N9) Kód splňuje zásady PEP8.

N10) Celý projekt je open-source, vydávaný pod licencí *Apache-2.0 License*.

N11) SDK je distribuováno pomocí oficiálního softwarového repozitáře PyPI.

N12) Testování:

a) Veškeré veřejné rozhraní je pokryto jednotkovými a integračními testy.

b) Funkční požadavky jsou ověřeny funkčními testy.

c) Pokrytí kódu testy je alespoň 80%.

2.2.3 Nice-to-have požadavky

O1) Autentizace je možná i pomocí OAuth protokolu.

O2) Kód je anotovaný (typovaný) dle normy PEP484.

O3) SDK je podporováno na všech hlavních operačních systémech – Linux, MacOS, Windows.

O4) Pokrytí kódu testy je alespoň 90%.

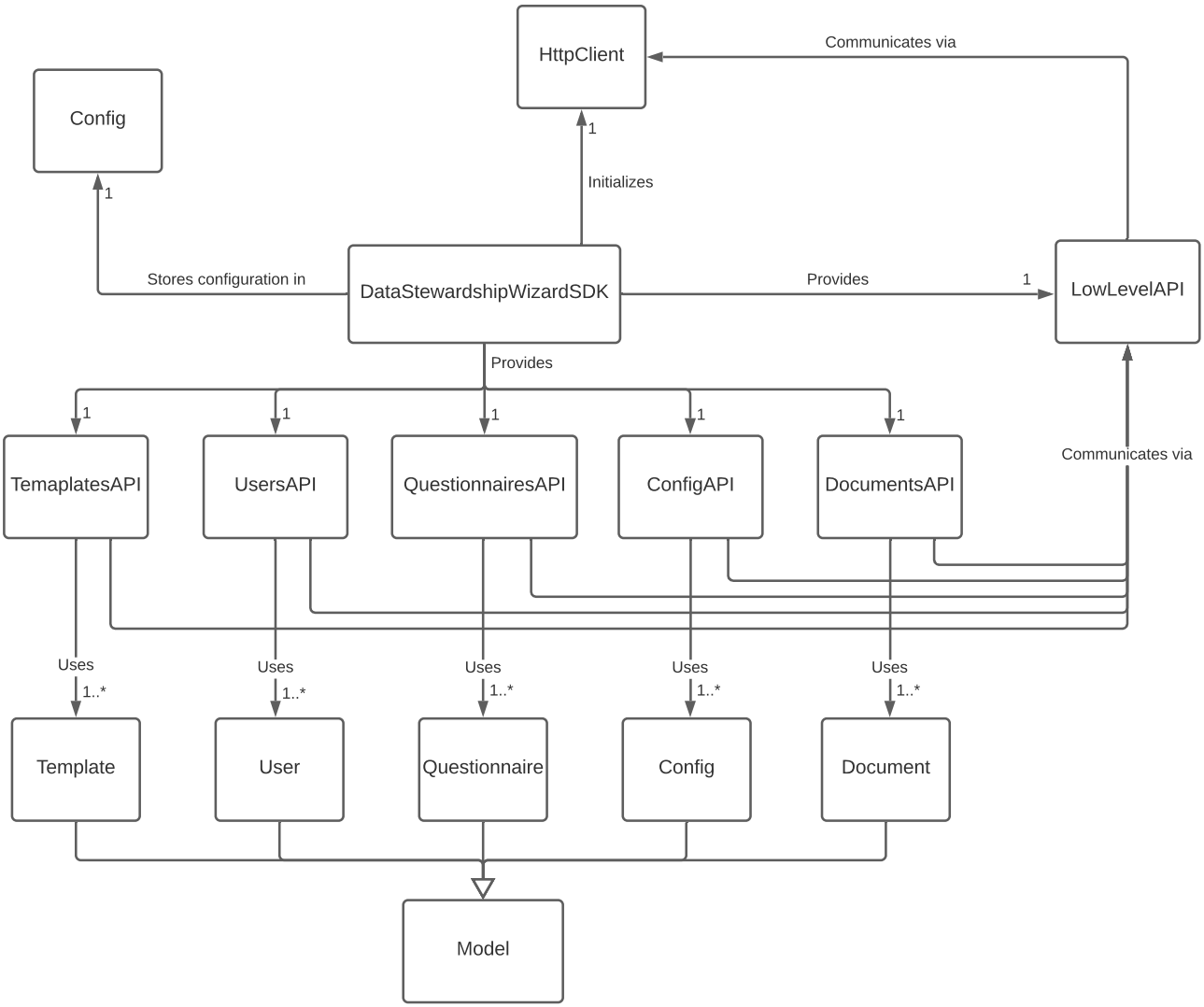
2.3 Architektura systému

Jak bylo již uvedeno, jedná se o knihovnu napsanou v jazyce Python 3.7 a vyšší (požadavek N8) pro práci s nástrojem Data Stewardship Wizard. Je strukturována do několika modulárních komponent, jež můžeme vidět na diagramu 2.1, který popisuje „veřejný“ doménový model celého SDK – tedy objekty, jež jsou zamýšleny pro přímé použití uživateli.

DataStewardshipWizardSDK je hlavní třída, se kterou budou interagovat uživatelé. Zaštiťuje všechny potřebné komponenty, ke kterým uživatelé mohou přistupovat skrze atributy a metody. Inicializace této třídy bude jediným nutným krokem ke správnému fungování celé knihovny (požadavek N3); v minimální konfiguraci je třeba poskytnout pouze URL adresu DSW API a přihlašovací údaje (e-mail a heslo; požadavek N1).

Jedná se o jakéhosi mediátora, který pouze sdružuje veřejně dostupné komponenty a má za úkol provést jejich inicializaci. Pomocí techniky nazývané *dependency injection* (konkrétně *constructor injection*) splníme požadavek na rozšiřitelnost ze strany uživatele (N6b). To znamená, že v některých případech bude možné předat již nakonfigurovaný objekt (jmenovitě logger a HTTP klient) z vnější. Pokud předán nebude, bude inicializován dle předaného nastavení nebo výchozích hodnot.

Config je spíše deklarativní komponentou. Definuje, jaké nastavení je pro celou knihovnu k dispozici, nastavuje rozumné výchozí hodnoty a udržuje ty nastavené uživatelem (jež také validuje). Jeho zodpovědností je také korektní načtení konfigurace všemi možnými cestami, tj. předáním hodnot při



Obrázek 2.1: Analytický doménový model celého SDK

```
1    def put_templates_files(self, template_id, file_uuid, body):
2    """
3    body: {
4        fileName*: string,
5        content*: string
6    }
7    """
```

Obrázek 2.2: Ukázka signatury metody nízkoúrovňového rozhraní

inicializaci třídy `DataStewardshipWizardSDK`, proměnnými prostředí a konfiguračním souborem ve formátu YAML (požadavek N4).

HttpClient zajišťuje komunikaci s DSW API na aplikační vrstvě. Stará se transparentně o proces autentizace (aby ji uživatel nikdy nemusel sám explicitně řešit), transformuje chybové stavy na výjimky a obecně řeší všechny komunikační záležitosti (např. jak dlouho se má držet otevřené spojení, SSL/TLS [požadavek N2], zasílání patřičných HTTP hlaviček apod.).

Důležité je, aby zbytek knihovny operoval jen s obecným rozhraním, nikoli s konkrétní implementací. Tento návrh zvaný *dependency inversion principle* vede k nízké provázanosti modulů a umožňuje nám snadné změny – v případě potřeby není problém vytvořit kompletně novou implementaci (např. nám nemusí vyhovovat implementace dodaná s knihovnou nebo se rozhodneme, že nám již nestačí synchronní zpracování a chceme nabídnout uživatelům i asynchronní způsob) a snadno tak změnit celý proces komunikace (požadavek N6a).

LowLevelAPI zrcadlí 1:1 celé API dané verze DSW. Tato třída tedy obsahuje jednu metodu pro každou kombinaci endpoint + HTTP metoda a odpovídá i vstupními parametry (požadavek F1). Jako příklad vezměme PUT požadavek na endpoint `/templates/{templateId}/files/{fileUuid}`. Ten obsahuje dva *path* parametry a zároveň přijímá data v těle HTTP požadavku. Ve třídě `LowLevelAPI` tedy bude existovat metoda se signaturou z obrázku 2.2. Názvy parametrů jsou záměrně převedeny do *snake case* notace, aby odpovídaly konvencím jazyka a splňovaly zásady PEP8 (požadavek N9). Všimněme si také dokumentačního řetězce, který odhaluje strukturu těla požadavku a typy jednotlivých atributů. To velmi zjednoduše práci, neboť uživatel vůbec nemusí opouštět svůj editor a všechny potřebné informace zjistí z deklarace a dokumentace funkce.

Model je abstraktní předek pro konkrétní implementace reprezentující určitou datovou entitu Data Stewardship Wizardu. Jeho účelem je udržovat data o těchto entitách objektovým způsobem. Bylo by pohodlné, kdybychom mohli

```
1    t = Template(name='template_name')
2    t.metamodel_version = '4'  # <-- Conversion to correct type
3    assert t.metamodel_version == 4
4    t.metamodel_version = False  # <-- Invalid type, raises
```

Obrázek 2.3: Ukázka možného použití datové entity

k datům na objektu přistupovat jako k atributům. Zároveň bychom mohli chtít ihned data *konvertovat* a *validovat* nově přiřazené hodnoty. To vše samozřejmě tak, abychom nemuseli pro každou nově přidanou entitu psát kompletní logiku od znovu a stačil jen nějaký deklarativní výpis všech datových atributů. Možné použití ilustruje ukázka kódu č. 2.3.

Také ví, jak se má konkrétní entita vytvořit, upravit nebo smazat. Zná momentální stav entity, tedy uživatel toto nemusí rozlišovat a k entitám se chová uniformně – ví, že chce danou entitu s určitými daty uložit na server; zavolá metodu `model.save()` a ta za něj transparentně celý proces vyřeší.

Konkrétní datové entity (*Questionnaire*, *Document*, *Template*, ...) přesně odráží definici uvedenou ve Swagger dokumentaci DSW API.

Vysokoúrovňová rozhraní (`TemplatesAPI`, `UsersAPI`, ...) obsahují hlavní funkcionalitu zadanou funkčními požadavky F2-F7. Využívají modely (inicializují a vracejí je uživateli jako výsledek svých operací) a jak lze vidět z diagramu 2.1, s API komunikují pomocí nízkoúrovňového rozhraní. Jejich úkolem je správně agregovat data v modelech a poskytovat jednoduché rozhraní pro vybrané operace (nejčastěji CRUD).

Kapitola 3

Realizace

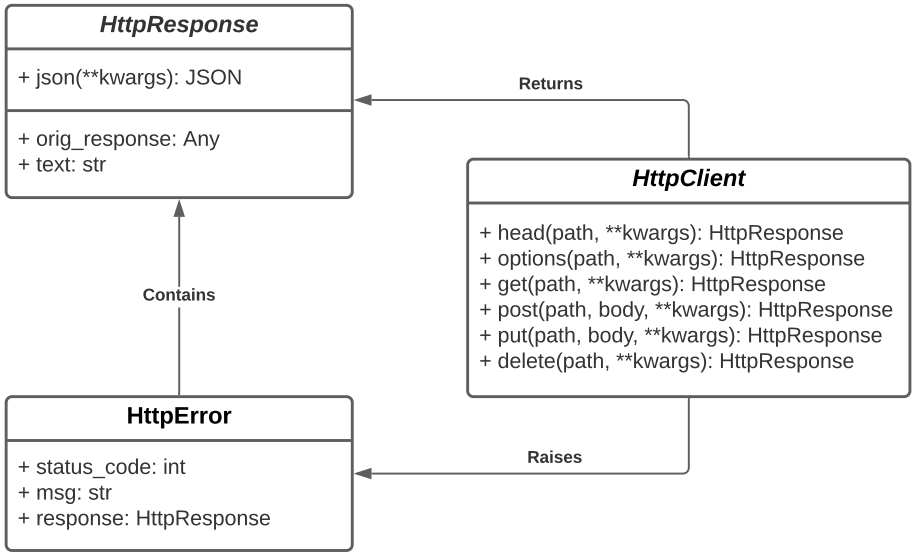
Cílem této kapitoly je představit implementaci SDK realizovanou dle návrhu z předchozí kapitoly. Zaměřím se na způsoby řešení hlavních problémů, představím pasáže kódu, které jsou něčím zajímavé a především také ukážu, jak se samotnou knihovnou pracovat. Lze tak tuto část chápat jako jakousi uživatelskou příručku.

3.1 Použité technologie

- Python 3.7
- Requests (HTTP knihovna)
- PyJWT (zpracování autentizačních JWT tokenů)
- PyYAML (parsování YAML konfiguračních souborů)
- Jinja2 (šablona pro nízkoúrovňové rozhraní)
- Sphinx (generování dokumentace)
- PyTest, Tox, Coverage, Betamax (testovací frameworky)
- Flake8, MyPy, PyLint (statická analýza, kontrola typů, formátování a kvality kódu obecně)

3.2 HTTP klient

Jak bylo uvedeno již v návrhu, celé SDK pracuje pouze s abstrakcí HTTP klienta, nikoli s konkrétní implementací. Tuto abstrakci popisuje diagram tříd na obrázku č. 3.1. Původně byla abstraktní třída i `HttpError`, ale posléze se ukázalo, že její obecná implementace naprosto postačuje i konkrétnímu použití.



Obrázek 3.1: Třídy rozhraní HTTP klienta

V každé odpovědi (třída `HttpResponse`) je obsažena i původní odpověď ze serveru, ať už je reprezentována libovolně. To nabízí uživatelům naprostou volnost a v případě potřeby mohou operovat přímo s tímto objektem.

Všimněme si, že každá metoda přijímá libovolné klíčové argumenty. Tímto způsobem lze každý jeden požadavek individuálně nastavit a předat tak cokoli, co samotná implementace HTTP metody dokáže zpracovat.

Spolu s výsledným SDK je dodána implementace za pomoci kanonické Pythonní knihovny *Requests*. Provádí klasické synchronní požadavky, využívá *Session* objekt k recyklaci HTTP konexí (a tím pádem zrychlení jednotlivých požadavků, neboť odpadá nutnost navazovat pokaždé nové spojení) a zajišťuje transparentní proces autentizace. Zároveň je tento HTTP klient plně konfigurovatelný a poskytuje i mechanismy pro jeho snadné rozšíření – tzv. `before_request` a `after_request` hooks. Tedy lze jakkoli upravit požadavek i odpověď ze serveru, případně provést libovolnou akci definovanou uživatelem.

3.3 Nízkoúrovňové rozhraní

Z výsledků rešerše v části 1.2 víme, že existuje více různých návrhů, jak řešit toto nízkoúrovňové rozhraní. Lze celé dynamicky vytvořit za běhu programu (tedy pouze v paměti), avšak to má dle mého názoru velkou nevýhodu – jako programátor využívající takové rozhraní se nemohu při psaní kódu podívat na dané funkce a přečíst si jejich dokumentaci, aniž bych program spustil.

Ideální přístup je tedy mít klasický zdrojový soubor s kódem, ten však neudržovat ručně, nýbrž generovat ze Swagger dokumentace (kde nalezneme veškeré potřebné údaje) pro každou novou verzi DSW API. Tím pádem lehce zaručíme 100% konzistenci s API a zároveň dokážeme velmi rychle reagovat na případné změny.

Prvním krokem při implementaci bylo získání tzv. *proof of concept* – důkaz, že navržený způsob řešení bude technicky proveditelný a že lze opravdu celé nízkoúrovňové rozhraní automaticky generovat.

To se povedlo; vznikl skript, jenž je součástí zdrojových kódu. Stačí předat URL adresu Swagger API dokumentace a vygeneruje celé rozhraní, které poté uloží do souboru. Kostra tohoto souboru je popsána pomocí Jinja šablony, kterou skript plní daty.

V podstatě jedinou akcí v kódu každé metody tohoto rozhraní, kromě volání samotného endpointu, je převedení všech poskytnutých argumentů do `camelCase` notace (neboť tu akceptuje server). Je tedy možné předávat query parametry i tělo requestu ve `snake_case` notaci, jak je tomu v Pythonu zvykem, a dodržovat zavedené konvence.

3.4 Atributy datových entit

Jedním z hlavních problémů, jenž implementace musela řešit, neboť návrh nespecifikuje jeho konkrétní provedení, bylo, jak elegantně definovat atributy datových entit (modelů), jak uchovávat přiřazená data a jak je validovat. Z návrhu naopak vyplývá, jak by mělo vypadat použití těchto entit:

- Všechna data lze předat přímo při inicializaci a vytvořit tak entitu v jediném kroku.
- Stejně tak lze ale všechna data nastavit přes jednotlivé atributy. Tedy nejdříve vytvořit kompletně prázdnou entitu a poté ji inicializovat postupně (to se může hodit například v situaci, kdy jednotlivé hodnoty získáváme v programu postupně a nemáme je k dispozici najednou).
- Při přiřazení dat (oběma způsoby uvedenými výše) rovnou nastává validace. To je výhodné, pokud používáme entitu např. v interaktivním prostředí (*REPL* či debugger). Zároveň uživatelům umožňujeme velmi jemnou kontrolu nad validací (lze tak třeba validaci některých atributů ignorovat).
- Při přiřazení zároveň nastává automatická konverze na správný datový typ. Tedy pokud je atribut typu „objekt“ a my poskytneme jako hodnotu slovník s odpovídajícími klíči, měl by se do příslušného atributu uložit objekt zinicializovaný předanými daty.

```
1    @dataclass
2    class Template:
3        id: str
4        files: List[TemplateFile] = None
```

Obrázek 3.2: Definice datové entity pomocí `dataclass` dekorátoru

K těmto nárokům by se pak měly přidat následující:

- Pokud atribut na datové entitě není nastaven, automaticky vrací `None`.
- Při definici atributu lze nastavit jeho výchozí hodnotu, zdali je pouze ke čtení, *immutable* (lze nastavit pouze jednou) a obor hodnot, kterých může nabývat.
- Musí existovat způsob, jak inicializovat entitu bez validací (může být definovaná špatně, přesto však půjde uživateli použít než dorazí oprava knihovny).

3.4.1 Dynamické atributy

Stejně jako při návrhu nízkoúrovňového API, i zde byla možnost získávat atributy dynamicky. Tzn. definovat skrytě uvnitř třídy, jak jednotlivá data vypadají, jak jsou strukturována, jak mají být validována apod. Pro uživatele by pak atributy byly přístupné pomocí magických metod, konkrétně `__getattr__` a `__getattribute__`. To má samozřejmě i totožné nevýhody – v době psaní kódu nevíme, s čím konkrétně pracujeme, jaké atributy se na entitě nachází či jakého jsou typu.

3.4.2 Dataclass dekorátor

Použití built-in dekorátoru `@dataclass` se jevilo jako další možnost. Tímto způsobem lze velmi minimalisticky definovat jednotlivé atributy, jejich datové typy, případně výchozí hodnoty. Automaticky jsou pak za nás vytvořeny metody `__init__`, `__eq__` a `__str__`. Ukázku takové třídy můžeme vidět na úryvku 3.2.

Nevýhody tohoto přístupu se však začaly ihned projevovat:

- Nelze jednoduše nastavit, jaké hodnoty jsou pouze pro čtení či neměnné, stejně tak obor hodnot.
- Neprovádí se validace ani konverze.
- Nelze inicializovat postupně; pokud není poskytnuta výchozí hodnota, musí být atribut předán přímo při tvorbě objektu.

```
1    class Descriptor:
2        def __set__(self, obj, value):
3            ...
4
5    class Entity:
6        attr = Descriptor()
7
8    obj = Entity()
9    obj.attr = 123    # <-- `Descriptor.__set__` method invoked
```

Obrázek 3.3: Ukázka jednoduchého deskriptoru

Zmíněné nedostatky lze pravděpodobně vyřešit vlastní implementací, jež bude vycházet z `dataclass` modulu za použití vhodné dědičnosti. avšak přišlo mi, že se jedná o ohýbání něčeho, co nebylo určeno k potřebám této knihovny.

3.4.3 Property dekorátor

Třetí možností byla implementace jednotlivých atributů pomocí dekorátoru `@property`. Umožňuje definovat metodu, která se zvenčí chová jako atribut, tedy pro uživatele schovává implementaci, o níž nemusí vědět. Šlo by takto lehce definovat návratové hodnoty, validovat každý atribut a zároveň nastavovat požadované vlastnosti. Deklarace datové entity by však byla poměrně rozsáhlá a bylo by zapotřebí hodně psaní.

3.4.4 Deskriptory

Pythonní dokumentace definuje deskriptor jako libovolnou třídu, jež implementuje alespoň jednu z metod `__get__`, `__set__` nebo `__delete__`. Tato třída lze pak použít k definici atributů jiné třídy, přičemž přístup či přiřazení přes klasickou tečkovou notaci (např. `obj.attr = 123`) využívají výše zmíněné metody na třídě deskriptoru. Pro lepší představu vizte ukázku č. 3.3.

Tento způsob definice atributů se zdál nejlepším řešením, neboť logika atributů zůstane oddělena od samotné deklarace datových entit, zároveň zápis těchto entit bude velmi kompaktní a přehledný. Splníme všechny požadavky sepsané na začátku této sekce a pro uživatele bude jednoduché nahlédnout, jaké atributy daná entita obsahuje a jakého jsou typu.

3.4.5 Finální řešení

Každý atribut je definován pomocí třídy `Attribute` (jak bylo zmíněno výše, jedná se o deskriptor). Tato třída má pouze jediný povinný parametr, a sice typ, jehož atribut může nabývat. Typem není myšlen standardní typ Pythonu (`str` či `int`), nýbrž třída s rozhraním znázorněným na obrázku č. 3.4. Kvůli separaci zodpovědností má typ vlastní třídu – logika validace typu a jeho

```
class Type:
  def validate(self, value):
  """
  Verify that the `value` is of correct data type.
  """

  def convert(self, value):
  """
  Try to convert the `value` to a correct data type.
  """

  def value_repr(self, value):
  """
  Return string representation of the given
  value for this particular type.
  """
```

Obrázek 3.4: Rozhraní třídy `Type`

konverze se tak může vyvíjet naprosto nezávisle. Velmi zjednodušenou implementaci (kvůli stručnosti byly odebrány některé metody) spolupráce mezi atributem a jeho typem můžeme vidět na ukázce č. 3.5.

Jak vidíme z kódu 3.5, každý deskriptor ukládá hodnotu svého atributu na instanci entity, na níž je definován. Tedy všechny datové entity musí mít „privátní“ slovník `_attributes`, kam se ukládají samotné hodnoty atributu (a odkud jsou také získávány při čtení). To je zajištěno pomocí prosté dědičnosti, kdy všechny datové entity dědí ze třídy `AttributesMixin`, která poskytuje veškeré potřebné mechanismy pro práci s atributy.

Úkolem třídy `Attribute` je kontrola následujících vlastností při přiřazení hodnoty (tento proces je znázorněn také na diagramu 3.6):

- Pokud je při inicializaci deskriptoru nastaven příznak `read_only`, přiřazení do atributu vyhazuje výjimku.
- Pokud je při inicializaci deskriptoru nastaven příznak `immutable`, přiřazení hodnoty se provede pouze jednou. Všechny ostatní pokusy končí výjimkou.
- Konverze a validace správného datového typu. Tuto činnost deleguje na svůj typ.
- Pokud je při inicializaci deskriptoru nastaven obor hodnot (parametr `choices`), zkontroluje, zda hodnota spadá do této množiny.

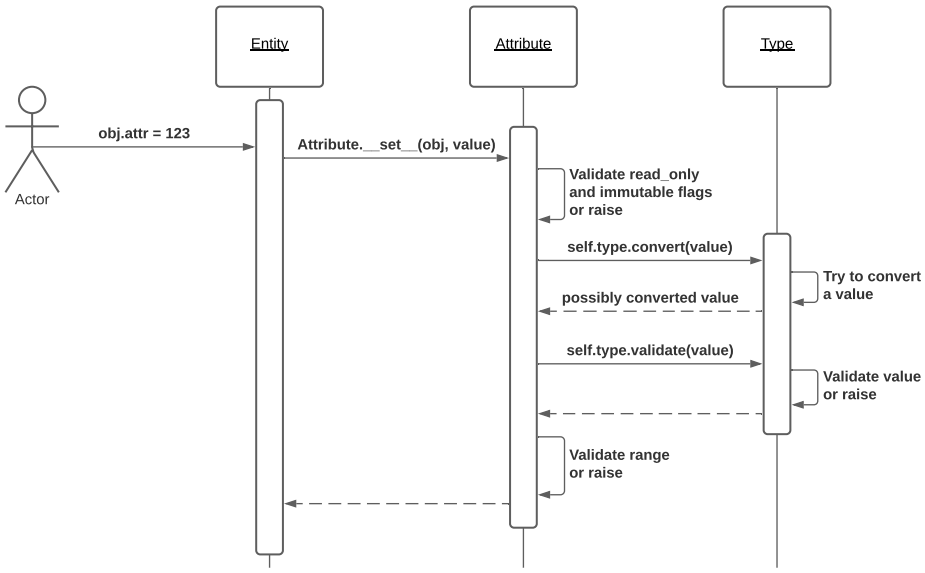
```
1   class IntegerType(Type):
2     def convert(self, value):
3       return int(value)
4
5     def validate(self, value):
6       if not isinstance(value, int):
7           raise ValueError
8
9   class Attribute:
10    def __set__(self, obj, value):
11      if self.read_only:
12          raise AttributeError
13      if self.immutable and self.name in obj._attributes:
14          raise AttributeError
15
16      value = self.type.convert(value)
17      self.type.validate(value)
18
19      if value not in self.choices:
20          raise ValueError
21
22      obj._attributes[self.name] = value
23
24  class Entity:
25    attr = Attribute(IntegerType(), choices=(1, 2 ,3))
26
27  entity = Entity()
28  entity.attr = '2'
29  entity.attr = 42  # <-- raises, not in range (1, 2, 3)
```

Obrázek 3.5: Implementace atributu a typu; jejich použití na entitě

Stejně, jako lze nastavit obor hodnot při definici atributu, lze také specifikovat výchozí hodnotu, pokud atribut nebude explicitně nastaven. Poté stačí implementovat na třídě `Attribute` metodu `__get__`, která buď z instance entity navrátí hodnotu požadovaného atribut, případně vrátí výchozí hodnotu nebo `None`.

3.4.6 Textová reprezentace

Každý potomek třídy `Type` navíc implementuje metodu `value_repr`. Tato metoda vrací textovou reprezentaci dané hodnoty pro konkrétní typ. To nám dává velmi jemnou kontrolu nad tím, jak potomky třídy `AttributesMixin`, a tím pádem datové entity, můžeme textově reprezentovat, což se opět velice hodí například v interaktivních prostředích. Taková textová reprezentace pak může vypadat následovně (jedná se o opravdovou reprezentaci šablony, jak ji zobrazí metoda `str()`):



Obrázek 3.6: Sekvenční diagram procesu přiřazení hodnoty do atributu

```
<Template
  allowed_packages=[...] (1 items)
  assets=[]
  created_at=2021-03-12 21:12:15
  description="Exported questions and
               answers from a questionnaire"
  files=[...] (2 items)
  formats=[...] (7 items)
  license="Apache-2.0"
  metamodel_version=3
  name="Questionnaire Report"
  organization=<OrganizationSimple ...>
  organization_id="dsw"
  readme="# Questionnaire Report..." (truncated)
  remote_latest_version="1.3.0"
  state="UpToDateTemplateState"
  template_id="questionnaire-report"
  usable_packages=[...] (1 items)
  uuid="dsw:questionnaire-report:1.3.0"
  version="1.3.0"
  versions=['1.0.0', '1.3.0']
>
```

3.5 Model

Jak již bylo zmíněno, každá datová entita dědí ze třídy `AttributesMixin`, která zajišťuje práci se samotnými atributy (jejich deskriptory). Avšak mezi touto třídou a konkrétní entitou (jako je například `Document` či `Template`) se nachází ještě jedna mezivrstva, a sice třída `Model`.

Ta definuje především metody `save` a `delete`. Jejich provedení však deleguje na třídu `State`, jež je součástí každého modelu. Jak název napovídá, jedná se o velmi jednoduchý stavový automat, implementovaný ve stylu návrhového vzoru State. Rozlišuje 3 stavy a dle nich také logiku metod `save` a `delete`:

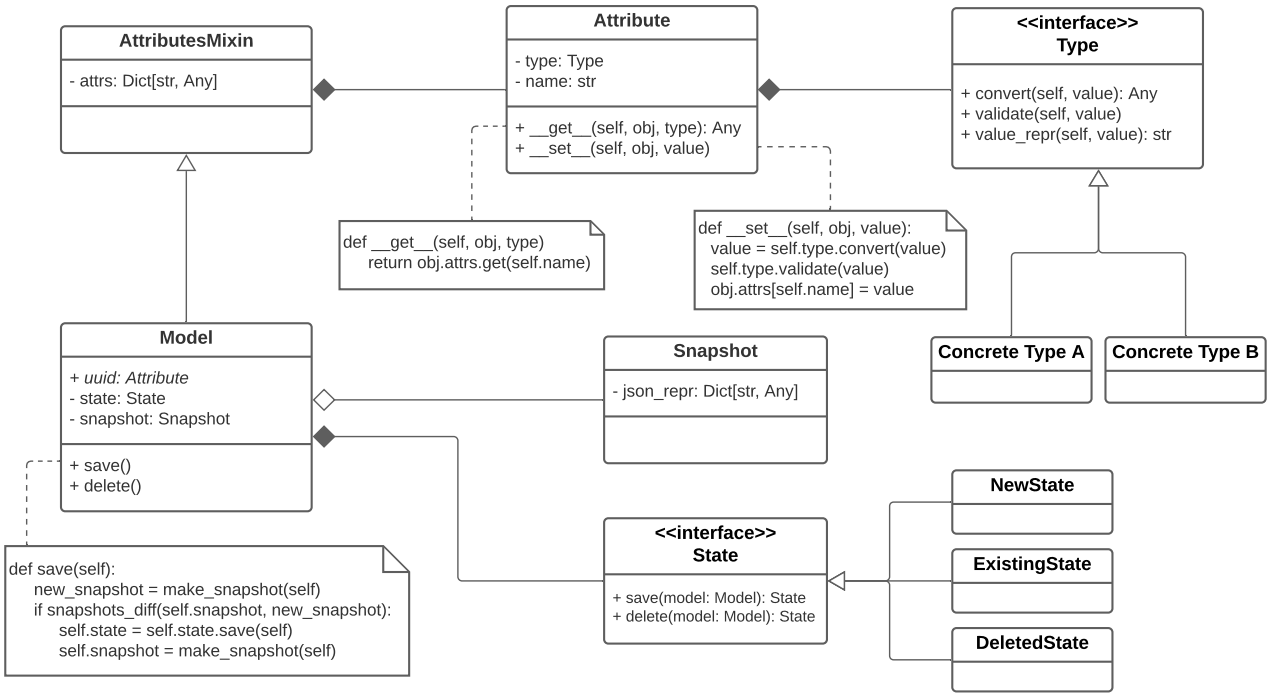
1. Nová (ještě neexistující) entita
   - `save` vytvoří entitu na serveru (zašle POST HTTP požadavek) a uvede entitu do stavu „existující“.
   - `delete` pouze změní stav na „smazáno“.
2. Existující entita
   - `save` upraví entitu na serveru (zašle PUT HTTP požadavek); stav entity se nemění.
   - `delete` smaže entitu na serveru (zašle DELETE HTTP požadavek) a změní stav na „smazáno“.
3. Smazaná entita
   - `save` vyhazuje výjimku (entitu už byla smazána a nelze znovu uložit).
   - `delete` nic nedělá.

Každá konkrétní entita tedy kromě svých atributů definuje, jak vypadá její vytvoření, úprava nebo smazání. Zároveň je zaveden mechanismus tzv. snapshotů pro detekci změn. Při každém volání metody `save` je vytvořen obraz momentálního stavu modelu a porovnán s původním stavem. Díky tomu model ví, kdy má smysl provolávat DSW API (můžeme tedy ušetřit zbytečná volání) a v některých případech se na základě změn rozhoduje, jaké konkrétní endpointy volat (např. uživatel má odlišné endpointy pro úpravu a pro změnu hesla).

Názorné schéma spolupráce všech tříd zapojených do celého životního cyklu modelu (a tím pádem datové entity) můžeme vidět na diagramu č. 3.7.

3.6 Konfigurace

Třída deklarující hodnoty, jež lze uživatelem nastavit, shodou okolností může s jemnými úpravami využít totožný koncept atributů jako datové entity. Zavedení takové třídy se poté omezí na pouhou definici jednotlivých atributů ke



Obrázek 3.7: Diagram tříd interních stavebních prvků každé datové entity

konfiguraci. Validace je již zajištěna každým jedním atributem. Jediné, co je třeba upravit, je chování v případě, že není nalezen atribut, který je definován jako povinný a zároveň nemá nastavenou žádnou výchozí hodnotu. V takové situaci musíme vyhodit výjimku a uživatele požádat o poskytnutí hodnoty pro daný atribut.

Nastavení pomocí souboru probíhá dle návrhu – při inicializaci SDK je možné předat cestu k souboru ve formátu YAML. Zde se očekává mapa s klíčem *dsw_sdk*. Nastavení pomocí proměnných prostředí načítá všechny proměnné začínající prefixem *DSW_SDK*.

3.7 Ukázka použití

Úryvek kódu č. 3.8 názorně představuje, jak se s celým SDK pracuje. Na řádcích 1-6 vidíme počáteční inicializaci celého SDK. Nastavení URL API a e-mailu probíhá klasicky přes argumenty konstruktoru, heslo nastavujeme pomocí proměnné prostředí.

Poté přes `TemplatesAPI` (atribut `dsw_sdk.templates`) načteme všechny šablony a ke každé šabloně přidáme jeden nový soubor.

```
1  dsw_sdk = DataStewardshipWizardSDK(
2    api_url='http://localhost:3030',
3    email='albert.einstein@example.com',
4    # Password is set via DSW_SDK_PASSWORD environment
5    # variable in order not to compromise our secrets
6  )
7
8  # If we don't specify any parameters, we get all templates
9  templates = dsw_sdk.templates.get_templates()
10
11 # For each template, add a new template  file
12 for temp in templates:
13   new_file = TemplateFile(dsw_sdk, content='...',
14                           file_name='new.txt')
15   temp.files.append(new_file)
16   temp.save()
17
18 # Create new user instance directly via model object
19 user = User(
20   dsw_sdk,
21   email='new.user@example.com',
22   first_name='John',
23   last_name='Doe',
24 )
25 user.password = 'password'
26 # User gets created on the server
27 user.save()
28
29 # User instance is updated with
30 # the data from the server
31 print(user.groups)
32
33 user.role = 'admin'
34 # Changes get propagated to the server
35 user.save()
36
37 # Finally we can delete the instance
38 user.delete()
```

Obrázek 3.8: Ukázka použití SDK

Na řádcích 19-24 vytváříme nový model datové entity `User`. Uživatele poté vytvoříme i na serveru pomocí volání `user.save()`. Ověříme, že se do modelu doplnily informace z backendu, provedeme nějaké změny a entitu znovu uložíme. Ukázku uzavírá metoda `delete`, která danou entitu smaže.

Kapitola 4

Testování a distribuce

V této kapitole seznámím čtenáře s testovacími postupy, jež jsem zvolil. Rovněž vylíčím proces distribuce knihovny na oficiální softwarový repozitář pro jazyk Python – PyPI. Na konci také shrnu, jak se povedlo naplnit všechny zadané požadavky a jaké jsou další kroky v rozvoji tohoto SDK.

4.1 Testování

Testy jsou rozdělené dle nefunkčních požadavků (N12a, N12b) na jednotkové, integrační a funkční.

4.1.1 Jednotkové testy

Ověřují vždy jeden dílčí celek. V tomto případě základní stavební prvky, s jejichž pomocí jsou poté vytvářeny komplexní komponenty. Jedná se především o testování datových typů (podtřídy třídy `Type`), atributů (třídy `Attribute` a `AttrbiutesMixin`) a modelu (třída `Model`). Nicméně jednotkovými testy je pokryt i HTTP klient či třídy pro autentizaci.

V tomto druhu testů jsou hojně uplatněny tzv. *mocky* – objekty, které imitují skutečnou implementaci. Díky tomu lze snadno oddělit jednotlivé komponenty od sebe a testovat v danou chvíli opravdu jen jednu konkrétní třídu nebo funkci. Zároveň lze na těchto objektech snadno ověřit, zda byly skutečně použity, s jakými argumenty byly zavolány apod. Příklad jednoho mocku můžeme vidět na ukázce č. 4.1.

4.1.2 Integační testy

Narozdíl od jednotkových testů již ověřuji fungování většího celku, zpravidla několika komponent. Integračními testy je pokryta třída `Config` a datové entity a práce s nimi. Pro tyto třídy nemá již moc smysl *mockování*, neboť při sebemenší úpravě kódu bychom museli měnit velkou část testů.

```
1  def test_bearer_auth():
2    mocked_request = mock.Mock(headers={})
3    auth = BearerAuth('some random token')
4    assert auth(mocked_request).headers == {
5      'Authorization': 'Bearer some random token',
6    }
```

Obrázek 4.1: Ukázka testu a techniky *mockování*

V rámci integračních testů jsem také použil knihovnu *Betamax*. Ta umožňuje nahrávat HTTP požadavky a odpovědi do tzv. kazet. Když poté přijde na zpracování konkrétního požadavku, klient se nejdříve podívá do těchto kazet, zda již stejná komunikace nebyla někdy v minulosti zaznamenána. Pokud ano, k samotnému HTTP spojení vůbec nedojde a klient vrátí data z kazety. To má několik výhod:

- Testy nepotřebují pro svůj běh reálnou instanci DSW API. Stačí komunikaci jednou nahrát do kazet a pak již mohou fungovat kompletně v offline módu. To samozřejmě velmi zvyšuje i rychlost jejich provedení.
- Nemusíme odpověď ze serveru tvořit sami. Šlo by to opět vyřešit pomocí *mocků*, avšak to je velmi náchylné na změny API. Kazety lze jednoduše při změně API rozhraní přenahrát a není třeba udělat jedinou změnu v kódu.

4.1.3 Funkční testy

Poslední vrstvou testovací struktury jsou funkční testy. Jak již název napovídá, testují funkční požadavky kladené na knihovnu. Tj. ověřují především jednotlivá vysokoúrovňová rozhraní a práci s datovými entitami. Hlavním rozdílem oproti integračním testům je black-box přístup k testování. Zatímco integrační testy ověřují, zda procesy proběhly, jak měly (např. zda se provolaly správné metody), funkční testy kontrolují pouze výsledek. Simulují hlavní scénáře užití a v podstatě zkouší běžné zacházení s knihovnou.

4.1.4 Spouštění testů

Veškeré testy jsou pouštěny oproti instalaci SDK, nikoli samotným zdrojovým souborům. Tím je myšleno, že knihovna je nejprve ubalena do softwarového balíku připraveného k instalaci. Poté je vytvořeno nové virtuální prostředí, kam se tento balík nainstaluje a uvnitř tohoto prostředí se teprve pouští testy, oproti této konkrétní instalaci.

To je velmi důležité, neboť jenom tak mohou testy simulovat reálně použití. Uživatelé také nebudou mít stejné prostředí jako máme my při vývoji. Tímto

Obrázek 4.2: Průběh testů na repozitáři GitHub (CI)

způsobem spouštění testů tak dokážeme eliminovat některé zákeřné chyby, které by se objevily až později při užívání uživateli.

Celý postup pak velmi zjednodušuje nástroj *Tox*, jenž se o kompletní proces postará a jehož spuštění je otázka jednoho příkazu.

4.1.5 Kontinuální integrace (CI)

Projekt je verzován pomocí Gitu a lze jej nalézt v GitHub repozitáři na adrese `https://github.com/ds-wizard/dsw-sdk`. V rámci GitHubu jsou k dispozici tzv. *GitHub Actions*, jež umožňují spustit libovolné workflow při určité události.

S jejich pomocí je nastavena kontinuální integrace tak, aby při každém nahrání nových změn či vytvoření pull requestu došlo ke spuštění všech testů. Testy jsou však pouštěny v tzv. testovací matici, kde jsou na jedné dimenzi definovány všechny knihovnou podporované verze jazyka Python (momentálně 3.7, 3.8 a 3.9) a na druhé dimenzi všechny hlavní operační systémy (Linux, MacOS, Windows). To znamená, že se testy pouštějí celkem 9x, pro každou kombinaci operačního systému a verze Pythonu zvlášť.

Tím je kontrolována kompatibilita napříč systémy i verzemi jazyka (požadavek O3).

4.2 Softwarové metriky

Pro zachování určité kvality kódu, která jistě ovlivňuje i jeho udržitelnost, jsou v projektu měřeny následující metriky:

- pokrytí kódu testy,
- statická analýza datových typů (díky anotaci kódu dle normy PEP484),
- cyklomatická složitost,
- duplicity v kódu,
- počet řádků funkcí, metod i modulů,
- vhodná pojmenování,
- dodržování konvencí jazyka,

a další – více lze nalézt v dokumentaci nástrojů `Coverage`, `Flake8`, `PyLint` a `MyPy`, které se o sběr a kontrolu metrik starají.

Tyto metriky jsou také součástí kontinuální integrace popsané v sekci výše. Tedy při každé změně zdrojových souborů (či při vytvoření pull requestu) je spuštěn sběr všech metrik a jejich vyhodnocení.

4.3 Distribuce

Pro distribuci knihovny na softwarový repozitář PyPI jsem zvolil standardní postup, to jest kombinace nástrojů `setuptools` a `wheels` (tvorba samotného balíku). Nahrání na PyPI pak zajišťuje nástroj `twine`. Instalace SDK je možná s pomocí následujícího příkazu:

```
pip install dsw-sdk
```

K dispozici jsou 2 druhy balíků – zdrojová distribuce (tzv. *tarball*) a *build distribuce* (ve formátu *wheel*). V obou případech se v zásadě jedná o obyčejný archiv obsahující zdrojové kódy. Wheel je však upřednostňován i samotným PyPI, neboť stačí pouze stáhnout, rozbalit do správné cesty a instalace je hotova. V případě zdrojové distribuce je nutné provést ještě sestavení, což může vyžadovat určité závislosti navíc na straně uživatele. Markantní rozdíl pak nastává, když knihovna neobsahuje čistě Pythonní kód, ale navíc třeba i *C extension*.

Distribuce SDK bude v budoucnu také automatizována pomocí GitHub Actions. Jedná se tedy vlastně o jakousi formu *Continuous Delivery* – při každém nahrání štítku do GitHub repozitáře dojde k vytvoření balíku a jeho nahrání na PyPI pod verzí odpovídající právě nahranému štítku.

4.4 Naplnění zadaných požadavků

Funkční požadavky byly splněny všechny – bylo vytvořeno jak nízkoúrovňové rozhraní zprostředkovávající komunikaci s DSW API (vystavující všechny dostupné metody), tak objektové rozhraní pro práci s požadovanými datovými entitami DSW.

Z nefunkčních je pouze částečně splněn požadavek na možnost měnit chování klíčových procesů z vnější. To je do velké míry možné díky *dependency injection* a *hooks* (které poskytuje výchozí HTTP klient). Avšak to se týká pouze HTTP komunikaci a logování, nelze například ovlivnit způsob vytváření datových entit uvnitř vysokoúrovňového API. Nicméně dle mého názoru se jedná o tak okrajovou funkcionalitu, že prozatím zůstává neimplementována.

Jediným požadavkem, který nebyl splněn, je autentizace pomocí OAuth protokolu, avšak od začátku se jednalo o *nice-to-have* funkcionalitu. Naopak všechny ostatní požadavky z této kategorie byly splněny, ačkoli se jedná spíše o doplňkové vlastnosti.

Tabulka 4.1 podrobně zhodnocuje každý jeden požadavek ze sekce 2.1. Zelená barva značí splnění, žlutá barva pouze částečné splnění a červená barva nesplnění požadavku.

4.5 Možná vylepšení SDK

Prvním krokem v následujícím vývoji by dle mého názoru mělo být vytvoření atributu, který by se načítal až v okamžiku, kdy je požadován (tzv. *lazy-loaded* či *on-demand* atribut). Taková vlastnost by byla velice výhodná u entit, jež mají vazbu na jinou entitu – např. při získání uživatele se musí načíst všechny dotazníky a při načítání každého dotazníku se ze serveru získávají ještě dokumenty daného dotazníku. Ne vždy ale potřebujeme uživatelovy dotazníky či dokumenty, tedy by se mohly načítat až ve chvíli, kdy k nim v kódu přistoupíme.

Drobnou nedokonalostí, jež si určitě zaslouží pozornost, trpí atributy reprezentující kolekce. Můžeme kupříkladu definovat atribut, jenž má být typu „list celých čísel“. A skutečně, pokud do takového atributu zkusíme přiřadit něco jiného než seznam celých čísel, nastane chyba. Kolekce lze ale modifikovat tzv. *in-place*, tedy k přiřazení nedojde. Tím pádem nedojde ani k provedení metody `Attribute.__set__` zajišťující validaci atributů. Chyba nastane až ve chvíli, kdy se pokusíme datovou entitu uložit, to je však nekonzistentní s klasickým chováním atributů.

V budoucnu by také mohly přijít požadavky na zprostředkování další funkcionality, kterou DSW API nabízí – může jít například o tvorbu dokumentů či knowledge-modelů. Implementace takové vlastnosti by však měla být díky návrhu poměrně přímočará a rychlá.

Tabulka 4.1: Přehled naplnění jednotlivých požadavků

Kód	Požadavek
F1	Nízkoúrovňové rozhraní zrcadlící DSW API
F2	Vysokoúrovňové API zajišťující objektový přístup
F3	Interakce s dotazníky
F4	Práce s šablonami
F5	Práce s uživateli
F6	Aktualizace šablon a knowledge modelů
F7	Správa nastavení instance DSW
N1	Autentizace pomocí e-mailové adresy a hesla
N2	(Nastavitelná) Šifrovaná komunikace s DSW API
N3	Snadná inicializace a konfigurace s výchozími hodnotami
N4	3 způsoby předání konfiguračních hodnot
N5a	Dokumentace veřejného rozhraní
N5b	Dokumentace celého projektu (Sphinx)
N5c	Krátké příklady použití jako součást dokumentace v kódu
N5d	Testování těchto příkladů pomocí nástroje `doctest`
N6a	Rozšiřitelnost – snadné přidání nové funkcionality
N6b	Možnost měnit chování klíčových procesů z vnější
N7	Sémantické verzování odrážející vždy danou verzi API
N8	SDK napsané v jazyce Python; podporuje verze 3.7 a vyšší
N9	Kód splňující zásady PEP8
N10	Zdrojové soubory vydány pod *Apache-2.0 License*
N11	Distribuce skrze softwarový repozitář PyPI
N12a	Pokrytí veřejného rozhraní jednotkovými a integračními testy
N12b	Funkční požadavky ověřené funkčními testy
N12c	Pokrytí kódu alespoň 80%
O1	OAuth autentizace
O2	Anotovaný (typovaný) kód dle normy PEP484
O3	Podpora operačních systémů Linux, MacOS, Windows
O4	Pokrytí kódu alespoň 90%

Vylepšení si zaslouží také skript pro generování kódu nízkoúrovňového rozhraní. Ten byl napsán v rychlosti na úplném začátku implementace jako *proof of concept*, nicméně v této podobně již zůstal. Je funkční, avšak rozhodně nepodléhá stejné kvalitě jako zbytek kódu. Také generování dokumentačních řetězců pro jednotlivé metody bohužel nebylo dotaženo k dokonalosti, tedy momentálně chybí popis struktury těla HTTP požadavku, jež je očekáváno.

Závěr

V diplomové práci jsem se zabýval analýzou nástroje Data Stewardship Wizard, který umožňuje výzkumníkům z nejrůznějších oborů sestavit tzv. *data management plán* – plán, jak zacházet s daty a jak je korektně spravovat. Tento software vyvíjený na naší fakultě si pak klade za cíl především vzdělávat výzkumníky v oblasti data managementu a odstínit je od zbytečné administrativy.

Po představení klíčové funkcionality, architektury DSW a skutečností důležitých pro vývoj budoucího SDK jsem se věnoval rešerši nejpopulárnějších SDK napsaných v jazyce Python. Jsou jimi knihovny *Boto3*, *Sentry SDK* a *Docker SDK*. Zaměřil jsem se na jejich obecný návrh, architekturu a přístupy ke konfiguraci, dokumentaci, verzování či testování. Poté následoval rozbor obecných kvalit kódu při psaní softwarové knihovny.

V následující kapitole jsem popsal předpokládané případy užití nově vznikajícího SDK, jež jsou ve velké míře formulovány tvůrci samotného nástroje DSW. Z těchto případů užití poté byly odvozeny funkční i nefunkční požadavky, na jejichž základě vznikl návrh celé knihovny. Architektura i designová rozhodnutí jsou přitom silně ovlivněny poznatky z analýzy DSW, rešerší stávajících SDK a rozborem kvalitativních aspektů softwarové knihovny. Návrh klade důraz na snadnou rozšiřitelnost, jednoduché použití a návaznost na jednotlivé verze DSW API.

Samotná implementace SDK musela vyřešit několik problémů, jejichž řešení návrh blíže nespecifikuje. Především šlo o nalezení způsobu, jak jednoduše a elegantně definovat jednotlivé datové třídy z domény DSW. Výsledné řešení představují třídy, jejichž atributy jsou definovány jakožto deskriptory. To umožňuje definovat sdílenou logiku každého typu atributu mimo definici samotné datové entity, čímž je zajištěno snadné přidávání či modifikace datových entit v budoucnu.

Kód splňuje konvence a zásady definované normami PEP8 a PEP484 (tedy je typovaný), je řádně zdokumentován a doplněn o externí dokumentaci, popisující všechny náležitosti podstatné pro užívání knihovny. Je pokryt z 95%

několika druhy testů a podléhá měření nejrůznějších softwarových metrik, obojí v rámci automatické kontinuální integrace. SDK bylo rovněž úspěšně distribuováno prostřednictvím softwarového repozitáře PyPI (do budoucna je v plánu automatizovat i tento krok a distribuovat tak každou novou verzi automaticky).

Všechny vytyčené cíle tedy považuji za splněné, především pak ten hlavní, jímž bylo navrhnout a implementovat snadno použitelnou a udržitelnou knihovnu, která umožní sjednotit, zjednodušit a zefektivnit přístup k DSW API.

Literatura

[1] Stránka. *Data Stewardship Wizard* [online] [cit. 9. 2. 2021]. Dostupné z: https://ds-wizard.org.

[2] PERGL, R., HOOFT, R., SUCHÁNEK, M., KNAISL, V., & SLIFKA, J. (2019). "Data Stewardship Wizard": A Tool Bringing Together Researchers, Data Stewards, and Data Experts around Data Management Planning. In: *Data Science Journal*, 18(1), p.59 [online] [cit. 3. 5. 2021]. DOI: 10.5334/dsj-2019-059. Dostupné z https://datascience.codata.org/articles/10.5334/dsj-2019-059/.

[3] SUCHÁNEK, Marek. *Re: Diplomová práce* [elektronická pošta]. Příjemce: drahoja9@fit.cvut.cz. 12. listopadu 2020 19:54 [cit. 4. 5. 2021]. *Osobní komunikace.*

[4] Data Management Plans. In: *USGS* [online]. © 2021 USGS [cit. 12. 2. 2021]. Dostupné z: https://www.usgs.gov/products/data-and-tools/data-management/data-management-plans.

[5] TEPEREK, Marta; Maria J. CRUZ; Ellen VERBAKEL; Jasmin K. BÖHMER a Alastair DUNNING. Data Stewardship – addressing disciplinary data management needs. In: *OSF* [online]. © 2011-2021 Center for Open Science [cit. 12. 2. 2021]. DOI: 10.31219/osf.io/5w9pj. Dostupné z: https://osf.io/5w9pj/.

[6] WILKINSON, M.; DUMONTIER, M.; AALBERSBERG, I. et al. The FAIR Guiding Principles for scientific data management and stewardship. In: *Sci Data* [online]. 2016, roč. 3, č. 160018 [cit. 12. 2. 2021]. DOI: 10.1038/sdata.2016.18. Dostupné z: https://www.nature.com/articles/sdata201618.

[7] FAIR Principles. In: *GO FAIR* [online] [cit. 12. 2. 2021]. Dostupné z: https://www.go-fair.org/fair-principles/.

[8] BERNERS-LEE, Tim. Linked Data. In: *W3C* [online]. Copyright © 2021 W3C [cit. 12. 2. 2021]. Dostupné z: https://www.w3.org/DesignIssues/LinkedData.html.

[9] Lesson 1: Introduction to the Web of Data. In *ReproNim* [online]. © 2016 Neurohackweek and ReproNim [cit. 13. 2. 2021]. Dostupné z http://www.repronim.org/module-FAIR-data/01-Web-of-Data/.

[10] What is the difference between "FAIR data" and "Open data" if there is one? In: *GO FAIR* [online] [cit. 12. 2. 2021]. Dostupné z: https://www.go-fair.org/resources/faq/ask-question-difference-fair-data-open-data/.

[11] LUQUE, Cristina. Open Data and FAIR Data: differences and similarities. In *OGoov* [online]. © 2019 MCA – Viafirma SL [cit. 13. 2. 2021]. Dostupné z https://www.ogoov.com/en/blog/open-data-and-fair-data-differences-and-similarities/.

[12] SUCHÁNEK, Marek. Data Stewardship Wizard: Představení. In: *Zenodo* [online] [cit. 13. 2. 2021]. DOI: 10.5281/zenodo.4534227. Dostupné z: https://zenodo.org/record/4534227.

[13] MONSE, Barend. *Data Stewardship for Open Science*. Implementing FAIR Principles. 1. vydání. Boca Raton: Chapman and Hall/CRC, 2018. 244 stran. ISBN 978-0-8153-4818-4.

[14] SUCHÁNEK, Marek. DS Wizard dokumentace. In: *Zenodo*[online] [cit. 14. 2. 2021]. DOI: 10.5281/zenodo.4495697. Dostupné z https://zenodo.org/record/4495697#.YDPVCav9aiM.

[15] SUCHÁNEK, Marek. Initial DSW diagrams. In: *Zenodo* [online] [cit. 14. 2. 2021]. DOI: 10.5281/zenodo.2617265. Dostupné z https://zenodo.org/record/2617265#.YDPU6Kv9aiM.

[16] Data Stewardship Wizard Engine Backend, tag v2.11.0. In: *GitHub* [online]. © 2021 GitHub, Inc. [cit. 20. 2. 2021]. Dostupné z: https://github.com/ds-wizard/engine-backend.

[17] Data Stewardship Wizard Engine Frontend, tag v2.11.0. In: *GitHub* [online]. © 2021 GitHub, Inc. [cit. 20. 2. 2021]. Dostupné z: https://github.com/ds-wizard/engine-frontend.

[18] Data Stewardship Wizard Document Worker, tag v2.11.0. In: *GitHub* [online]. © 2021 GitHub, Inc. [cit. 20. 2. 2021]. Dostupné z: https://github.com/ds-wizard/document-worker.

[19] Data Stewardship Wizard Template Development Kit, tag v2.11.0. In: *GitHub* [online]. © 2021 GitHub, Inc. [cit. 20. 2. 2021]. Dostupné z: https://github.com/ds-wizard/dsw-tdk.

[20] DSW Swagger. In: *Swagger* [online] [cit. 20. 2. 2021]. Dostupné z: https://api.demo.ds-wizard.org/swagger-ui/.

[21] SHUKLA, Shashvat. Framework vs Library vs Platform vs API vs SDK vs Toolkits vs IDE. In *Medium* [online] [cit. 23. 2. 2021]. Dostupné z: https://medium.com/@shashvatshukla/framework-vs-library-vs-platform-vs-api-vs-sdk-vs-toolkits-vs-ide-50a9473999db.

[22] Explainer: SDKs vs Libraries. In: *Digiday* [online] [cit. 23. 2. 2021]. Dostupné z: https://digiday.com/media/explainer-sdks-vs-libraries/.

[23] MENSCH, Tim. What is difference between Toolkit, Library and SDK? In: *Quora* [online] [cit. 23. 2. 2021]. Dostupné z: https://www.quora.com/What-is-difference-between-Toolkit-Library-and-SDK.

[24] MICHON, Mark. API vs. SDK. In: *Bearer* [online] [cit. 23 2. 2021]. Dostupné z: https://blog.bearer.sh/api-vs-sdk/.

[25] Boto3, tag 1.17.16. In: *GitHub* [online]. © 2021 GitHub, Inc. [cit. 26. 2. 2021]. Dostupné z https://github.com/boto/boto3.

[26] Oficiální dokumentace Boto3, verze 1.17.16. In: *AmazonAWS* [online]. © Copyright 2021, Amazon Web Services, Inc. [cit. 26. 2. 2021]. Dostupné z: https://boto3.amazonaws.com/v1/documentation/api/latest/index.html.

[27] AWS SDKs and Tools Maintenance Policy. In: *AWS* [online]. © 2021, Amazon Web Services, Inc. [cit. 26. 2. 2021]. Dostupné z: https://docs.aws.amazon.com/credref/latest/refdocs/maint-policy.html.

[28] Stránka. *Sémantické verzování 2.0.0* [online] [cit. 26. 2. 2021]. Dostupné z: https://semver.org/lang/cs/.

[29] Oficiální dokumentace Python Sentry SDK, verze 1.0.0. In: *Sentry* [online] [cit. 5. 3. 2021]. Dostupné z: https://docs.sentry.io/platforms/python/.

[30] Sentry-python, tag 1.0.0. In: *GitHub* [online]. © 2021 GitHub, Inc. [cit. 5. 3. 2021]. Dostupné z https://github.com/getsentry/sentry-python.

[31] Oficiální dokumentace Docker SDK pro Python, verze 4.4.4. In: *Read The Docs* [online]. ©2021 Docker Inc. [cit. 5. 3. 2021]. Dostupné z: https://docker-py.readthedocs.io/en/4.4.4/.

[32] Docker-py, tag 4.4.4. In: *GitHub* [online]. © 2021 GitHub, Inc. [cit. 5. 3. 2021]. Dostupné z https://github.com/docker/docker-py.

[33] FRANTZ, Rafael Z.; CORCHUELO, Rafael; ROOS-FRANTZ, Fabricia. On the design of a maintainable software development kit to implement integration solutions. In: *ournal of Systems and Software* [online]. 2016, svazek 111. DOI: 10.1016/j.jss.2015.08.044. ISSN 0164-1212. Dostupné z: https://www.sciencedirect.com/science/article/pii/S0164121215001880.

[34] YURKOVICH, James T.; YURKOVICH Benjamin J.; DRÄGER, Andreas; PALSSON, Bernhard O.; KING, Zachary A. A Padawan Programmer's Guide to Developing Software Libraries. In: *Cell Systems* [online]. 2017, roč. 5, č. 5 [cit. 8. 3. 2021]. DOI: 10.1016/j.cels.2017.08.003. ISSN 2405-4712. Dostupné z: https://www.sciencedirect.com/science/article/pii/S2405471217303368.

[35] LEVINSKY, Gal. 10 Tips on How to Build the Perfect SDK. In. *DZone* [online] [cit. 8. 3. 2021]. Dostupné z: https://dzone.com/articles/10-tips-on-how-to-build-the-perfect-sdk.

[36] SANDOVAL, Kristopher. Best Practices for Building SDKs for APIs. In: *Nordic APIs* [online]. © 2013-2021 Nordic APIs AB [cit. 7. 3. 2021]. Dostupné z: https://nordicapis.com/best-practices-for-building-sdks-for-apis/.

[37] GUO, Jenks. Build an API Client SDK, the Right Way. In: *Sweetcode* [online]. © 2020 Fixate IO, LLC [cit. 8. 3. 2021]. Dostupné z: https://sweetcode.io/build-api-client-sdk/.

[38] IEEE Standard Glossary of Software Engineering Terminology. In *IEEE Std 610.12-1990* [online] [cit. 6. 3. 2021]. DOI: 10.1109/IEEESTD.1990.101064. Dostupné z: https://ieeexplore.ieee.org/document/159342.

[39] ESTDALE J., GEORGIADOU E. Applying the ISO/IEC 25010 Quality Models to Software Product. In: *Communications in Computer and Information Science* [online]. 2018, svazek 896. Springer, Cham. DOI: 10.1007/978-3-319-97925-0_42. Dostupné z: https://link.springer.com/chapter/10.1007/978-3-319-97925-0_42.

Příloha **A**

Seznam použitých zkratek

API Application Programmer Interface

AWS Amazon Web Services

CRUD Create, Read, Update, Delete

DMP Data Managemet Plan

DSW Data Stewardship Wizard

FAIR Findability, Accessability, Interoperability, Reusability

KM Knowledge Model

LOD Linked Open Data

SDK Software Development Kit

TDK Template Development Kit

Příloha **B**

Obsah přiloženého CD

```
│ README.md ..................................... stručný popis obsahu CD
├── src
│   ├── impl ..................................... zdrojové kódy implementace
│   └── thesis ................................... zdrojová forma práce ve formátu LaTeX
└── text ......................................... text práce
    ├── DP_Drahos_Jakub_2021.pdf ............... text práce ve formátu PDF
    └── DP_Drahos_Jakub_2021.ps ................ text práce ve formátu PS
```