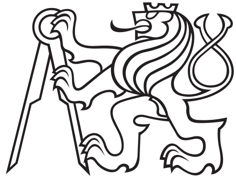
Diplomová práce



České
vysoké
učení technické
v Praze

F3

Fakulta elektrotechnická
Katedra počítačů

Generování kódu z modelů Enterprise Architect

Bc. Martin Mašata

Vedoucí: Ing. Jiří Šebek
Oponent: Ing. David Kadleček, Ph.D.
Studijní program: Otevřená informatika
Specializace: Softwarové inženýrství
Květen 2023

ČVUT
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Mašata** Jméno: **Martin** Osobní číslo: **474595**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Otevřená informatika**

Specializace: **Softwarové inženýrství**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Generování kódu z modelů Enterprise Architect

Název diplomové práce anglicky:

Generating code from Enterprise Architect models

Pokyny pro vypracování:

Motivace: Velké množství firem používá pro architekturu aplikací Enteprise Architect
a na základě toho vývojáři tvoří aplikaci.
Tvorba základní struktury aplikace (API endpointy, payloady, DTO, ...) si vezme svůj čas.
Pro zrychlení procesu je vhodné vygenerovat dle modelů skeleton aplikace,
do kterého se už pouze naimplementuje business logika aplikace.
Cíl práce: Cílem práce je zanalyzovat databázovou strukturu sparxEA databáze a na základě toho vytvořit
generátor kódu.
Generátor by měl být navržen tak, aby byl do jisté míry konfigurovatelný a rozšiřitelný. Požadavky na funkcionalitu:
1) Konfigurovatelnost metodiky modelování ve Enterprise Architect
2) Modulárnost - umožnění snadno přidat generování do dalšího programovacího jazyka
2) Generování struktury pro API endpointy
3) Generování DTO, Payloadů a jiných modelů
4) Generování Kafka Consumerů a Producerů
Výsledná práce bude obsahovat analýzu, návrh, implementaci a dokumentaci pro používání.
Implementace bude obsahovat generátor pro Java kód, ale bude navržena tak, že bude snadné
rozšířit o další programovací jazyky.

Seznam doporučené literatury:

Cao, Hanyang, Jean-Rémy Falleri, and Xavier Blanc. "Automated generation of REST API specification from plain HTML documentation." Service-Oriented Computing: 15th International Conference, ICSOC 2017, Malaga, Spain, November 13–16, 2017, Proceedings. Springer International Publishing, 2017.
SparxEA documentation -
https://sparxsystems.com/enterprise_architect_user_guide/15.2/guidebooks/tools_ba_documentation.html
Kafka documentation - https://kafka.apache.org/documentation/
Lankhorst, Marc. Beyond Enterprise Architecture. Springer Berlin Heidelberg, 2005.
Garg, Nishant. Learning Apache Kafka. Packt Publishing, 2015.

Jméno a pracoviště vedoucí(ho) diplomové práce:

Ing. Jiří Šebek kabinet výuky informatiky FEL

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

Datum zadání diplomové práce: **10.02.2023** Termín odevzdání diplomové práce: _____________

Platnost zadání diplomové práce: **22.09.2024**

Ing. Jiří Šebek podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

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

Datum převzetí zadání	Podpis studenta

© VUT v Praze, Design: VUT v Praze, VIC

Poděkování

Velice rád bych poděkoval mému vedoucímu Ing. Jiřímu Šebkovi za cenné rady a za vedení této diplomové práce. Dále bych také rád poděkoval celé mé rodině a mé přítelkyni za nesmírnou podporu, které se mi během mého studia dostávalo.

Prohlášení

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

V Praze, 23. května 2023

Abstrakt

Tato diplomová práce se zabývá tvorbou frameworku pro generování kódu a dokumentace z Enterprise Architect modelů, který bude konfigurovatelný. Dokument obsahuje analýzu, návrh, implementaci a testování aplikace. Analýza se skládá z pochopení nástroje, model driven developmentu a z potřeby modelování ve dvou společnostech. V návrhu se poté řeší volba technologií a návrh jednotlivých komponent, vše je doprovázeno diagramy. V implementaci a testování se řeší problémy, které nastaly a jaké chyby byly opraveny. Výsledná aplikace byla otestována a je již používána, díky čemu vznikají návrhy pro další vylepšování.

Klíčová slova: sparxEA, EA, OpenApi, Swagger, UML, generátor, MDD

Abstract

This master's thesis deals with the creation of a framework for generating code and documentation from Enterprise Architect models, which will be configurable. The document contains the analysis, design, implementation and testing of the application. The analysis consists of understanding the tool, model driven development and the need for modeling in two companies. The design then addresses the choice of technologies, the design of the components, all accompanied by diagrams. In implementation and testing, problems that occurred are addressed, what bugs were fixed. The resulting application has been tested and is already in use, resulting in suggestions for further improvements.

Keywords: sparxEA, EA, OpenApi, Swagger, UML, generator, MDD

Title translation: Generating code from Enterprise Architect models

Obsah

1 Úvod 1

1.1 Motivace 1

1.2 Cíle diplomové práce 2

2 Analýza 3

2.1 Model Driven Engineering 3

2.1.1 Model Driven Development . . . 4

2.2 Enterprise Architect 4

2.2.1 O nástroji 5

2.2.2 Verze nástroje 5

2.2.3 Komunikace s generátorem . . . 6

2.2.4 Struktura databáze 7

2.3 UML Class diagram 9

2.4 Generování 11

2.4.1 Datový model 12

2.4.2 REST API endpointy 13

2.4.3 Avro schemata pro Kafku . . . 15

2.5 Již zavedené způsoby modelování 16

2.5.1 Model ze společnosti A 16

2.5.2 Model ze společnosti B 17

2.6 Funkční požadavky 18

2.7 Nefunkční požadavky 18

2.8 Use case 19

3 Návrh 21

3.1 Volba techstacku 21

3.2 Architektura frameworku 22

3.2.1 Common-api 24

3.2.2 Mapování z EA do common-api 26

3.2.3 Generování souborů z common-api 27

3.2.4 Validátor common-api 28

3.3 Konfigurovatelnost frameworku . 29

4 Implementace 31

4.1 SpringBoot starter knihovna . . . 31

4.2 Implementace frameworku 32

4.2.1 Konfigurační soubor 32
4.2.2 Inicializace kontextu 33
4.2.3 Spuštění frameworku 35
4.2.4 Nadefinování API pro uživatele 36
4.3 Vestavěné generátory 39
4.3.1 Java generátor 39
4.3.2 Swagger generátor 41
4.3.3 Avro schema generátor 42
4.4 Použité best practices 43
4.5 Použité vzory 43
4.6 Problémy při implementaci a řešení 44
4.7 Využití aplikace v konkrétní společnosti 45
5 Testování **47**
5.1 Způsob testování 47
5.2 Testy jádra frameworku 48
5.3 Uživatelské testy 49
5.3.1 Persona 49
5.3.2 Testovací scénář 49
5.3.3 Výsledky uživatelských testů 50
5.4 Nalezené chyby a návrhy na vylepšení 51
6 Závěr **53**
6.1 Výsledky práce 53
6.2 Splnění cílů 53
6.3 Další rozšiřitelnost a doporučení 54
Literatura **55**
A Seznam použitých zkratek **59**
B Readme dokumentace frameworku **61**

Obrázky

2.1 MDE a jeho podoblasti [1] 3

2.2 Databázová struktura EA nejdůležitějších entit 7

2.3 Třída v UML diagramu 9

2.4 Vztahy v UML třídách [2] 10

2.5 Zakreslení jednotlivých vztahů UML Class diagramu 11

2.6 Typy datových modelů [3] 12

2.7 Příklad možných způsobů kreslení API 13

2.8 Využití Avro schemat s Kafkou [4] 15

2.9 Cesta API ve společnosti A 16

2.10 API resource ve společnosti A . 16

2.11 API resource ve společnosti B . 17

2.12 Request a Response objekty API ve společnosti B 17

2.13 Use case diagram pro výsledný framework na generování 19

3.1 Komponent diagram navrhovaného frameworku 23

3.2 Activity diagram jednoúčelných generovacích skriptů 24

3.3 Activity diagram generovacího frameworku s common-api 24

3.4 Model struktury common-api . . . 25

3.5 Komponent diagram mapování do common-api 26

3.6 Komponent diagram tvorby a spouštění generátorů 27

3.7 Sekvenční diagram validátoru common-api 28

3.8 Sekvenční diagram konfigurace dle vstupních parametrů 30

5.1 Testovací scénář 49

Tabulky

2.1 Přehled posledních major verzí EA [5] 5

2.2 Seznam možných multiplicit ve vztazích u UML Class diagramu [6] 9

2.3 Seznam požadavků na generátor datového modelu 12

2.4 Seznam požadavků na generátor REST API 14

3.1 Souhrn techstacku a verzí. 21

3.2 Souhrn vstupních parametrů generátoru 29

4.1 Seznam parametrů pro Java generátor 39

4.2 Seznam parametrů pro Swagger generátor 41

4.3 Seznam parametrů pro Avro schema generátor 42

5.1 Seznam unit testů 48

5.2 Seznam chyb a návrhů na vylepšení 51

Kapitola 1

Úvod

1.1 Motivace

Velké množství společností používá Model Driven Software Development metodiku spolu s aplikací Enterprise Architect, kde zakresluje modely. Následně řeší, jakým způsobem zautomatizovat tvorbu kódu a dokumentace z nakreslených modelů. Všechny tyto společnosti řeší úplně stejný problém, jediné co se liší jsou procesy ve firmě a pravidla, které dodržují při modelování. Na druhé straně je zde výstup zautomatizovaného skriptu, který je téměř totožný - standardizovaná dokumentace či vygenerovaný kód.

Z toho vyplývá, že všechny tyto společnosti řeší totožný problém, ale musí si vymyslet své vlastní řešení. Celý tento proces by usnadnil konfigurovatelný nástroj, u kterého by stačilo doplnit pravidla modelování dané společnosti a generování výstupu by již bylo v nástroji zahrnuto. Tvorbou tohoto nástroje by se adopce Model Driven Development v dané společnosti razantně zrychlila a celý proces by netrval tak dlouho. Z tohoto důvodu vznikl nápad na téma diplomové práce, a proto se v této práci budeme tímto problémem zabývat.

1.2 Cíle diplomové práce

Cílem této práce je pochopit základní principy Enterprise Architect, jaká jsou pravidla u UML class diagramů a celkově princip Model Driven Development. Na základě toho provést analýzu nad konkrétními způsoby modelování ve dvou společnostech, které již Model Driven Development zavádějí a pochopit, jaké mají potřeby pro modelování a generování výstupů. Z toho všeho by měli vzniknout funkční a nefunkční požadavky.

Práce by poté měla obsahovat návrh aplikace, která bude konfigurovatelná a bude splňovat všechny požadavky, které vyšly z analýzy. Hotová aplikace by měla být snadno použitelná, snadno rozšiřitelná s dobrou dokumentací, pro její uživatele. Kód aplikace by měl dodržovat všechny náležitosti, jak má vypadat čistý kód, včetně testů.

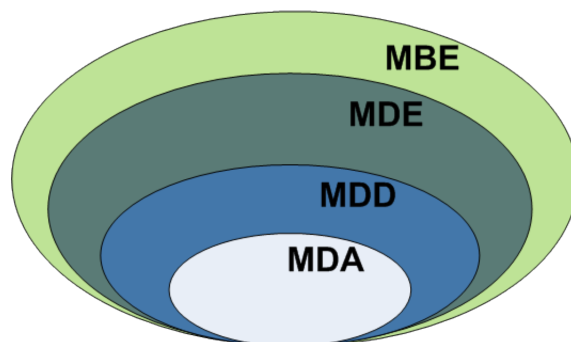
Kapitola 2

Analýza

2.1 Model Driven Engineering

Model Driven Engineering (MDE) je metodika vývoje softwaru, která se zaměřuje na vytváření abstraktních modelů a následně na generování kódu z těchto modelů. Cílem MDE je zlepšit produktivitu a kvalitu vývoje softwaru, snížit náklady a zkrátit dobu vývoje.

MDE si zakládá na myšlence, že tvorba modelů je daleko efektivnější než psaní kódu, a že modely jsou lepším způsobem pro popis a analýzu složitých systémů. Lze ho použít v různých oblastech softwarového vývoje, kde jsou vysoké úrovně abstrakce a složitosti. Ať už se jedná o webové aplikace, embedded systémy nebo jiné [7]. Na obrázku 2.1 jsou vyobrazeny podoblasti MDE.



Obrázek 2.1: MDE a jeho podoblasti [1]

2.1.1 Model Driven Development

Model Driven Development (MDD) je jedním z hlavních přístupů v rámci MDE, který se zaměřuje na tvorbu modelů jako hlavního zdroje pro generování kódu. Je to způsob vývoje software, který má být rychlý a efektivní. Jako jednou z hlavních výhod je minimalizace nedorozumění mezi rolemi napříč vývojem software a to hlavně díky tomu, že danému modelu rozumí všichni. Jelikož je kód generován z modelů, tak od úpravy modelů až po nasazení nové verze aplikace je celý proces velice rychlý. Další výhodou modelu je, že nejsou závislé na konkrétním programovacím jazyku, díky tomu model může používat více týmů, kteří vyvíjí v různých programovacích jazycích a s celkově jiným tech stackem [8, 9, 10].

Jako jednu z velkých nevýhod můžeme brát nutnost kvalitních a správně navržených modelů. Pokud je již model jako takový špatně navržený, pak to může vývoj velice zpomalit. Další potencionální nevýhodou je, že vývojář nemá dostatečnou úroveň znalostí modelování.

2.2 Enterprise Architect

V této části analýzy se budeme zabývat nástrojem Enterprise Architect (EA) od společnosti Sparx Systems. Zjistíme, k čemu všemu lze nástroj využít, jak moc je komplexní a co vše umí. Dále zjistíme, jaká je aktuální verze, jak se liší od historických verzí a vydefinujeme, od jaké verze budeme podporovat komunikaci s naší výslednou aplikací. Jelikož budeme muset pracovat s daty z EA, tak bude nutné také pochopit základní databázovou strukturu EA, co si hlavní tabulky ukládají, a jaké vazby spolu mají.

2.2.1 O nástroji

EA je komplexní nástroj pro modelování, návrh, vizualizaci, správu softwarových systémů a dalších podnikových procesů. Disponuje velkým množství funkcí, které pokrývají celý životní cyklus vývoje systému, včetně jeho struktury, chování a interakce s okolním prostředím. Byl vyvinut společností Sparx Systems, která vydala jeho první verzi již v roce 2000 [11].

Mezi hlavní funkce EA patří především vytváření a správa modelů softwarových systémů pomocí různých typů diagramů. Další důležitou funkcí je integrace s velkým množstvím externích nástrojů jako je kupříkladu Jira, Confluence, SharePoint nebo také verzovací nástroje a nástroje pro řízení změn. Umožňuje sdílení modelů a diagramů s ostatními týmy v rámci organizace [12].

Přestože je nástroj natolik robustní, tak pro naše účely v rámci této diplomové práce budeme potřebovat pouze základní funkce a to je konkrétně modelování UML diagramu tříd - to bude naše priorita. Pouze v tomto typu UML diagramů budou kresleny modely, ze kterých chceme generovat výstupy [11, 12].

2.2.2 Verze nástroje

K aktuálnímu dni je dostupná nejnovější verze nástroje 16.1. Nástroj se pak kromě různých verzí ještě liší edicemi - Ultimate, Unified, Corporate, Professional. Každá z těchto edic disponuje jinou množinou funkcí, které nabízí a od toho se také odvíjí cena za licenci. Pro nás jsou důležité z pohledu generátoru pouze UML diagramy tříd, které podporují všechny tyto edice, proto edice řešit nemusíme. Lze říci, že naše výsledná aplikace bude podporovat všechny typy edic. V tabulce 2.1 jsou uvedeny verze a jejich rok vydání.

Verze	Rok vydání
16	2022
15	2019
14	2018
13	2016
12	2015

Tabulka 2.1: Přehled posledních major verzí EA [5].

Je nutné si také říci, od jaké verze EA budeme podporovat integraci s naším generátorem. Každá nová verze přináší mnoho nových funkcí a vylepšení těch stávajících. Z tohoto pohledu by bylo nejrozumnější podporovat pouze verzi 16 a novější. Bohužel v jedné ze společnosti, do které budeme chtít tento nástroj zavádět se používá EA verze 13. Z toho důvodu náš generátor bude podporován již od verze EA 13 a výše. Naštěstí základní funkce, které budeme potřebovat se již nemění a zůstavají stejné napříč verzemi. Pokud by někdo chtět generátor využít i ve starších verzích, tak nelze na 100 % zaručit, že by fungoval, ale je velmi pravděpodobné, že bude [13].

2.2.3 Komunikace s generátorem

Dalším důležitým bodem k analýze je najít způsob, jak bude generátor získávat data z EA. Nabízí se dvě možnosti - použít API a nebo se připojit přímo k databázi dané instance EA. Musíme zvážit výhody a nevýhody těchto dvou možností a zvolit tu nejlepší pro naše účely.

Prvním ovlivňujícím faktorem je bezpečnost. Posílání dat je daleko bezpečnější přes API než-li přes přímé připojení do databáze. Na úrovni API můžeme integrovat další bezpečností mechanismy. V našem případě bude generátor vždy ve vnitřní síti společnosti, stejně jako databáze EA, proto v tomto případě nás tento bod neomezuje.

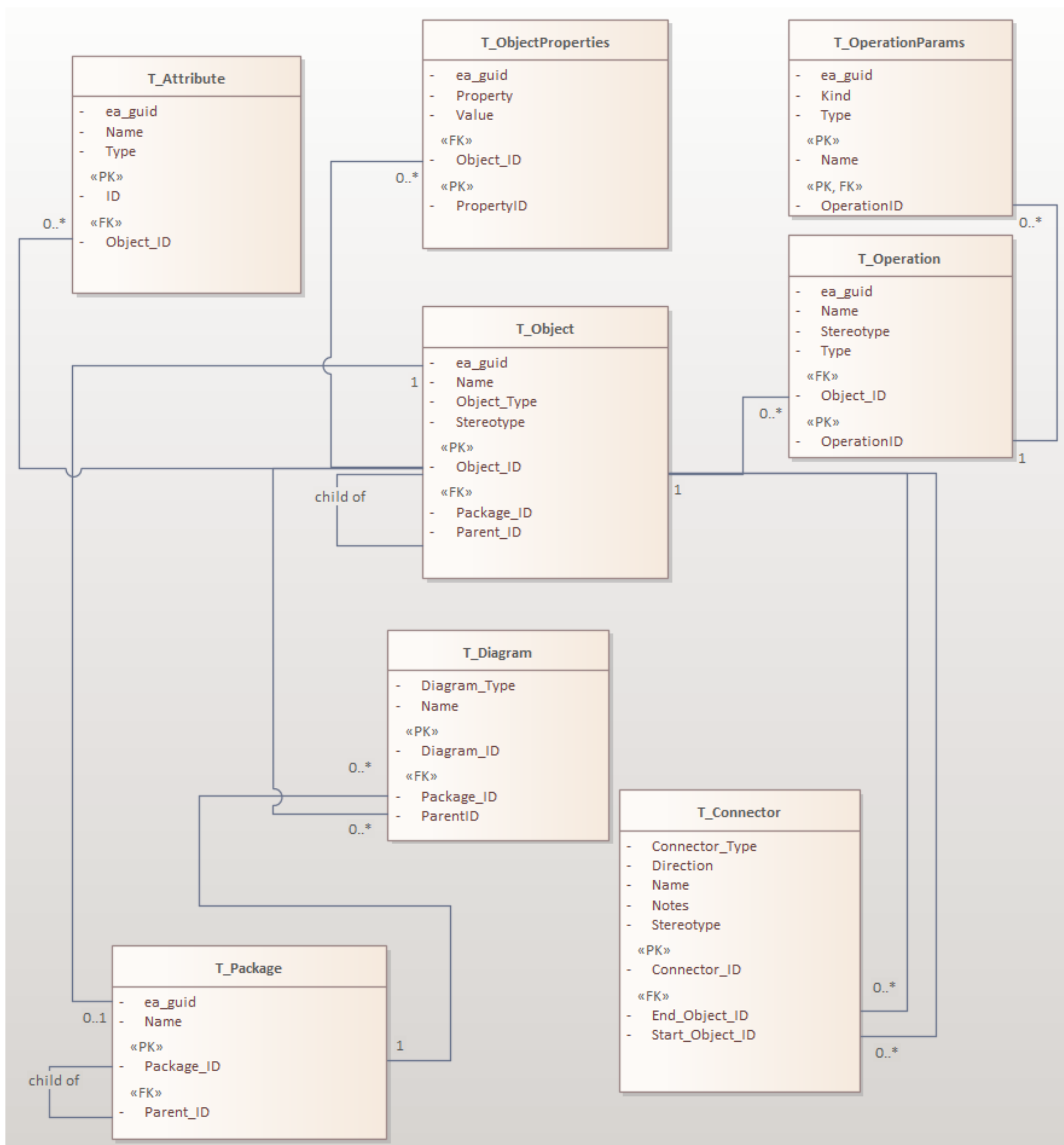
Dalším faktory k zamyšlení je výkon. Přímé připojení k databázi bude vždy rychlejší než API, protože mezi daty a generátorem nejsou žádné další vrstvy abstrakce, které by zpomalovaly jakkoliv komunikaci. Cílem je, aby generátor vygeneroval výstupy co nejrychleji, proto v tomto bodě vítězí přímé připojení do databáze.

Posledním pro nás důležitým faktorem je flexibilita. Potřebujeme přístup ke všem datům, nikoliv jen k vybrané podmnožině, které nám API zpřístupní. Jelikož generátor potřebuje být flexibilní je pro nás i v tomto ohledu lepší volbou databáze.

S ohledem na několik faktorů se pro naše účely zdá vhodnější použít přímé připojení do databáze. Generátor tedy použije tento způsob komunikace. V dané společnosti bude potřeba zajistit uživatele do databáze pro generátor (pouze pro čtení) a síťové prostupy mezi strojem, na kterém běží generátor, a kde se nachází instance EA databáze.

2.2.4 Struktura databáze

Na základě rozhodnutí použití přímého databázového připojení navazuje tato kapitola, která popisuje základní strukturu EA databáze. Tuto strukturu je nutné znát, protože s ní generátor bude pracovat. Na oficiálním blogu Sparx Systems jsem nalezl jeden článek s obrázkem s EA databázovou strukturou, který mi pomohl si vytvořit vlastní diagram 2.2 s tabulkami, které budu potřebovat. Některé tabulky byly zmíněny v článku a některé jsem musel pomocí SQL dotazů dozjistit sám [14].



Obrázek 2.2: Databázová struktura EA nejdůležitějších entit

Nejdůležitější databázovou tabulkou je T_Object. Každý její záznam obsahuje základní prvky jako je třeba Třída, Složka, Datový typ, Enumerace a další. O jaký typ prvku se jedná je uvedeno ve sloupci Object_Type. Dále nesmí chybět sloupce s názvem (Name), Stereotype a další. Nad touto tabulkou pravděpodobně budeme nejvíce vyhledávat, abychom našli prvky pro nás důležité.

Další velmi důležitou tabulkou je T_Connector. Obsahuje základní informace o propojení mezi dvěma prvky v diagramu (T_Object). Diagramy tříd podporují více typů vazeb, tato informace o typu vazby je uložena ve sloupci Connector_Type. Směr vazby lze jednoynačně určit ze tří sloupců - Start_Object_ID, End_Object_ID a Direction. Stejně jako prvky může mít i vazba svůj Stereotype. Další informace o propojení se pak nacházejí v dalších tabulkách jako je třeba T_ConnectorProperty, ale pro naše účely by měla postačit pouze T_Connector tabulka.

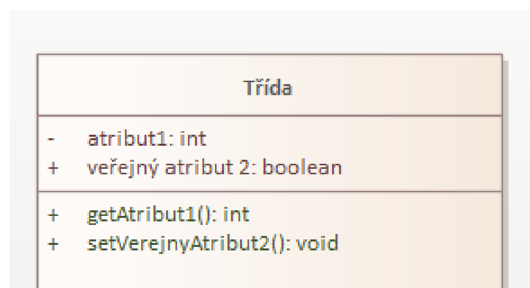
Atributy jednotlivých tříd a jejich datové typy, viditelnosti a další informace jsou uloženy v tabulce T_Attribute. Metody tříd nebo jiné funkce a jejich vstupy a návratové hodnoty se nachází v tabulce T_Operation.

Všechny ostatní tabulky zakreslené, v obrázku výše, jsou z pohledu generátoru poddružné. Nebudeme je využívat často, ale pokud budeme potřebovat detailnější informace o hlavních prvcích, tak je můžeme získat odtamtud.

2.3 UML Class diagram

V této kapitole si shrneme pravidla pro UML Class diagram, což je typ diagramu, který se používá pro vizualizaci struktury tříd v programovacích jazycích. Jeho pochopení je nutné pro další části práce, které ze znalosti toho typu diagramu vychází.

Základním stavebním blokem je entita třída. Reprezentuje konkrétní typ objektu, který disponuje atributy a operacemi (metodami). Každá třída musí mít jméno, které vystihuje účel daného objektu. Atributy musejí mít nadefinované svoje datové typy a viditelnosti. U operací musejí být také vydefinované viditelnosti a k tomu jejich návratový typ. Viditelností máme na mysli přístup k daným prvkům třídy, ty můžou nabývat hodnotám public, protected či private [15, 16]. Ukázku entity třída lze vidět na obrázku 2.3.



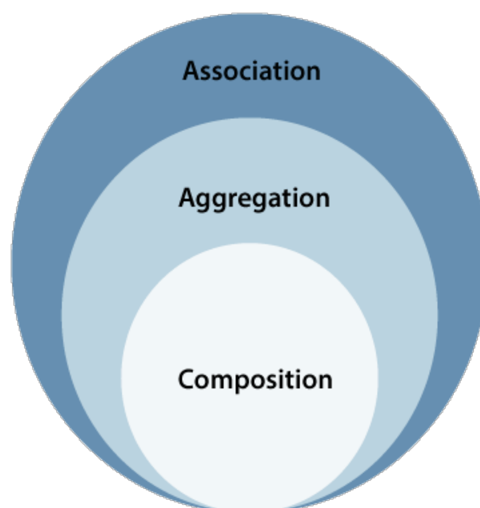
Obrázek 2.3: Třída v UML diagramu

Mezi entitami se definují vztahy. Vztahy určují, jak jsou jednotlivé třídy propojeny a jakým způsobem spolu komunikují. Vztahy se dále dělí na několik kategorií, nejčastěji mezi Asociaci, Generalizaci, Agregaci nebo Kompozici. Každý vztah musí obsahovat také multiplicitu. Multiplicita říká, kolik instancí jedné třídy může být spojeno s instancemi třídy druhé. Seznam možných multiplicit a jejich reprezentací zápisu jsou uvedeny v tabulce 2.2 níže.

Zápis	Význam
0..1	jedna nebo žádná instance
1	právě jedna instance
0..*	žádná nebo více instancí
1..*	jedna nebo více instancí
*	jakýkoli počet instancí
n..m	alespoň n instancí a nejvýše m instancí

Tabulka 2.2: Seznam možných multiplicit ve vztazích u UML Class diagramu [6]

Kategorie vztahů Asociace, Agregace a Kompozice jsou spolu úzce spojeny. Agregace a Kompozice jsou speciálním případěm Asociace. Kompozice je poté speciálním případem Agregace. Lze říci, že Agregace je přísnější než Asociace a Kompozice je přísnější než-li Agregace. Vztah mezi kategoriemi lze také reprezentovat pomocí Vennových diagramů, který se nachází na obrázku 2.4.



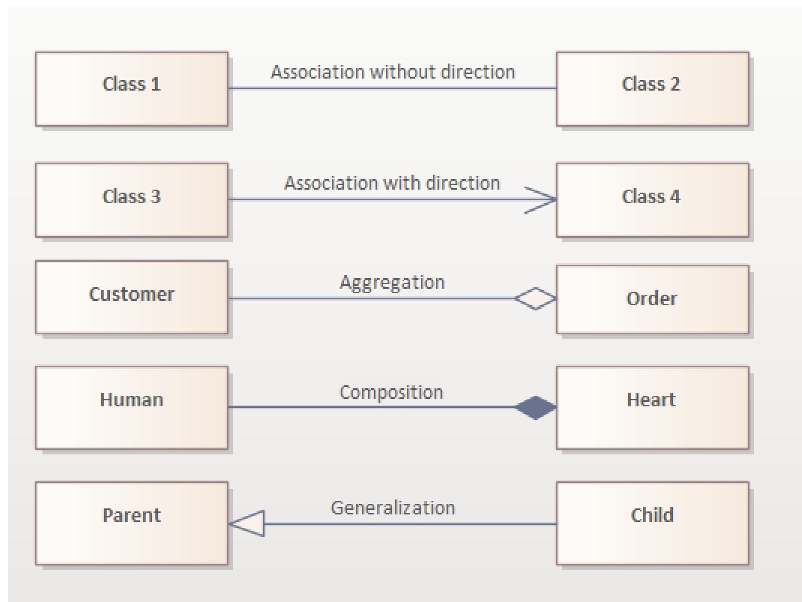
Obrázek 2.4: Vztahy v UML třídách [2]

Nejméne striktní vazbou je Asociace. Ta reprezentuje propojení mezi dvěma nebo více třídami. Pokud je vazba bez směrové šipky, pak mají na sebe referenci obě třídy navzájem. Pokud naopak vazba směrovou šipku obsahuje, pak si drží jedna třída referenci na druhou a nikoliv naopak. Tato vazba nedefinuje žádnou závislost, tedy třídy jsou schopny existovat i bez existence druhé třídy [17].

Následuje Agregace, která je již striktnější než Asociace. Zde jedna třída (celek) obsahuje druhou třídu (součást). Součást nemůže existovat bez existence celku, ale celek může existovat bez existence součásti. Agregace je značena vazbou, kde na straně součásti je značena nevyplněným symbolem tvaru diamantu [18].

Nejstriktnější vazbou ja Kompozice, která je přísnější než-li Agregace. Je velice podobná Agregaci, až na ten rozdíl, že všechny spolu spojené třídy touto vazbou nemohou existovat samostatně. Značení je stejné jako u Agregace s jediným rozdílem - symbol tvaru diamantu je plný (je vybarven) [18].

Poslední vazbou, která je pro nás důležitá je Generalizace nebo taky Dědičnost. Jak již z názvu vyplývá, tento vztah reprezentuje dědičnost mezi dvěma třídami. Jedna třída (potomek) zdědí vlasnostni od druhé třídy (rodič). Generalizace se v diagramu značí šipkou směrem od potomka k rodiči. Existuje také vazba, která definuje opačný proces - Specializace [19]. Ukázku zakreslení jednotlivých vztahů lze vidět na obrázku diagramu 2.5.



Obrázek 2.5: Zakreslení jednotlivých vztahů UML Class diagramu

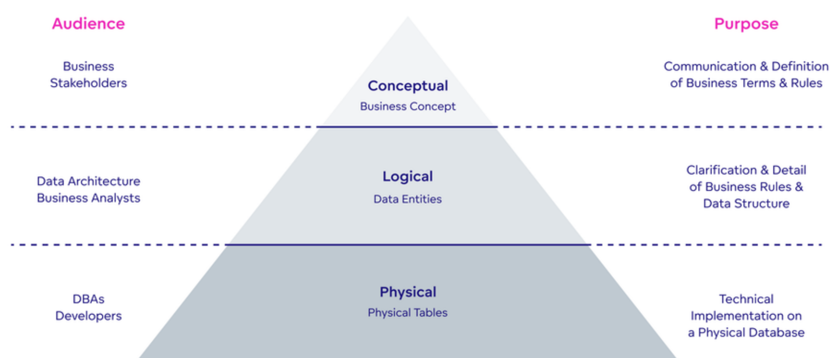
2.4 Generování

V této kapitole se budeme zabývat konkrétními oblastmi generování, které budeme chtít umět generovat ve frameworku. Hlavní a nejdůležitější jsou API endpointy, do čehož spadá pochopení a respektování pravidel správně navržené REST API a Swagger dokumentace. Druhou oblastí jsou datové modely, které jsou úzce spojeny s předchozí oblastí. Poslední oblastí jsou Avro schemata, konkrétně v kontextu využití s Apache Kafkou.

Výstupem této kapitoly by mělo být vydefinování klíčových vlastností, které generátory musí umět, abychom byli schopni sepsat detailní funkční a nefunkční požadavky.

2.4.1 Datový model

Datový model je abstraktní repezentace dat a vztahů mezi nimi v daném systému. Definuje objekty, které představují objekty z realného světa a jejich prvky, vztahy a celkový způsob jejich organizace. Datový model je klíčový při tvorbě systémů. Pomáhá zajistit, aby data byla přesná a konzistentní. Datový model lze dělit na několik kategorií podle typu využití - na konceptuální, logický a fyzický [3], což je vyobrazeno také na obrázku 2.6.



Obrázek 2.6: Typy datových modelů [3]

Z pohledu výsledného generovacího frameworku musíme zajistit, aby u generování modelů uměl generátor přijímat všechny potřebné informace. Je třeba si vydefinovat všechny druhy informací, které datový model uchovává.

Je potřeba, aby dokázal identifikovat objekt jako takový. Dále je nutné, aby u daného objektu uměl rozpoznat všechny jeho atributy. U jednotlivých atributů poté musí rozpoznat název a jejich datový typ - to může být primitivní datový typ jako int, boolean, char, ale taky se může jednat standartní objekt jako je String, Date nebo také úplně nový objekt definován v rámci datového modelu. Dalším požadavkem je rozpoznání multiplicit atributu - zda se jedná o pole hodnot či o jednu hodnotu a pak také povinnost atributu. U atributů nesmí chybět podpora identifikace viditelnosti atributu. Všechny požadavky jsou v tabulce 2.3.

Požadavky na generátor datového modelu
Identifikace objektu v diagramu
Identifikace atributů v objektu
Identifikace datového typu atributu
Identifikace multiplicity atributu
Identifikace povinnosti vyplnění atributu
Identifikace viditelnosti atributu

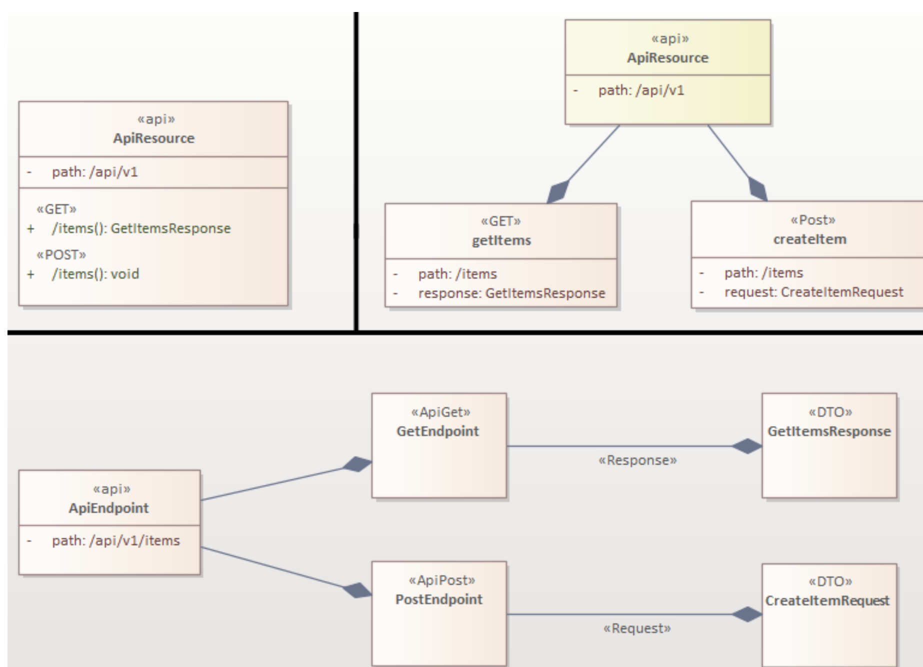
Tabulka 2.3: Seznam požadavků na generátor datového modelu

2.4.2 REST API endpointy

API je sada pravidel, které definují jak se aplikace spolu spojí za účelem komunikace. REST API je architektonický styl komunikace aplikací přes HTTP protokol. Náš generátor musí podporovat řádné zvyklosti tohoto architektonického stylu, tím může do budoucna nasměrovat uživatele, aby do značné míry při modelování tyto pravidla dodržovali [20].

Jedním z těchto pravidel je jednotné rozhraní (Uniform interface), které říká že všechny API dotazy se stejným resource mají vypadat stejně bez ohledu na to, odkud přichází. Z toho vzniká požadavek na generovací framework, aby uměl sdružovat jednotlivé API endpointy do celků - již dříve zmíněných resources [20].

Dalším pravidlem je bezestavovost (Statelessness). Toto pravidlo říká, že každý požadavek musí obsahovat všechny informace potřebné pro jeho zpracování [20]. Toto můžeme ovlivnit nepřímo, tak že generátor bude umět operovat se všemi druhy informací REST API - to můžou být header, query parameters, path parameters, HTTP methods a request/response objekty. Request a response objektu budeme uvažovat pouze JSON objekty (nic jiného náš generátor řešit nebude). Pro tyto účely můžeme využít a přepoužít informace sesbírané v předchozí kapitole na Datové modely [21, 22]. Na obrázku 2.7 lze vidět stejnou reprezentaci dat různými způsoby.



Obrázek 2.7: Příklad možných způsobů kreslení API

Nyní musíme vydefinovat, jaké výstupy bude umět aplikace tvořit. Generovací framework musí podporovat generování Java kódu - konkrétně API interface, které budou kompatibilní se Spring frameworkem. Musí být vygenerované tak, že uživateli bude stačit použít vygenerovaný kód a pouze doplnit business logiku. Dále chceme, aby generátor uměl produkovat Swagger soubory (OAS). Všechny požadavky na generátor REST API jsou uvedeny v seznamu 2.4.

OAS

OpenAPI specifikace je standardizovaný formát pro popis RESTful API. Jedná se o YAML nebo JSON soubor, který detailně popisuje, jaké endpointy API poskytuje, jaké parametry a typy dat očekávají a jaké odpovědi vrací. Slouží jako dokumentace pro vývojáře a lze ji také využít k vygenerování API klientů v různých programovacích jazycích [23].

Velkou výhodou je, že jelikož se jedná o standardizovaný formát, tak téměr všechny nástroje a frameworky pro vývoj API jej podporují. Lze například uvést aplikace jako: Swagger, Postman, Insomnia [24, 25, 26].

Požadavky na generátor REST API
Podpora sdružování API endpointů - podpora resources
Identifikace HTTP metody API endpointu
Identifikace path parametrů
Identifikace query parametrů
Identifikace headers
Podpora Request/Response objektů
Podpora návratových HTTP status kódů
Generování swagger dokumentace

Tabulka 2.4: Seznam požadavků na generátor REST API

2.4.3 Avro schemata pro Kafku

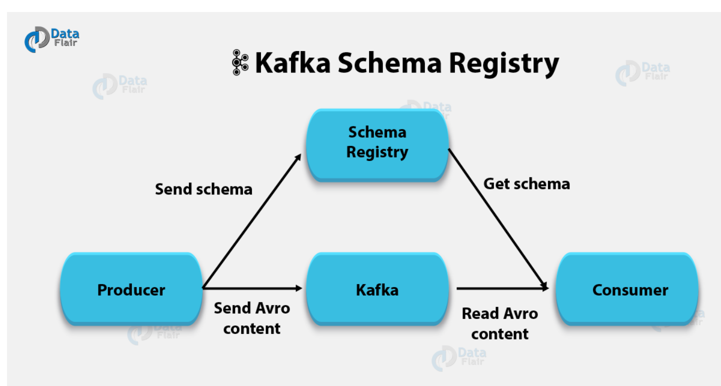
Dalším původním požadavkem bylo generování Kafka consumerů a producerů, kde bylo následně zjištěno, že to vlastně není třeba, místo toho je potřeba generovat Avro schemata, která slouží jako reprezentace formátu, ve kterým data z Apache Kafka odcházejí a příchazejí. Na základě tohoto zjištění byl požadavek upraven na generování Avro schemat.

Avro schema je formát pro serializaci dat používáný v Apache Avro, což je framework sloužíćí k serializaci a deserializaci dat. Schéma popisuje strukturu dat a vychází z JSON formátu. Je nezávislý na programovacím jazyce a plattformě. Z pohledu generátoru se jedná pouze o reprezentaci dat - můžeme přepoužít znalosti z kapitoly Datové modely a z nich vyjít. Struktura modelování bude stejná, pouze výstup se bude lišit - výstupem bude Avro schema (soubor s suffixem .avsc) [27, 28].

Apache Kafka

Apache Kafka je distribuovaný systém pro streamování dat, který umožňuje komunikaci v reálném čase. Tento nástroj je úzce spojen s Event driven architekturou, která je v tuto chvíli velice populární ve spojitosti s mikroservisní architekturou. Hlavní entitou je Kafka topic, což je logická skupina zpráv.

Aplikace může zvolený topic konzumovat nebo produkovat. Pokud topic konzumuje, pak čeká na novou zprávu a při příchodu nové zprávy do taného topicu reaguje akcí. Pokud daný topic produkuje, tak na základě logiky aplikace přidává nové zprávy do topicu [29], což lze vidět na obrázku 2.8.



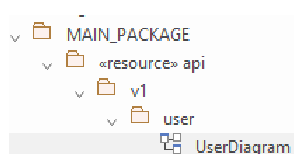
Obrázek 2.8: Využití Avro schemat s Kafkou [4]

2.5 Již zavedené způsoby modelování

Jelikož v několika společnostech se již používá generování, tak můžeme vycházet z historických dat, jakým způsobem modelují. V těchto společnostech se používá jednoúčelný skript pro daný způsob modelování, tedy tento skript se bez modifikace nedá použít jinde. Konkrétní názvy firem nemohu jmenovat, proto zde budou uvedeny jako společnost A a společnost B.

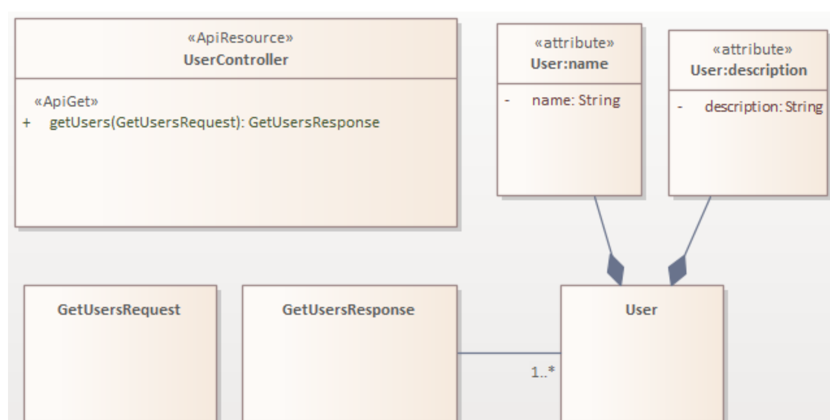
2.5.1 Model ze společnosti A

V první společnosti je model jednoduchý, nepodporují se path parametry, query parametry ani headery. API má definovanou pouze cestu, request a response objekt. Cesta je tvořena dle struktury složek, ve které se nachází daný diagram s objektem (obrázek 2.9). Dále je definovaný objekt se stereotype



Obrázek 2.9: Cesta API ve společnosti A

ApiResource, který obsahuje jednotlivé endpointy jako operations oddělené podle stereotype. Ty pak mají v sobě název Request a Response objektu (obrázek 2.10).

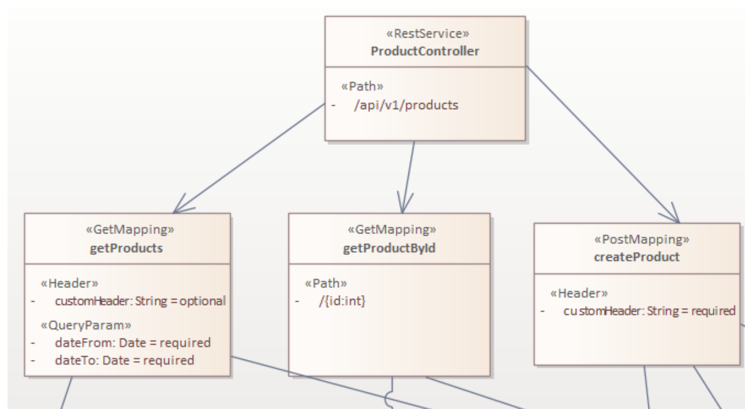


Obrázek 2.10: API resource ve společnosti A

Vnořené objekty jsou spojeny přes Asociaci s multiplicitou a primitivní datové typy jsou spojeny přes kompozici a mají stereotype attribute.

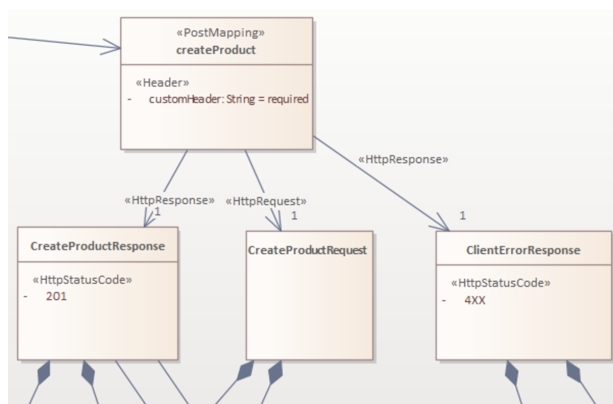
2.5.2 Model ze společnosti B

Druhá společnost vycházela ze způsobů ze společnosti A. Nicméně pro potřeby query parametrů, headerů, path parametrů a více typů HTTP response na základě statového kódu musel být model výrazně modifikován, z toho důvodu je celkově komplexnější. Cesta zde byla zjednodušena a je uvedena v Cont-



Obrázek 2.11: API resource ve společnosti B

rolleru jako atribut se stereotype path. Zbytek cesty je opět v jednotlivých endpointech jako atribut se stejnou stereotype jako Controller. Controller je značen stereotype RestService a s jednotlivými endpointy je provázan Asociací se směrem. Každý endpoint má stereotype podle své HTTP metody (GetMapping, PostMapping, ... - inspirováno SpringBoot anotacemi). V objektu endpointu lze definovat podle stereotype Headers, QueryParametry. PathParametry jsou v cestě evidováno dle složených závorek ve formátu názvu, oddělené dvojtečkou a datovým typem (obrázek diagramu 2.11). End-



Obrázek 2.12: Request a Response objekty API ve společnosti B

point je s objektem spojen opět Asociací se směrem a obsahuje stereotype HttpRequest nebo HttpResponse. Objekt s HttpResponse navíc obsahuje atributy vyjadřující k jakému HTTP stavu se vztahují (na obrázku 2.12).

2.6 Funkční požadavky

Byli vydefinovány tyto funkční požadavky, aby framework generátoru uměl následující věci.

- Konfigurovatelnost
  - Možnost si dopsat vlastní mapování z EA databáze
  - Možnost si dopsat vlasatní generátor souborů
  - Aplikace přijímá vstupní parametry
- Generování
  - Aplikace umí generovat Java modely a controllery pro SpringBoot
  - Aplikace umí generovat swagger specifikace
  - Aplikace umí generovat Avro schemata

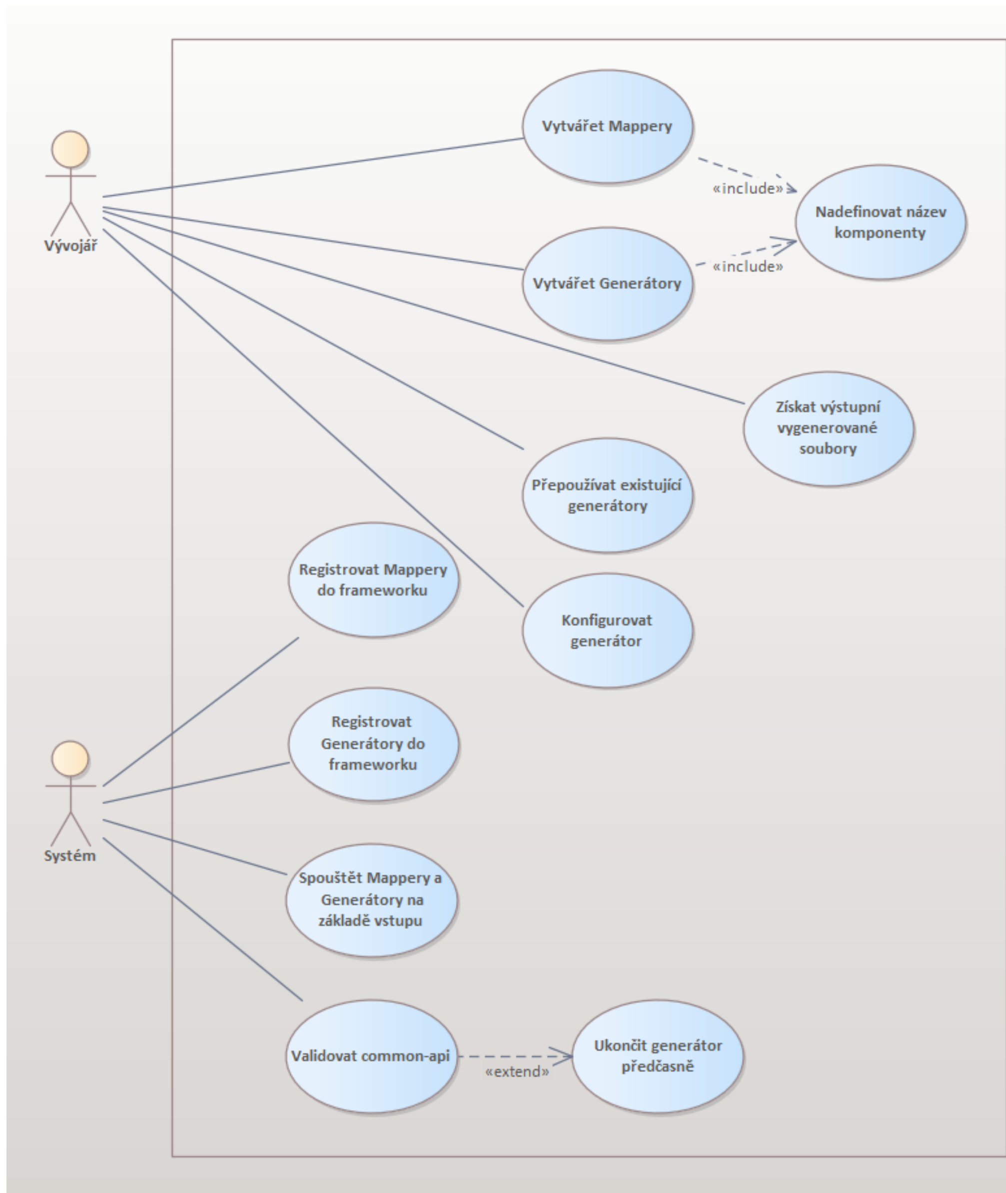
2.7 Nefunkční požadavky

Na základě funkčních požadavků byli vydefinovány tyto nefukční požadavky.

- Dohledávání chyb
  - Aplikace musí být schopna upozornit na chybovost modelu v EA
  - Aplikace musí validovat common-api
- Dokumentace
  - Aplikace obsahuje technickou dokumentaci pro uživatele

2.8 Use case

Use case diagram obsahuje pouze 2 aktéry: vývojáře (uživatele frameworku) a samotný systém. Z diagramu lze vidět, že aktér, vývojář, pouze definuje pravidla (generátory a mappery), zatímco o veškerou funkcionalitu se stará systém, který je spouští. Diagram lze vidět níže na obrázku 2.13.



Obrázek 2.13: Use case diagram pro výsledný framework na generování

Kapitola 3

Návrh

3.1 Volba techstacku

Pro následnou implementaci byl zvolen následující techstack. Jako programovací jazyk bude použita Java, konkrétně verze 17 a OpenJDK, což je v tuto chvíli nejnovější LTS verze. Pro project management bude využíván Maven. V aplikaci poté celý framework generátoru poběží na SpringBootu 3 a Springu 6. Pro redukování boilerplate kódu bude použita knihovna Lombok. Pro tvorbu souborů budou použity Freemarker šablony. Pro práci s databází EA bude použit Hibernate.

Tento TechStack byl zvolen primárně, protože potenciální uživatelé, kteří budou rozšiřovat framework jsou Java vývojaři. Díky této volbě TechStacku podpoříme a urychlíme adopci výsledného nástroje. Celý seznam technologií je v tabulce 3.1.

Technologie/knihovna	Verze	Oficiální dokumentace
Java	17LTS	https://openjdk.org/projects/jdk/17/
Maven	3.8.7	https://maven.apache.org
SpringBoot	3.0.5	https://spring.io/
Lombok	1.18.26	https://projectlombok.org/
Freemarker	2.3.32	https://freemarker.apache.org/
Hibernate	– (in SpringBoot)	–

Tabulka 3.1: Souhrn techstacku a verzí.

3.2 Architektura frameworku

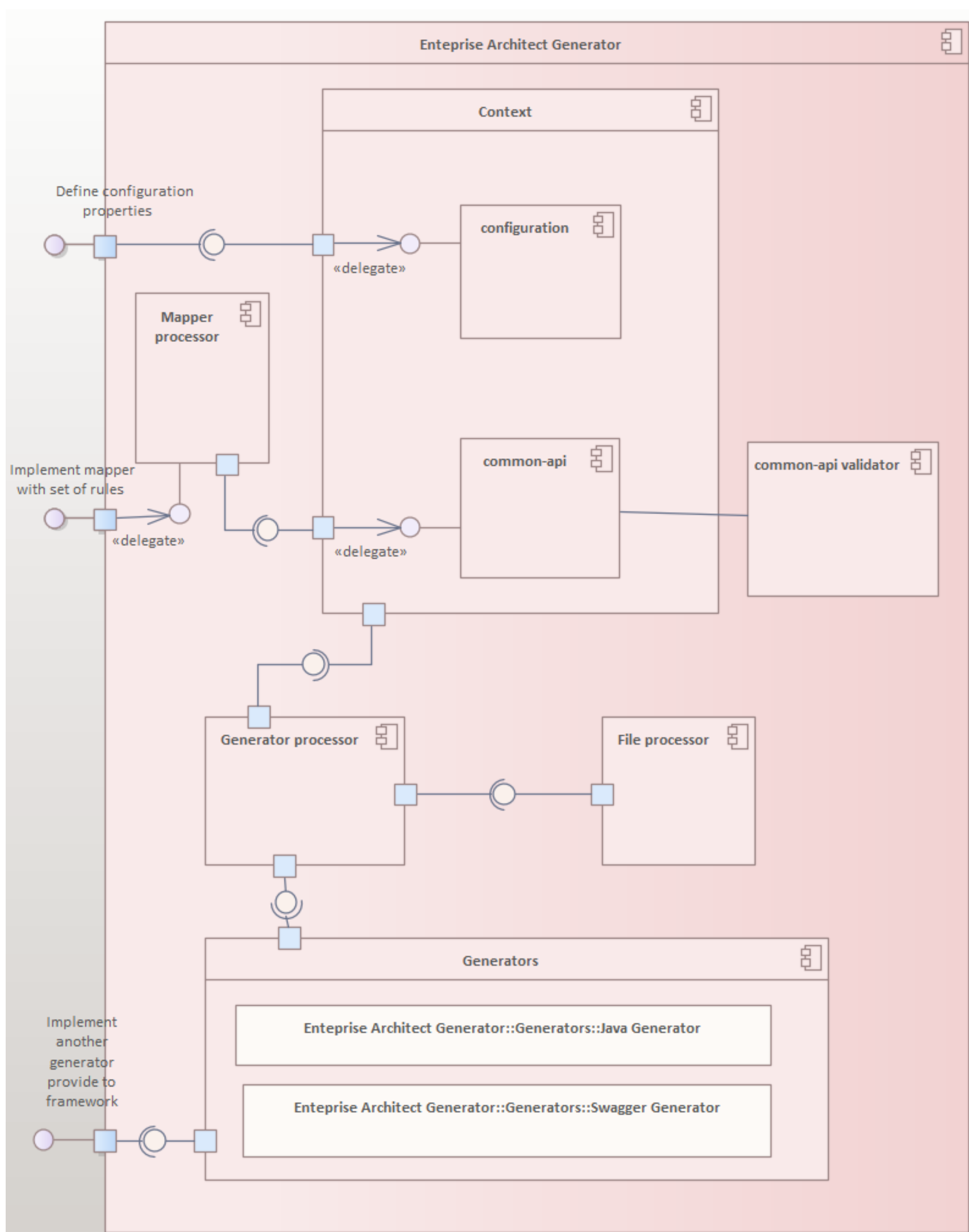
Framework rozdělíme do několika částí. Tou hlavní bude kontext frameworku, který má za cíl uchovávat common-api a konfigurační parametry ze vstupu aplikace. Common-api bude obecný model API endpointů, DTO Objektů a dalších, do kterého bude uživatel mapovat logiku.

Další částí frameworku je "mapper processor". K němu patří i interface poskytnuté uživateli frameworku. Uživatel skrze interface doplní mapovací logiku a processor se postará o to, aby se model namapoval do common-api a vložil se do kontextu frameworku.

Úzce související komponentou s "mapper processorem"a common-api je validátor common-api. Jeho hlavní úlohou je zalogovat chybějící části common-api. Příčinou těchto chyb může být špatně naimplementovaný mapper uživatele nebo špatně vymodelovaný diagram v EA. Díky validátoru bude pro uživatele hledání chyb snadnější.

Pak se zde nachází "generator processor", který po zmapování přijme kontext. Z kontextu použije konfigurační parametry, které taky obsahují množinu generátorů, které se spustí včetně jejich nastavení. Každý z těchto generátorů spustí a předá jim kontext, především kvůli common-api, ze které budou generovat soubory. "Generator processor"bude mimo jiné obsahovat interface, aby bylo možné pro uživatele si dopsat své vlastní další generátory.

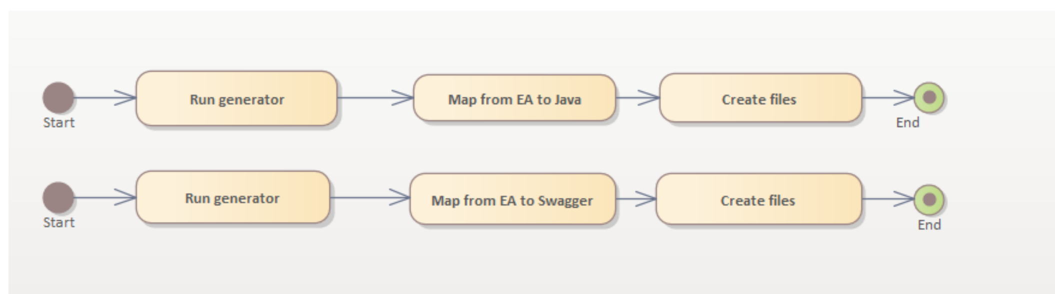
Na závěr tu je "file processor", který se postará o vytvoření vygenerovaných souborů do složky. Tím práce generátoru končí. Architektura celého frameworku je vyobrazena na obrázku komponent diagramu 3.1.



Obrázek 3.1: Komponent diagram navrhovaného frameworku

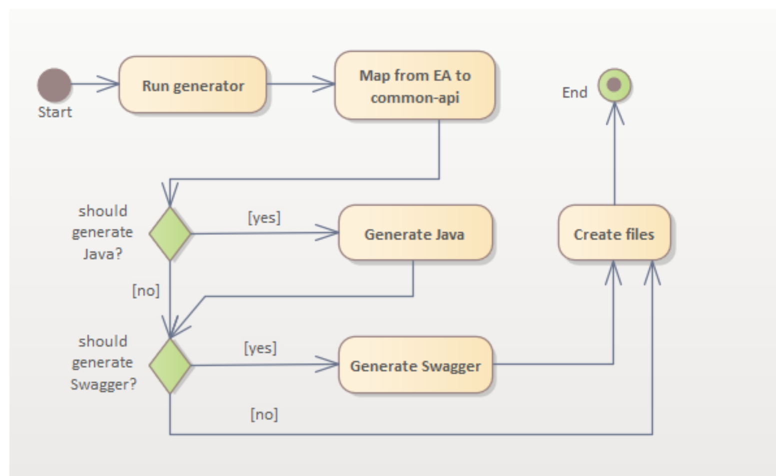
3.2.1 Common-api

Common-api vznikla jako potřeba pro mezivrstvu mezi mapováním z EA a generováním výsledných souborů. Tato mezivrstva vznikla z důvodu, aby konkrétní generátory byly odstíněny od mapování z EA a byly nezávislé. To znamená, že uživatel frameworku pouze tvoří mapování z EA databáze do common-api a nemusí tvořit již logiku.



Obrázek 3.2: Activity diagram jednoúčelných generovacích skriptů

Na diagramu 3.2 je názorná ukázka, jak probíhají základní generovací skripty. Pro každý výstup existoval skript nebo několik skriptů, které byly aplikovatelné pouze pro jednu společnost, pro které byly šity na míru.

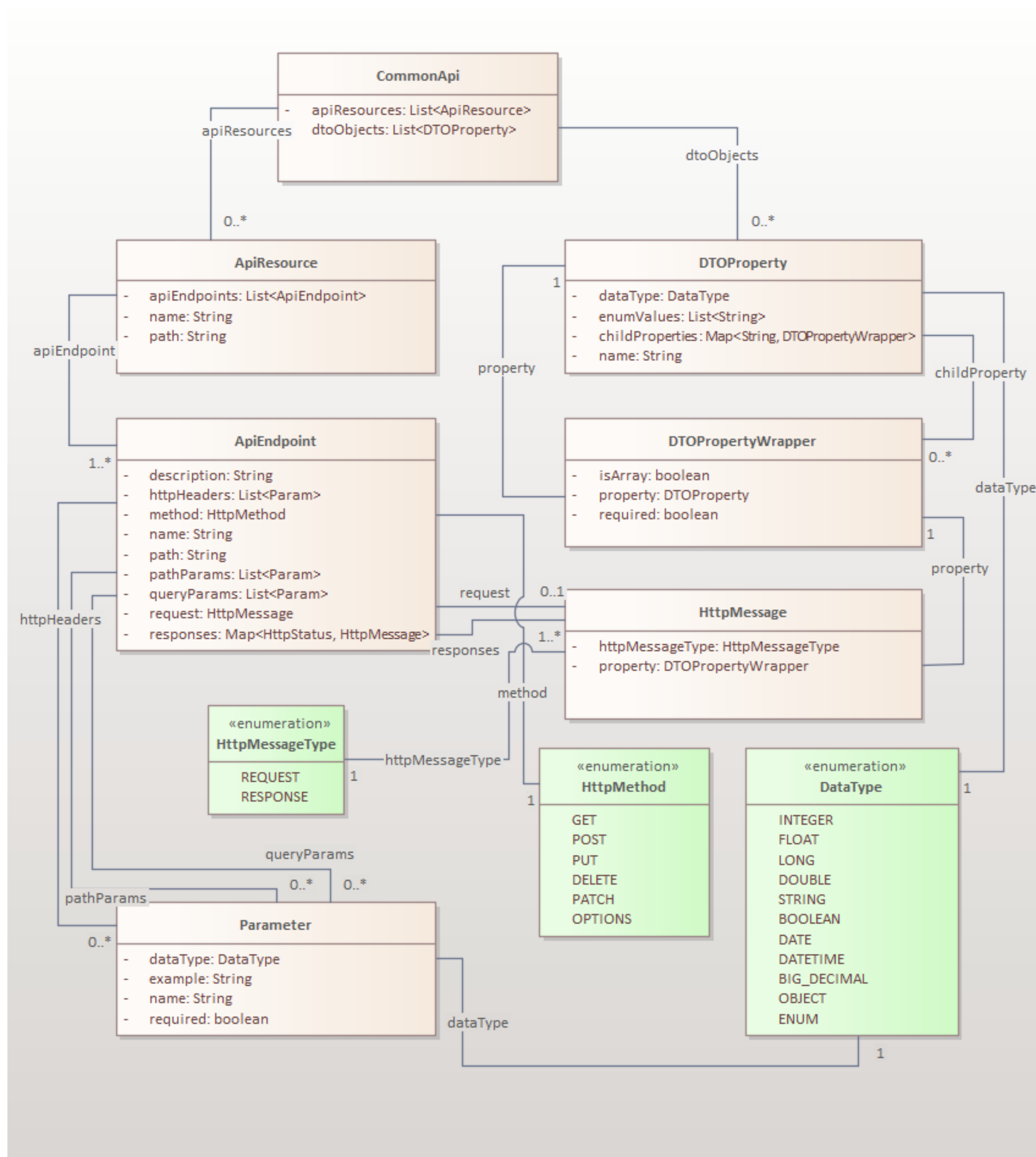


Obrázek 3.3: Activity diagram generovacího frameworku s common-api

Naopak na diagramu 3.3 je znázorněno, jak lze přepoužívat common-api. Na vstupu lze pustit více generovacích profilů a všechny proběhnou. V případě potřeby, přepoužití generátoru v jiné společnosti, je jen potřeba upravit mappery z EA do common-api. Vše z common-api dál bude navždy funkční.

Model common-api

Na obrázku 3.4 níže je vydefinován model Common-api, který pokrývá všechny požadavky, které vyšly z analýzy.

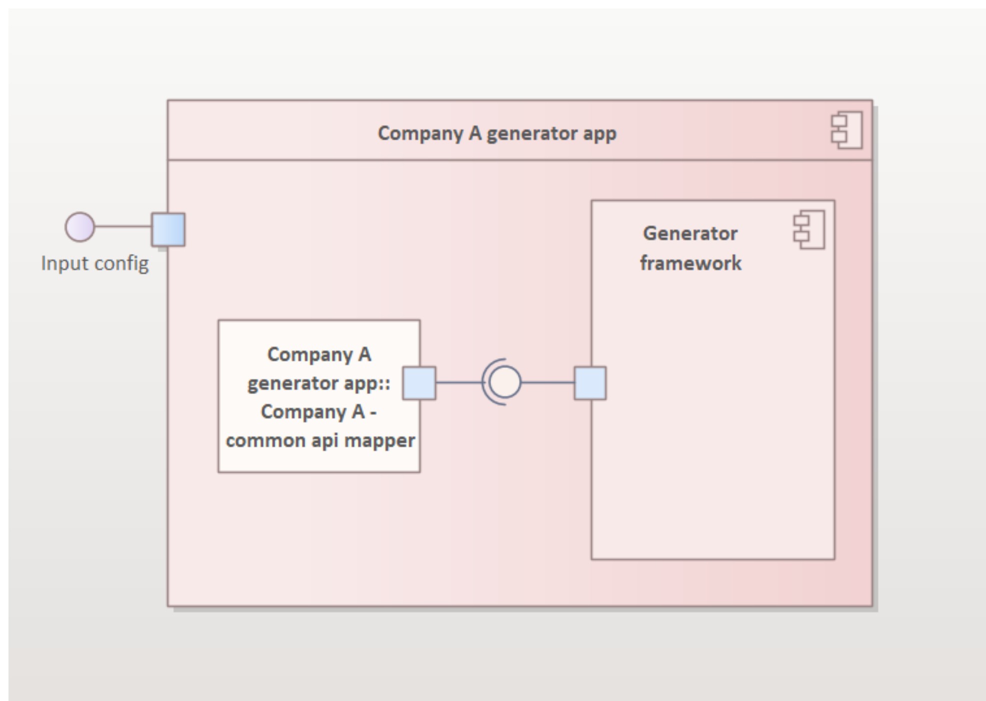


Obrázek 3.4: Model struktury common-api

3.2.2 Mapování z EA do common-api

Pro účely generovacího frameworku je nutné vymyslet mechanismus, jak jednoduše umožnit uživateli si nastavit vlastní mappery, nebo více mapperů a poté si vybrat, s jakým se generátor spustí. Jelikož jsme si již v předchozí kapitole vydefinovali TechStack a budeme používat SpringBoot, tak můžeme využít jeho přednosti - tou je Dependency injection (DI) a Inversion of Control (IoC).

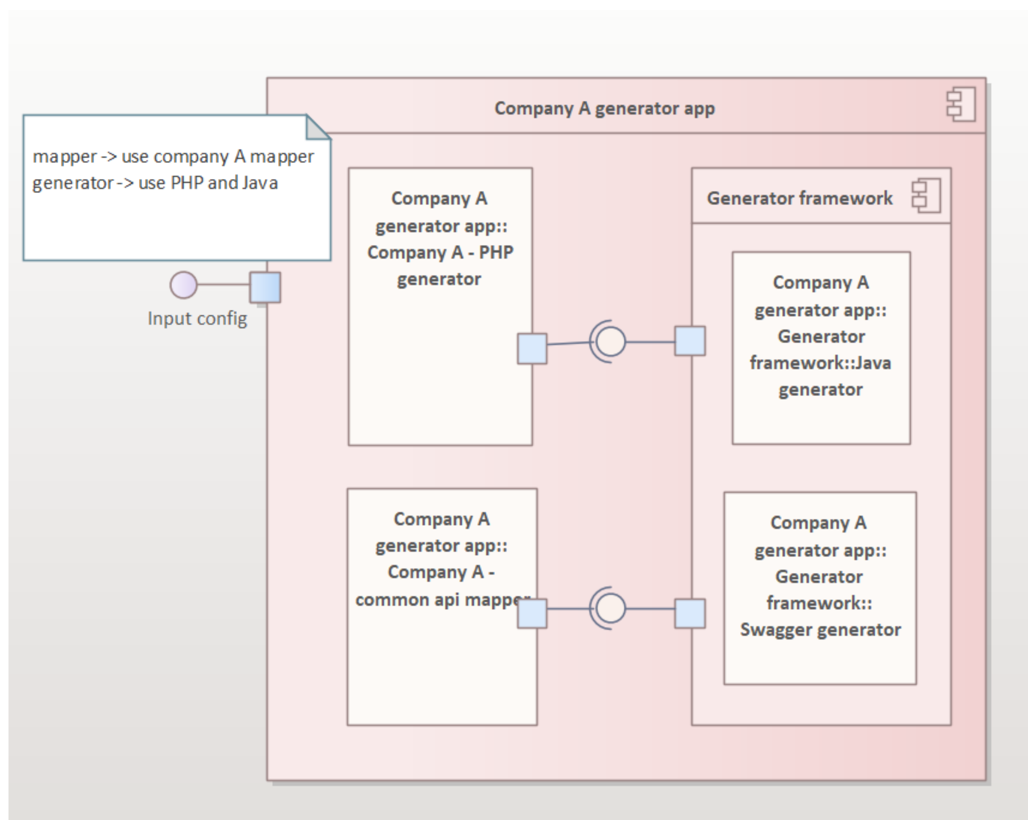
Generovací framework bude mít své vlastní anotace, které řeknou, že tato třída a její metody jsou mapovací logikou z EA do common-api. Aplikace k těmto třídám s danou anotací přistoupí za běhu programu pomocí Java reflexí. Ještě před tím, než je spustí, je třeba dle vstupních parametrů si vytvořit Spring context - nastavit všechny potřebné Beany, jako třeba připojení do databáze a další. Poté, co se tato inicializace dokončí, tak by se měl mapper spustit. Jakmile mapper doběhne, tak framework doplní výsledná data (common-api) do Spring contextu, aby generátor mohl začít pracovat na generování. Mechanismus za anotacemi je na diagramu 3.5.



Obrázek 3.5: Komponent diagram mapování do common-api

3.2.3 Generování souborů z common-api

Generování souborů jako takových, mají na starosti samostatné implementace generátorů. Proto, když si uživatel bude psát vlastní generátory, je zcela na něm, jakým způsobem export do souborů zvolí. Tvorba vlastních generátorů probíhá obdobně jako tvorba mapperů - stačí použít anotaci a framework si je přes reflexe dotáhne do svého kontextu a dokáže s nimi pracovat. Mechanismus zpracování jednotlivých generátorů můžete vidět na diagramu 3.6.



Obrázek 3.6: Komponent diagram tvorby a spouštění generátorů

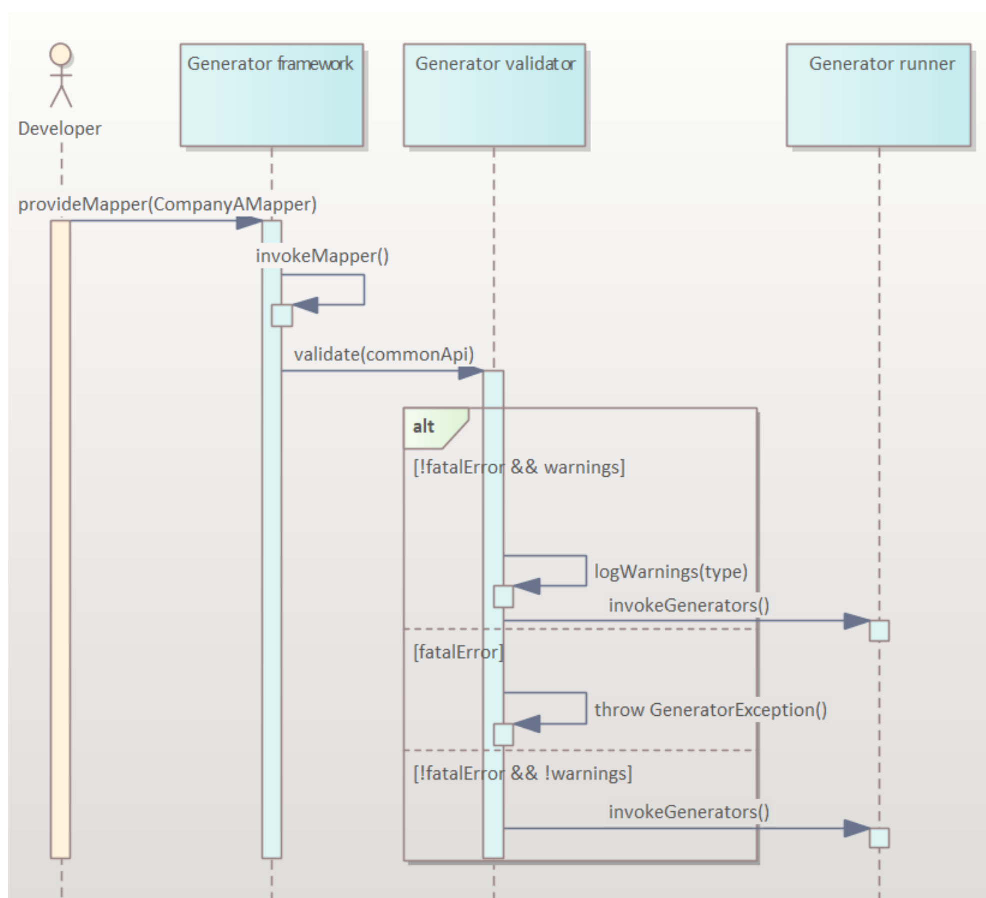
Co se týče generátorů, které budou již ve frameworku zahrnuty, tak ty používají template engine - Freemarker [30]. Díky tomu stačí nadefinovat šablonu a proměnné, které do šablony doplníme a vytvoříme nový soubor. To razantně zjednodušuje tvorbu souborů.

Pak již stačí na vstupu aplikace vydefinovat právě jeden mapper, který bude mapovat do common-api a seznam generátorů, které chceme, aby nad common-api proběhly.

3.2.4 Validátor common-api

Při MDD přístupu se budou při generování vyskytovat chyby v mapperu jen z počátku vývoje, kde se poladí a poté se budou vyskytovat jen vyjímečně. To ale neplatí u modelů jako takových, kde generátor nemůže zaručit, že dostane správná data. Ať už se jedná o špatně nakonfigurovaný generátor, nebo model neodpovídá metodice mapperu, která se ve firmě vydefinovala. Všechny tyto chyby můžou vést k chybové hlášce na straně generátoru.

Chybám nelze zabránit, ale lze logovat hlášky, které mohou alespoň částečně navést uživatele k chybě, kterou při modelování udělal. Z toho důvodu se v generovacím frameworku nachází jednoduchá třída, která kontroluje základní parametry common-api a pokud něco nesedí, tak zaloguje onu informaci. Pokud se jedná o závažnou chybu, tak ukončí celý generátor. Validátor tedy neeliminuje chyby, ale pomáhá je uživateli najít. Detailní fungování validátoru je vyobrazeno na sekvenčním diagramu 3.7.



Obrázek 3.7: Sekvenční diagram validátoru common-api

3.3 Konfigurovatelnost frameworku

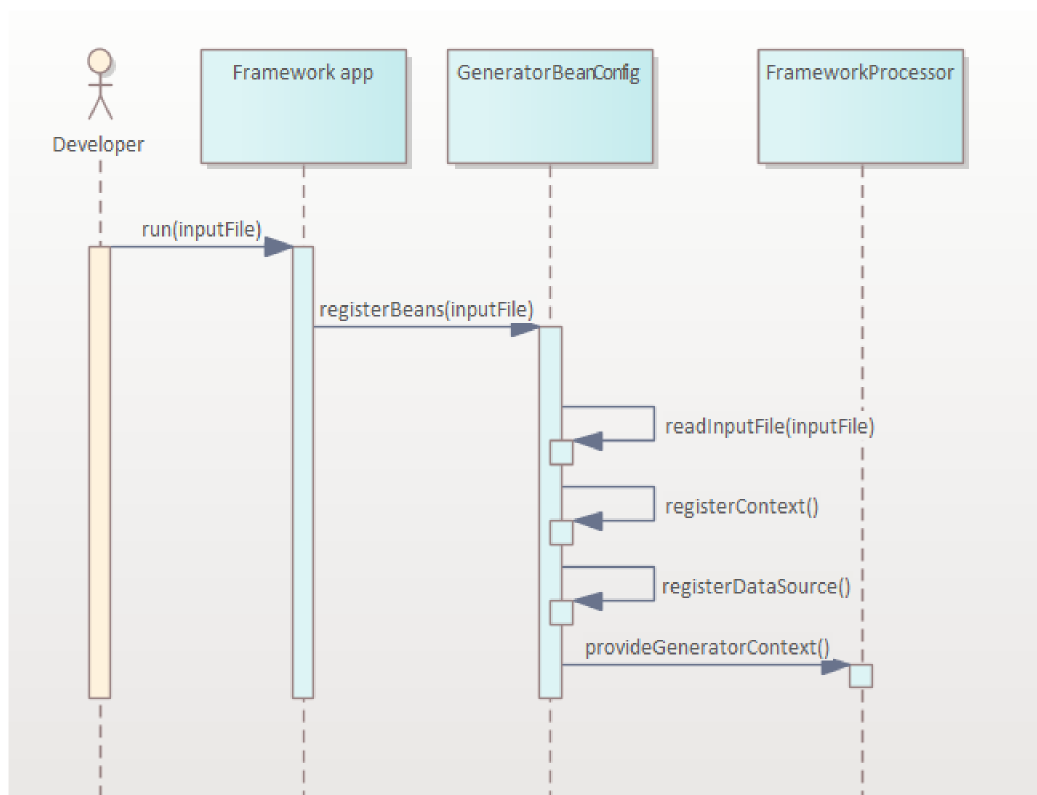
Framework jako takový bude přijímat na vstupu parametry ke konfiguraci, pak také parametry pro připojení k databázi EA - jdbc url, uživatele a heslo. Dále bude přijímat parametr s volbou mapperu, který bude aplikován nad generátorem. Dalším parametrem bude list generátorů, které chce uživatel nad common-api spustit. Pak další důležitou položkou, která nesmí chybět je počáteční složka v EA, kde se cílové diagramy nachází (generovat nad celou databází by bylo neefektivní). Na závěr zbývají jen parametry, které se vztahují ke konkrétním generátorům.

Parametr	**Popis**	**Povinnost**
DATABASE_URL	JDBC adresa pro připojení k EA databázi	✓
DATABASE_USER	Credentials pro připojení k EA databázi	✓
DATABASE_PASSWORD	Credentials pro připojení k EA databázi	✓
MAPPER_NAME	Název aplikovaného mapperu do common-api	✓
ENABLED_GENERATORS	Seznam generátorů ke spuštění.	✓
EA_START_PACKAGE	Složka v EA, odkud procházet modely	✓
OTHER_PARAMETERS	Ostatní parametry vztahující se ke konkrétním generátorům	X

Tabulka 3.2: Souhrn vstupních parametrů generátoru

Pokud by parametrů bylo pár, tak by bohatě stačilo dát jako jednotlivé argumenty při spouštění aplikace. Jelikož bude parametrů velké množství, tak bude pro uživatele příjemnější si vytvořit konfigurační soubor a předat aplikaci jediný argument - název s cestou konfiguračního souboru. Soubor bude ve formátu JSON - tento formát zná téměř každý a lze z něj snadno mapovat přímo do objektů v daném programovacím jazyce.

Jak funguje získávání vstupních parametrů a jejich předání do aplikačního kontextu je vyobrazeno na sekvenčním diagramu 3.8.



Obrázek 3.8: Sekvenční diagram konfigurace dle vstupních parametrů

Některé z těchto vstupních parametrů budou předány do kontextu generátoru. Kontext generátor je třída, kde budou uloženy všechny tyto atributy jako Beana a bude možné si je kdykoliv dotáhnout v jakékoliv Spring komponentě. Dalším typy vstupních parametrů slouží k nadefinování důležitých Bean pro spuštění Springu jako takového. Konkrétním příkladem může být definice DataSource pro databázové připojení.

Kapitola 4

Implementace

V této kapitole se budeme zabývat implementací. Vysvětlíme si jednotlivé části generátoru, jakým způsobem byly naimplementovány a jaké konkrétní metody byly použity.

4.1 SpringBoot starter knihovna

Cílem je dodat uživateli framework, který se co nejjednodušeji používá a rozšiřuje. Framework je koncipován jako starter knihovna, která obsahuje autokonfiguraci. Nejprve si vysvětlíme princip autokonfigurací. Autokonfigurační knihovna je speciální knihovna, která pří startu SpringBoot aplikace nastaví konfigurace pro různé části aplikace. V našem případě budeme používat k tomu, abychom uživateli do aplikace přidali Beany související s generátorem. Celý mechanismus využívá reflexí a anotací Spring frameworku, který skenuje jednotlivé třídy. Konfigurace, které chceme použít jako autokonfigurační, musíme přidat do souboru se jménem spring.factories v resources/META-INF. Ukázku autokonfiurační třídy s jednoduchou Beanou lze vidět na Listingu 4.1.

```
1  @AutoConfiguration
2  public class MyAutoconfiguration {
3      @Bean
4      public DataSource dataSource() {
5          //logic here
6      }
7  }
```

Listing 4.1: Ukázka autokonfigurační třídy

4.2 Implementace frameworku

V této části kapitoly bude popsána konkrétní implementace frameworku do detailu s ukázkami důležitých částí kódu a vysvětlením, proč to takto bylo řešeno.

4.2.1 Konfigurační soubor

Jako úplně první, co musíme udělat je připravit datový model pro vstupní parametry. Z návrhu jsme si vydefinovali, které parametry budeme potřebovat a taky to, že budou uloženy v souboru ve formátu JSON.

Aplikace bude přijímat jako argument cestu k danému konfiguračnímu souboru, díky čemu zajistíme, že se aplikace nemusí buildit při změně konfiguračních parametrů. Tento přístup zrychlí poté celou pipeline, protože nebude muset probíhat rebuild.

Dále si musíme v aplikaci přidat Java modely, abychom byli schopni namapovat z formátu JSON přímo do nich. Ty pak namapujeme přes ObjectMapper, což je třída z knihovny jackson. Jak vypadá Java model, je ukázáno v Listingu 4.2.

```java
/**
 * A context model of the framework generator storing input
   parameters.
 */
@Data
public final class GeneratorConfiguration {

    private String eaStartPackage;
    private String version;

    private MappingConfiguration mappingConfiguration;

    private DatabaseConnection databaseConnection;

    private Set<String> enabledGenerators;

    private Map<String, Object> parameters;
}
```

Listing 4.2: Model konfiguračního souboru v Javě.

4.2.2 Inicializace kontextu

Dalším krokem je připravit všechny Beany, které bude generátor nebo uživatel potřebovat za běhu aplikace. Některé Beany budou vyžadovat parametry ze vstupního souboru a některé budou mít závislosti na jiných Beanách, proto je nutné si dát pozor na posloupnost tvorby Bean. Posloupnost tvorby Bean lze řešit několika způsoby, všechny jsou na Listingu 4.3, 4.4, 4.5.

```
1 @AutoConfigureAfter(value=A.class)
2 @AutoConfigureBefore(value=B.class)
3 public class AutoConfigurationClass
```

Listing 4.3: Posloupnost pomocí posloupnosti autokonfiguračních tříd

```
1 @Bean
2 @DependsOn("otherBean")
3 public MyBean myBean()
```

Listing 4.4: Posloupnost pomocí anotace DependsOn

```
1 @Bean
2 @Order(2)
3 public MyBean myBean()
```

Listing 4.5: Posloupnost pomocí anotace Order

V našem případě budou dvě autokonfigurační třídy - GeneratorProcessorAutoConfiguration a GeneratorRunnerAutoConfiguration. GeneratorProcessorAutoConfiguration musí být spuštěna první, což je zajištěno metodou z 4.3. Vytvoří kontext generátoru, DataSource pro připojení k databázi a validátor common-api.

GeneratorRunnerAutoConfiguration se musí spustit jako druhá v pořadí a potřebuje některé Beany z první autokonfigurace - lze řešit importem. Tato autokonfigurace vytvoří embedded generátory pro Javu, Swagger a Avro schema, včetně jejich dopomocných tříd.

Nastavení obou autokonfigurací v kódu vypadá následovně viz Listing 4.6.

```
1 @AutoConfiguration
2 @AutoConfigureAfter({FreeMarkerAutoConfiguration.class})
3 public class GeneratorProcessorAutoConfiguration {}
4
5 @AutoConfiguration
6 @AutoConfigureAfter({GeneratorProcessorAutoConfiguration.class})
7 @Import({GeneratorContext.class, FileProcessor.class})
8 @RequiredArgsConstructor
9 public class GeneratorRunnerAutoConfiguration {}
```

Listing 4.6: Nastavení autokonfigurací

Čtení dat ze vstupního souboru proběhne úplně první Beanou v GeneratorProcessorAutoConfiguration. Ta díky interface SpringBootu ApplicationArguments přistoupí k argumentům, které přečte a zapíše do kontextu. Ukázka tohoto kódu je v Listingu 4.7.

```
1   @Bean
2   public GeneratorContext generatorContext(Arguments args) {
3       if (args.getSourceArgs().length < 1) {
4           log.error("Path to input file is missing!");
5           System.exit(0);
6       }
7
8       var filePath = args.getSourceArgs()[0];
9       var ctx = new GeneratorContext();
10      ctx.setConfiguration(getConfig(filePath));
11
12      log.trace("GeneratorContext Bean has been created");
13      return context;
14  }
15
16  private GeneratorConfig getConfig(String filePath) {
17      var file = new File(filePath);
18      if (!file.exists()) {
19          System.exit(0);
20      }
21
22      var objMapper = new ObjectMapper();
23      return objMapper.readValue(file, GeneratorConfig.class);
24  }
```

Listing 4.7: Zápis vstupního souboru do kontextu generátoru

4.2.3 Spuštění frameworku

V této fázi máme pomocí autokonfigurace nastaveny všechny Beany. Nyní potřebujeme skanovat celou aplikaci a hledat třídu s anotací @Mapper. Aby se začla vykonávat logika přesně po inicializaci a uživatel nemusel přímo zavolat nějakou metodu frameworku, se nabízí využít interface Springu - ApplicationListener a to s událostí ApplicationReadyEvent. Celou logiku frameworku pak vykonává metoda v Listingu 4.8.

```
1  @Override
2  public void onApplicationEvent(ApplicationReadyEvent event) {
3
4      var enabledGens = generatorContext
5          .getConfiguration()
6          .getEnabledGenerators();
7
8      mapperProcessor.run();
9
10     if (enabledGens == null) {
11         throw new GeneratorException("No enabled generators!");
12     }
13
14     var genBeans = event.getApplicationContext()
15         .getBeansWithAnnotation(Generator.class);
16
17     var relevantBeans = genBeans.keySet().stream()
18             .map(genBeans::get)
19             .map(Object::getClass)
20             .map(clazz -> clazz.getAnnotation(Generator.class)
21             .filter(annot -> enabledGens.contains(annot.name()))
22             .toList();
23
24     if (CollectionUtils.isEmpty(relevantBeans)) {
25         throw new GeneratorException("No enabled generators!");
26     }
27
28     relevantBeans.forEach(relevantBean -> {
29         if (relevantBean instanceof GeneratorHandler) {
30             ((GeneratorHandler) relevantBean).run();
31         } else {
32             throw new GeneratorException("Unknown instance!");
33         }
34     });
35 }
```

Listing 4.8: Exekuce frameworku pro generování

4.2.4 Nadefinování API pro uživatele

Přímo pro uživatele bude vystaveno několik interfaců, anotací a tříd s dopomocnou logikou, aby práce s frameworkem byla co nejlepší. Veškeré utility vystavené pro uživatele jsou do detailu popsány v JavaDocu.

Mapper

Pro tvorbu vlastních mapperů je vytvořena anotace @Mapper, která musí obsahovat parametr s jeho názvem. Anotace může být pouze nad třídou. Ta samá třída musí zároveň implementovat interface MapperHandler, a doplnit logiku do jejich metod, se kterými dále bude pracovat framework. Ukázka, jak vypadá naimplementování mapperu je v Listingu 4.9 a detail interfaců v Listingu 4.10. Mapper si uživatel musí vždy naimplementovat sám (narozdíl od generátorů).

```
1  import com.mmasata.eagenerator.MapperHandler;
2  import com.mmasata.eagenerator.annotations.Mapper;
3
4  @Mapper(name = "my-mapper")
5  public class MyMapper implements MapperHandler {
6
7      @Override
8      public List<ApiResource> mapApiResources() {
9          //implement mapping to common-api ApiResources here
10     }
11
12     @Override
13     public List<DTOProperty> mapDtoObjects() {
14         //implement mapping to common-api DTOProperties here
15     }
16 }
```

Listing 4.9: Ukázka implementace vlastního mapperu

```
1  @Documented
2  @Target(ElementType.TYPE)
3  @Retention(RetentionPolicy.RUNTIME)
4  @Component
5  public @interface Mapper {
6      String name() default "";
7  }
8
9  public interface MapperHandler {
10
11     List<ApiResource> mapApiResources();
12     List<DTOProperty> mapDtoObjects();
13 }
```

Listing 4.10: Interfacy mapperu

Další věc, která se musí v mapperu řešit je připojení do databáze EA. V aplikaci je nadefinována Bean objektu DataSource. Uživatel může použít jakoukoliv databázovou technologii. V případě, že bude chtít použít JPA, pak framework již disponuje entitami. JPA Entity se musí implicitně zapnout přes anotaci @EnableGeneratorJpaEntities, která zapne skenování nad složkou, kde jsou umístěny. Na následujícím Listingu 4.11 je vidět použití této anotace nad hlavní třídou aplikace. Detail tohoto interface lze vidět v ukázce 4.12

```
1 @Documented
2 @Target(ElementType.TYPE)
3 @Retention(RetentionPolicy.RUNTIME)
4 @EntityScan("com.mmasata.eagenerator.database.entity")
5 public @interface EnableGeneratorJpaEntities {
6 }
```

Listing 4.11: Zapnutí skenu JPA entit generátoru

```
1 @SpringBootApplication
2 @EnableGeneratorJpaEntities
3 public class MyApplication {
4
5     public static void main(String[] args) {
6         SpringApplication.run(MyApplication.class, args);
7     }
8
9 }
```

Listing 4.12: Detail anotace ke skenování JPA entit

Generátor

Implementace vlastních generátorů má obdobnou logiku jako mappery - pro uživatele je připravena anotace @Generator, která musí obsahovat parametr s jeho názvem. Anotace může být také pouze nad třídou a musí v tomto případě vždy implementovat GeneratorHandler. Ukázka vlastního generátoru je vidět na Listingu 4.13 a detail interfaců poté na Listingu 4.14.

```
import com.mmasata.eagenerator.GeneratorHandler;
import com.mmasata.eagenerator.annotations.Generator;
import com.mmasata.eagenerator.processor.FileProcessor;
import lombok.RequiredArgsConstructor;

import java.io.FileWriter;
import java.io.StringWriter;
import java.util.Map;

@Generator(name = "my-generator")
@RequiredArgsConstructor
public class MyGenerator implements GeneratorHandler {

    private final FileProcessor fileProcessor;

    @Override
    public void run() {
        //implement generator logic here
        Writer fileData = new StringWriter();
        fileProcessor.generate("myFile.txt", fileData);
    }
}
```

Listing 4.13: Ukázka implementace vlastního generátoru

```
@Documented
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Component
public @interface Generator {
    String name() default "";
}

public interface GeneratorHandler {
    void run();
}
```

Listing 4.14: Interfacy generátoru

Na Listingu 4.13 můžeme vidět třídu FileProcessor. Jejím účelem je pomoci uživateli s vytvořením souboru a zapsáním dat do něj. Obsahuje také několik metod, které pracují přímo s freemarker knihovnou, takže si uživatel může připravit svoje šablony a pracovat s nimi skrze tuto třídu. FileProcessor je Beana, stačí použít @Autowired a pomocí dependency injection ji využít v jakékoliv své komponentě.

4.3 Vestavěné generátory

Tato kapitola se bude zabývat implementací vestavěných generátorů. Dle funkčních požadavků aplikace musí v sobě již obsahovat generátory pro Javu, Swagger a Avro schema. Ty jsou součástí frameworku a uživatel je může kdykoliv použít. Všechny z uvedených vestavěných generátorů jsou naimplementované přes anotaci @Generator.

4.3.1 Java generátor

Tento generátor produkuje modely a interface pro REST controllery pro Spring. Je pomerně parametrizovatelný - lze si zvolit zda generovat modely jako třídy s klasickým gettery a settery nebo zda generovat už Java recordy, které jsou podporované od verze Javy 14 [31]. Další zásadní věcí je možnost si zvolit typ controllerů - standartní nebo reaktivní přístup. Reaktivní přístup obaluje response objekty Mono či Flux (využívá knihovnu reactor) [32]. Výhoda reaktivního přístupu je, že vše probíhá non-blocking, takže aplikace je efektivnější, nevýhodou je náročnost implementace, která je vyšší. V tabulce 4.1 níže je uveden seznam všech parametrů, které je potřeba vyplnit do vstupního souboru, aby tento generátor proběhl úspěšně.

Název	Popis
java-spring	Název generátoru
javaDistributionManagementRepoId	ID distribučního repositáře
javaDistributionManagementRepoUrl	URL adresa distribučního repositáře
javaDtoType	Typ modelů (třídy s LOMBOK či RECORDS)
javaControllerType	Typ controllerů (REACTIVE či STANDARD)
javaVersion	Verze javy
javaGroupId	GroupId vygenerovaného projektu
javaProjectName	Název vygenerovaného projektu
javaPackage	Root package vygenerovaných java souborů

Tabulka 4.1: Seznam parametrů pro Java generátor

Výstupy z generátoru poté vypadají následovně: na obrázku 4.15 je datový model s Lombokem, poté na 4.16 se nachází jak vypadá ten stejný model přes recordy. Controllery poté vypadají reaktivně takto viz. 4.17 a standartně viz. 4.18.

```
1 @Data
2 public class Product {
3     private String name;
4     private String id;
5     @Nullable
6     private String description;
7 }
```

Listing 4.15: Ukázka vygenerovaného modelu s Lombokem

```
1 public record Product(@Nullable String description,
2                                String name,
3                                String id){}
```

Listing 4.16: Ukázka vygenerovaného modelu jako recordy

```
1  @RestController
2  @RequestMapping(value = "/api/v1/products")
3  public interface ProductsControllerReactive {
4      // Override this method
5      @RequestMapping(method = RequestMethod.GET)
6      default ResponseEntity<Flux<Product>> getProducts() {
7          return new ResponseEntity<>(HttpStatus.NOT_IMPLEMENTED);
8      }
9
10     // Override this method
11     @RequestMapping(method = RequestMethod.GET, value = "/{id}")
12     default ResponseEntity<Mono<Product>> getProductById(
13             @PathVariable String id) {
14         return new ResponseEntity<>(HttpStatus.NOT_IMPLEMENTED);
15     }
16 }
```

Listing 4.17: Ukázka vygenerovaného reaktivního controlleru

```
1  @RestController
2  @RequestMapping(value = "/api/v1/products")
3  public interface ProductsControllerStandard {
4      // Override this method
5      @RequestMapping(method = RequestMethod.GET)
6      default ResponseEntity<List<Product>> getProducts() {
7          return new ResponseEntity<>(HttpStatus.NOT_IMPLEMENTED);
8      }
9
10     // Override this method
11     @RequestMapping(method = RequestMethod.GET, value = "/{id}")
12     default ResponseEntity<Product> getProductById(
13             @PathVariable String id) {
14         return new ResponseEntity<>(HttpStatus.NOT_IMPLEMENTED);
15     }
16 }
```

Listing 4.18: Ukázka vygenerovaného standartního controlleru

4.3.2 Swagger generátor

Druhým vestavěným generátorem je generátor Swaggeru, konkrétně generuje OpenApi specifikaci verze 3.0.3. Parametrem si lze pouze zvolit název celého dokumentu, seznam parametrů je uveden v tabulce 4.2 níže.

Název	Popis
swagger	Název generátoru
swaggerTitle	Název vygenerovaného swaggeru dokumentu

Tabulka 4.2: Seznam parametrů pro Swagger generátor

Výstupní soubor Swaggeru poté může vypadat takto - viz. Listing 4.19.

```
1  openapi: 3.0.3
2  info:
3    title: Swagger title of API
4    version: 1.0.0
5  paths:
6    /api/v1/products:
7      get:
8        operationId: getProducts
9        description: Return list of Products
10       responses:
11         200:
12           description: The request sent by the client was
     successful.
13           content:
14             application/json:
15               schema:
16                 type: array
17                 items:
18                   $ref: '#/components/schemas/Product'
19 components:
20   schemas:
21     Product:
22       type: object
23       required:
24         - name
25         - id
26       properties:
27         name:
28           type: string
29         id:
30           type: string
31         description:
32           type: string
```

Listing 4.19: Ukázka vygenerovaného Swaggeru

4.3.3 Avro schema generátor

Posledním vestavěným generátorem je generátor pro Avro schemata, ten nepotřebuje žádné parametry, pouze musí být v konfiguračním souboru zapsán název tohoto generátoru - viz. tabulka 4.3.

Název	Popis
avro-schema	Název generátoru

Tabulka 4.3: Seznam parametrů pro Avro schema generátor

Z pohledu konfigurace i implementace se jednalo o nejméně složitý generátor ze všech tří uvedených. Výstupní soubor ve formátu .avsc poté vypadá takto 4.20.

```
{
  "type": "record",
  "name": "Product",
  "fields": [
    {
      "name": "type",
      "type": "enum",
      "symbols": [
        "A",
        "B"
      ]
    },
    {
      "name": "subProducts",
      "type": {
        "type": "array",
        "items": {
          "type": "record",
          "name": "SubProduct",
          "fields": [
            {
              "name": "name",
              "type": "string"
            },
            {
              "name": "description",
              "type": "string"
            }
          ]
        }
      }
    }
  ]
}
```

Listing 4.20: Ukázka vygenerovaného Avro schematu

4.4 Použité best practices

Při vývoji celého frameworku jsem se řídil základními principy, díky kterým je kód čitelný, jednoduchý a dobře testovatelný.

Jedním z těchto principů je KISS, který říká, že kód by měl být co nejjednodušší a vývojář by se měl vyhýbat složitým návrhům. Tohoto principu jsem se během celé fáze vývoje držel a veřím, že kód je jednoduchý a přehledný. Lze demonstrovat na třídě GeneratorInitializrListener, která vykonává celou logiku frameworku, ale přesto je naprosto jasné, co se v ní děje již od pohledu [33].

Dalším principem, kterým jsem se během celé fáze řídil je DRY. Aplikace neobsahuje žádný duplicitní kód, v případě opakující se logiky byla vytvořena například Util třída [33].

Posledním důležitým principem, který bych rád uvedl je SoC. I přestože se jedná o framework, tak kód je mezi sebou provázán minimálně [34].

4.5 Použité vzory

Celý vývoj se opíral o základní principy, na kterých je postavený celý Spring framework. Konkrétně jde o reflexe, DI a IoC [35].

Reflexe je schopnost za běhu programu zkoumat interní vlastnosti programu. Je možné zkoumat třídy, jejich metadata, informace o metodách, nebo také přímo exekuovat nějakou metodu jen podle názvu nebo vlastností. Jedná se o unikátní funkci jazyka Java.

Spring framework je založený na DI a IoC. DI říká, že objekty se nestarají o svoje vytvoření a zaniknutí, ale místo toho jsou vloženy do objektu z vnějšku. IoC naopak znamená, že kontejner spravuje všechny komponenty aplikace a stará se o životný cyklus objektů a jeho vkládání do jiných objektů. Spring framework má vlastní IoC kontejner, jehož výhody jsem používal a tvorbou Bean delegoval tento problém na Spring [35].

4.6 Problémy při implementaci a řešení

V této kapitole zmíním všechny problémy, které jsem musel během implementace řešit a jakým způsobem jsem je nakonec vyřešil.

První problém nastal hned po úspěšném vygenerování prvních java artifaktů, kde se po přidání tohoto artifaktu do jiného projektu nechtěla aplikace vůbec spustit. Problém byl, že pro generování standartních a reaktivních controllerů bylo potřeba do pom.xml vydefinovat jiné závislosti. Tyto dvě závislosti se vzájemně vylučují, protože tvoří stejné Beany a jedna přepíše konfigurace té druhé. Jediným adekvatním řešením bylo umožnit generovat pouze standartní či pouze reaktivní controllery - vzniknou tedy 2 artifakty a každý tým využije jeden podle své potřeby.

Druhým velkým problémem bylo, že v mapperu jsme chtěli číst z databáze a proto bylo nutné použít anotaci @Transactional. Po přidání této anotace nemohl framework najít třídu s anotací @Mapper. To se dělo z toho důvodu, že anotace @Transactional používá Proxy pattern a nad danou mapovací třídou po přidání @Transactional vůbec anotace generátoru nebyla. V tuto chvíli jako nejjednodušší řešení se jevilo mít třídu mapperu a třídu s prácí s databází v jiné třídě. Do budoucích verzí bude nutné se zamyslet nad elegantnějším řešením.

Třetí problém, se kterým jsme se setkali není úplně tak problém generátoru, ale bylo potřeba ho vyřešit. Jedná se o to, že Avro schema nepodporuje pole Enumů. Jako řešení v tomto případě je využít pole Stringů nebo jiný datový typ, podporovaný Avrem.

Další problémy se nejvíce týkaly chybně připravených freemarker šablon, které se ladily, dokud výstupem nebyl validní kód nebo validní dokumentace dle dané syntaxe. Pak také spousta menších chyb, které byly většinou spojené se špatně nakresleným modelem v EA.

Mimo tyto problémy je nutné říci, že všechny knihovny, které v aplikaci jsou využívány, obsahují minimum chyb a jsou udržovány a pravidelně aktualizovány.

4.7 Využití aplikace v konkrétní společnosti

Na závěr kapitoly implementace bych rád zmínil, že framework se již aktivně používá. Konkrétně ve společnosti B, která byla již zmíněna v kapitole analýza. V tuto chvíli byl ve společnosti naimplementován konkrétní mapper a používají se tam vestavěné generátory pro Javu a pro Swaggery a fungují v pořádku. Generátor si zatím spouští vývojáři lokálně na svém zařízení, ale další krok je proces zautomatizovat přes CI/CD pipeline v aplikaci Jenkins. Na základě používání dalších vývojářů se již do Jira backlogu začali zakládat úkoly pro rozšíření generátorů. Zde uvádím alespoň úryvek mapperu, viz Listing 4.21.

```
1  @Mapper(name = "company-b-mapper")
2  @RequiredArgsConstructor
3  public class BMapper implements MapperHandler {
4      private final BApiResourcesMapper resourcesMapper;
5
6      @Override
7      public List<ApiResource> mapApiResources() {
8          return resourcesMapper.map();
9      }
10 }
11
12 @Service
13 @RequiredArgsController
14 public class BApiResourcesMapper {
15     private final GeneratorContext context;
16     private final BMapperHelper helper;
17
18     @Transactional(readOnly = true)
19     public List<ApiResource> map() {
20         var config = context.getConfiguration();
21         var startPackage = helper.findPackageByPath(config);
22         var classes = helper.findObjects(startPackage,CLASS);
23         var services = helper.getByStereotype(classes,SERVICES);
24
25         return services.stream()
26             .map(service -> {
27                 var attribute = helper.attrWithStereotype(
28                         service, PATH);
29                 var basePath = BUtils.getPath(attribute);
30                 var endpoints = helper.getConnector(
31                         service.sourceConns(), ASSOCIATION)
32                         .stream()
33                             .map(TConnector:getEndObject)
34                             .filter(Object::nonNull)
35                             .toList();
36                 return new ApiResource(basePath,
37                     service.name(),
38                     mapApiEndpoints(endpoints);
39             }).toList();
40     }
41 }
```

Listing 4.21: Úryvek kódu mapperu ve společnosti B

Úplně na začátku si mapper dotáhne konfiguraci z kontextu a jelikož je to Beana, tak je to velice snadné. Následně rekurzivně prochází složky, dokud nenajde cílovou složku. Nad cílovou složkou potom najde objekty, které jsou typu třídy a z nich poté ještě vyfiltruje ty, které jsou označeny stereotype, která definuje, že se jedná o API resource. Poté iteruje všechny API resource a ke každému najde API cestu a jeho endpointy, které jsou spojeny přes vazbu typu Asociace. Jednotlivé endpointy se musejí také namapovat do common-api, proto se volá metoda mapApiEndpoints().

Celá logika mapperu je daleko komplexnější, ale na tomto úryvku lze vidět, jak je jednoduchá práce s frameworkem a API, které nabízí.

Kapitola 5

Testování

V této kapitole se budeme zabývat tím, jakým způsobem se přistupovalo k testování frameworku a jaké konkrétní chyby byly nalezeny.

5.1 Způsob testování

Testování frameworku bylo poměrně náročné. Hlavní problém byl, že nebylo úplně jasné, co testovat. Jelikož se jedná o framework a každý uživatel si může dopsat jakoukoliv logiku chce, tak jaksi nedává smysl tyto části kódu testovat. Generátory, které už jsou součástí frameworku by teoreticky otestovat šly, ale opět je zde jeden problém - jak otestovat, že se soubory generují v pořádku, což opět může ovlivňovat špatně naimplementovaný mapper uživatelem.

Z toho vyplynulo, že v logice psanou uživatelem testy nebudou, ale bude do frameworku přidána validační komponenta common-api. Ta bude upozorňovat na nesouvislosti v datech namapovaných do common-api a bude logovat důležité informace pro uživatele. To samozřejmě nezakazuje uživateli udělat si vlastní unit test pro svůj mapper či generátor.

Jako další metodu pro ověření správnosti jsem používal build přes maven. Tím se snadno kontrolovalo, zda kód vygenerovaný z freemarker šablon je správný nebo ne (z pohledu daného programovacího jazyka). Tato metoda se velice osvědčila a pomohla mi opravit všechny nalezené chyby. Toto jsem

kontroloval manuálně, nebyl k tomu vytvořen žádný skript, i tak se jednalo o velice efektivní způsob testování.

Dále se využily uživatelské testy vývojářů, kde si vývojáři nakreslili modely a následně se sami snažili vygenerovat výstup. Na základě jejich postřehů bylo opraveno několik chyb a ve společnosti, kde je framework využíván jsou požadavky v Jira backlogu na další rozšiřování.

Na závěr nesmí chybět unit testy, které ovšem testují jen jádro frameworku, protože to je jediná komponenta, kterou neovlivňuje uživatel, a proto ji lze snadno testovat. Strategie testování jádra frameworku bude detailně rozebrána v následující kapitole na testy jádra frameworku.

5.2 Testy jádra frameworku

Jediné tři třídy, které dává smysl testovat unit testy jsou CommonApiValidator, MapperProcesor a GeneratorInitializrListener. Druhá a třetí třída fungují na stejném principu - přes reflexe si dotáhnou Beanu mapperu či generátoru a snaží se ji spustit. Unit testy jsou navrženy tak, že testují všechny možné chyby, které mohou nastat a kladný scénář. Seznam všech testů je uveden v tabulce 5.1 níže.

Testy třídy GeneratorInitializrListener
Test nenalezení žádného generátoru a vyhození správné výjimky Test o nevyužití předepsaného interface generátorem a vyhození správné výjimky Test úspěšného průchodu
Testy třídy MapperProcesor
Test nenalezení žádného mapperu a vyhození správné výjimky Test o nevyužití předepsaného interface mapperem a vyhození správné výjimky Test úspěšného průchodu
Testy třídy CommonApiValidator
Parametrizované testy možných kombinací ApiResource objektu Parametrizované testy možných kombinací ApiEndpoint objektu

Tabulka 5.1: Seznam unit testů

5.3 Uživatelské testy

Uživatelské testy probíhaly ve společnosti B, kde si vývojáři z týmu standartně vzali implementační úkol pro přidání nového endpointu či celého API resource. Strukturu API měli nejprve vymodelovat, vygenerovat a následně implementovat logiku do dané mikroslužby. Pro tento účel byl vydefinován jeden jednoduchý testovací scénář. Scénář vychází z toho, že pro generátor je již připraven mapper, který je dostupný na interním GIT.

5.3.1 Persona

Personou pro uživatelské testy je zkušený Java vývojář, který má zkušenosti z korporátního prostředí. Dále má alespoň základní znalost aplikace EA a má zkušenosti s designem REST API. Také by měl mít alespoň základní povědomí o MDD.

5.3.2 Testovací scénář

Uživatelé zkoušeli tento jednoduchý scénář na obrázku 5.1 níže.

Krok	Název	Popis	Podmínky	Kroky	Očekávaný výsledek
#1	Návrh REST API	Vymodelování API endpointu v EA	Uživatel má oprávnění zapisovat do EA společnosti	1. Otevřít aplikace EA 2. Vytvořit složku, kde bude umístěn diagram 3. Označit složku stereotype "ApiGenerator" 4. Nakreslit API endpoint do diagramu dle úkolu z Jira	1. Model je nakreslený v EA
#2	Příprava generátoru	Příprava generátoru na lokálním zařízení vývojáře	Projekt s implementovaným mapperem je připraven na GITu	1. Naklonovat si GIT repository s projektem 2. Nastavit konfigurační soubor s cestou na složku s nakresleným modelem v EA. 3. Nastavit informace pro připojení k DB do konfiguračního souboru 4. Nastavit správný mapper a generátor "java-spring" do konfiguračního souboru.	1. Generátor je připravený ke spuštění
#3	Spuštění generátoru	Vývojář spustí generátor na svém zařízerní	Hotové kroky 1 a 2	1. Spustit generátor s argumentem,který odpovídá umístění konfiguračního souboru	1. Generátor zaloguje o úspěšném připojení do databáze EA 2. Generátor zaloguje o nalezení mapperu a jeho spuštění 3. Generátor zaloguje o spuštění generátoru "java-spring" 4. Po skončení se vytvoří složka export s vygenerovanými soubory
#4	Deploy artifaktu	Vývojář vygenerovaný kód nahraje do Nexus repository	Byl vygenerován validní java kód	1. Přejít v terminálu do vygenerovaného projektu 2. Spustit příkaz "mvn clean deploy"	1. Deploy proběhne úspěšně 2. Po zkontrolování je artifakt v Nexus repository nahrán

Obrázek 5.1: Testovací scénář

5.3.3 Výsledky uživatelských testů

Dva uživatelé, kteří spadají do persony (kapitola 5.3.1) provedli jednoduchý scénář, který je vydefinován v kapitole 5.3.2. Ještě před začátkem uživatelského testu si ve svém Jira backlogu vzali implementační úlohu a namodelovali API endpoint v EA.

Do výsledných testů nebyly zahrnuty testy, které skončily chybou kvůli špatně nakresleným modelům v EA. V těchto případech se jednalo o vnější vliv, který generátor nemůže ovlivnit, proto je nebereme v potaz.

U obou proběhl testovací scénář úspěšně a vygeneroval se Java kód. Po vygenerování si ještě prošly jednotlivé vygenerované třídy, aby mohly dát zpětnou vazbu k vylepšení, co například chybí v modelech nebo, co lze udělat jinak v REST interface. Návrhy na vylepšení, které navrhly budou zmíněny v následující kapitole.

S generátorem jsou uživatelé spokojeni a hodlají jej nadále používat.

5.4 Nalezené chyby a návrhy na vylepšení

V této části se nachází seznam chyb, které jsem nalezl v průběhu vývoje. Také se zde nacházejí návrhy na vylepšení, které byly sesbírany na základě uživatelských testů. Některé byly již zapracovány a některé je nutné ještě dodělat, popřípade zanalyzovat, zda dané vylepšení dává smysl. Seznam všech připomínek a nápadů je na obrázku 5.2 níže.

Název	**Popis**
Názvy proměnné u HTTP headers	Standartní HTTP hlavičky jako Authorization, nebo Accept-Language vygenerují nekorektní název proměnné. Je nutné, aby výsledné proměnné byly převedeny do formátu CamelCase.
Chybí možnost přidat repositories a pluginRepositories do pom.xml	Do vygenerovaného pom.xml nelze nakonfigurovat repositories a pluginRepositories. V případě, že se deploy artefaktu spouští ve vnitřní síti společnosti, pak může potřebovat nasměrovat na jiná úložiště, než je standartní maven-central.
Umožnit v datových modelech dědičnost	Zamyslet se nad rozšířením common-api, aby uměla pracovat s dědičností. V případě rozšíření také rozšířit do vestavěných generátorů.
Zamyslet se nad změnou java REST interface	V tuto chvíli se controllery generují jako REST interface. Návrh spočívá v tom, že by se místo toho mohli generovat abstraktní třídy, je nutné se zamyslet nad výhodami a nevýhodami.
Podpora validačních anotací v Javě	Zamyslet se nad rozšířením Java modelů, aby podporovaly validační anotace knihovny jakarta (jako například @Min, @Max, @Range, @NotBlank).

Tabulka 5.2: Seznam chyb a návrhů na vylepšení

Kapitola 6

Závěr

6.1 Výsledky práce

Cílem této diplomové práce bylo vytvořit konfigurovatelný generátor kódu z Enterprise Architect. Výsledná práce obsahuje analáýzu, návrh, samotný kód, testy a dokumentaci. Nakonec se nejedná o pouhý generátor, ale na konfigurovatelnost byl opravdu kladen důraz, jedná se tedy spíše o framework, který je daleko více robustní než bylo původně zamýšleno. Výsledná aplikace kromě generování také loguje celý průběh pro snadnější hledání chyb v modelech. Framework je dále navržen tak, že jeho rozšiřování o další funkcionality bude pro vývojáře opravdu snadné a spoustu věcí již řeší za vývojáře.

6.2 Splnění cílů

V průběhu práce byl mírně poupraven jeden požadavek, jelikož bylo identifikováno, že je od generátoru potřeba mírně něco jiného. Konkrétně jde o generování Kafka Consumer/Producer, které bylo nahrazeno za Avro schemata, která slouží jako formát zpráv při komunikaci s Kafkou. V závěru práce lze již říci, že poupravení požadavku bylo zvoleno správně, protože Avro schema je obecnější a může být využito i mimo Kafku.

Na závěr lze o práci říci, že splnila všechny své požadavky, ba naopak disponuje i funkcemi navíc. Aplikace je poměrně robustní framework, do kterého snadno přidávat další generátory. Z již obsažených generátorů obsahuje navíc také generátor Swaggeru. Za velký úspěch práce považuji i to, že tento generátor je již používán v jedné společnosti a zatím tam bezchybně generuje všechny výstupy.

6.3 Další rozšiřitelnost a doporučení

Aplikaci mám v plánu dále do budoucna rozšiřovat. Hlavním dlouhodobým cílem je vymyslet způsob, jak udělat generický mapper, který pokryje základní požadavky, díky čemuž by uživatelé nemuseli dopisovat své mappery. Jako další velké téma k zamyšlení na rozšíření je udělat z této aplikace webovou službu a k tomu i uživatelské rozhraní. Třetím bodem k zamyšlení do budoucna je možnost snadné integrace s DevOps. Celkově z pohledu rozšiřitelnosti jsou zde velké možnosti, je i zároveň více směrů, kterými se aplikace může vydat.

Literatura

[1] “Modeling languages - concepts.” [Online]. Available: https://modeling-languages.com/clarifying-concepts-mbe-vs-mde-vs-mdd-vs-mda/

[2] “Javatpoint - uml entity relations.” [Online]. Available: https://www.javatpoint.com/uml-association-vs-aggregation-vs-composition

[3] “Good data - data model.” [Online]. Available: https://www.gooddata.com/blog/what-a-data-model/

[4] “Data flair - kafka schema registry.” [Online]. Available: https://data-flair.training/blogs/kafka-schema-registry/

[5] “Sparxea - history of versions.” [Online]. Available: https://sparxsystems.com.au/products/ea/history.html

[6] “Bizzdesign - uml multiplicity.” [Online]. Available: https://support.bizzdesign.com/display/knowledge/Setting+the+multiplicity+for+a+UML+attribute%2C+operation+or+association

[7] “Modeling languages- mde.” [Online]. Available: https://modeling-languages.com/useful-presentations-model-driven-engineering-dsls-uml-eclipse-modeling-technologies-2/

[8] “Quidgest - mdd.” [Online]. Available: https://quidgest.com/en/blog-en/what-is-model-driven-development/

[9] “Techtarget - mdd.” [Online]. Available: https://www.techtarget.com/searchsoftwarequality/definition/model-driven-development

[10] S. J. Mellor, A. N. Clark, and T. Futagami, "Guest editors' introduction: Model-driven development," *IEEE Software*, vol. 20, no. 05, pp. 14–18, 2003.

[11] "Sparxea - basic info." [Online]. Available: https://sparxsystems.com/resources/user-guides/16.1/basics/getting-started.pdf

[12] "Sparxea - external integration providers." [Online]. Available: https://sparxsystems.com/enterprise_architect_user_guide/14.0/model_repository/integrate_external_provider_data.html

[13] "Sparxea - compare editions of tool." [Online]. Available: https://sparxsystems.com/products/ea/compare-editions.html

[14] "Sparx systems blog - ea db schema." [Online]. Available: https://blog.sparxsystems.de/ea/ea-features/ea-model-search/enterprise-architect-db-schema/

[15] "Sparxea - uml class diagram." [Online]. Available: https://sparxsystems.com/resources/tutorials/uml2/class-diagram.html

[16] "Visual paradigm - uml class diagram." [Online]. Available: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-class-diagram-tutorial/

[17] "Vsb - asociace." [Online]. Available: https://www.cs.vsb.cz/benes/vyuka/upr/texty/objekty/ch01s02s02.html

[18] "Vsb - agregace a kompozice." [Online]. Available: https://www.cs.vsb.cz/benes/vyuka/upr/texty/objekty/ch01s02s03.html

[19] "Uml diagrams - generalization." [Online]. Available: https://www.uml-diagrams.org/generalization.html

[20] M. Masse, *REST API design rulebook: designing consistent RESTful web service interfaces*. "O'Reilly Media, Inc.", 2011.

[21] "Ibm - rest api." [Online]. Available: https://www.ibm.com/topics/rest-apis

[22] "Redhat - rest api." [Online]. Available: https://www.redhat.com/en/topics/api/what-is-a-rest-api

[23] "Swagger - oas specification." [Online]. Available: https://swagger.io/specification/

[24] "Postman." [Online]. Available: https://www.postman.com/

[25] "Insomnia.rest." [Online]. Available: https://insomnia.rest/

[26] "Swagger.io." [Online]. Available: https://swagger.io/

[27] “Apache avro - docs.” [Online]. Available: https://avro.apache.org/docs/1.10.2/spec.html

[28] D. Vohra and D. Vohra, “Apache avro,” *Practical Hadoop Ecosystem: A Definitive Guide to Hadoop-Related Frameworks and Tools*, pp. 303–323, 2016.

[29] “Apache kafka - documentation.” [Online]. Available: https://kafka.apache.org/documentation/

[30] “Apache freemarker - template engine.” [Online]. Available: https://freemarker.apache.org/

[31] “Openjdk - java records.” [Online]. Available: https://openjdk.org/jeps/359

[32] “Spring - reactive programming.” [Online]. Available: https://spring.io/reactive

[33] “Software engineering principles.” [Online]. Available: https://vpodk.medium.com/principles-of-software-engineering-6b702faf74a6

[34] “Separatrion of concerns.” [Online]. Available: https://dev.to/tamerlang/separation-of-concerns-the-simple-way-4jp2

[35] D. Prasanna, *Dependency injection: design patterns using spring and guice*. Simon and Schuster, 2009.

Příloha A

Seznam použitých zkratek

- IDE - Vývojové prostředí
- EA - Enterprise Architect
- UML - Unified Modeling Language
- API - Application Programming Interface
- UI - User Interface
- MDD - Model Driven Development
- MDE - Model Driven Engineering
- HTTP - Hypertext Transfer Protocol
- LTS - Long-term support
- DI - Dependency injection
- IoC - Inversion of Control
- JSON - JavaScript Object Notation
- OAS - OpenAPI Specification
- YAML - Ain't markup language
- KISS - Keep It Simple, Stupid
- DRY - Don't Repeat Yourself
- SoC - Separation of Concerns

Příloha B

Readme dokumentace frameworku

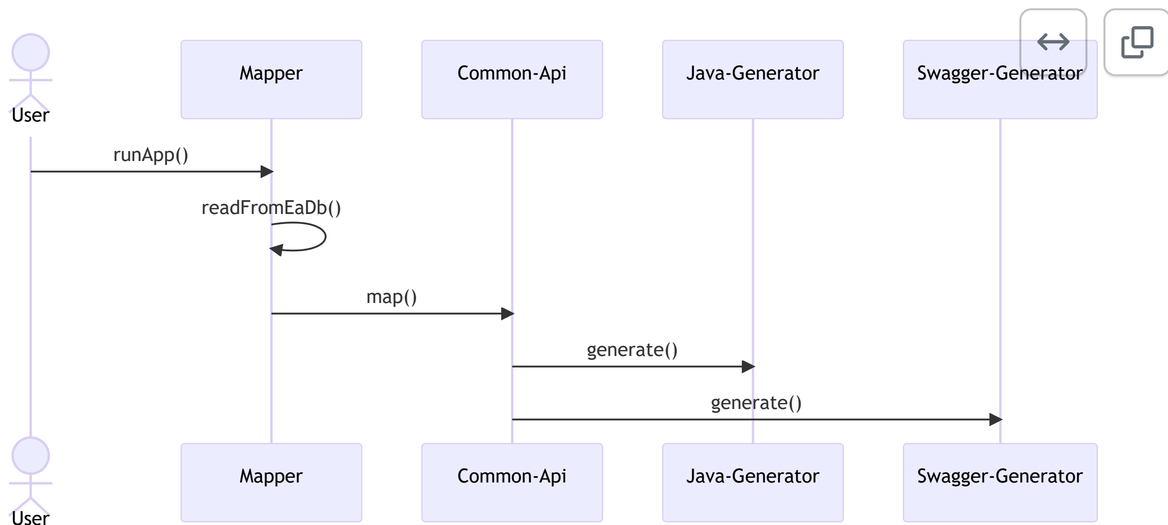
Česká verze dokumentace

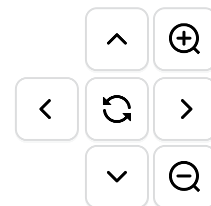
Obsah

- Základní informace
- Kroky ke spuštění aplikace
- Požadavky na použití
- Konfigurační soubor
- Komponenty
  - Common-api
  - Mapper
  - Generátory
- Existující generátory
  - Java SpringBoot
  - Swagger
  - AvroSchema

Základní informace

Tato aplikace slouží k generování kódu z Enterprise Architect modelů. Je plně konfigurovatelné, lze si dopsat vlastní mappery a vlastní generovací profily. Aplikace již obsahuje generátory pro Java (SpringBoot), Swagger a Avro schema. Je navržena jako SpringBoot starter aplikace.





Uživatel si vždy musí napsat svůj vlastní mapper!

Kroky ke spuštění aplikace

- Uživatel si přidal do pom.xml dependency na tento starter
- Uživatel si naimplementoval mapper
- Uživatel si připravil konfigurační .json soubor
- Uživatel spustil aplikaci s argumentem s cestou na konfigurační .json soubor

Požadavky na použití

- Java verze 17 nebo vyšší
- SpringBoot verze 3 nebo vyšší
- Maven

Konfigurační soubor

Konfigurační soubor je ve formátu JSON a vypadá následovně:

```
{
  "databaseConnection": {
    "url" : "DB_URL",
```

```
      "user": "DB_USER",
      "password": "DB_PASSWORD"
    },
    "mappingConfiguration": {
      "type": "custom",
      "profile": "test-mapper"
    },
    "eaStartPackage": "Api.TestPath",
    "version": "1.0.0",
    "enabledGenerators": [
      "generator-a",
      "generator-b",
      "generator-c"
    ],
    "parameters": {
    }
  }
```

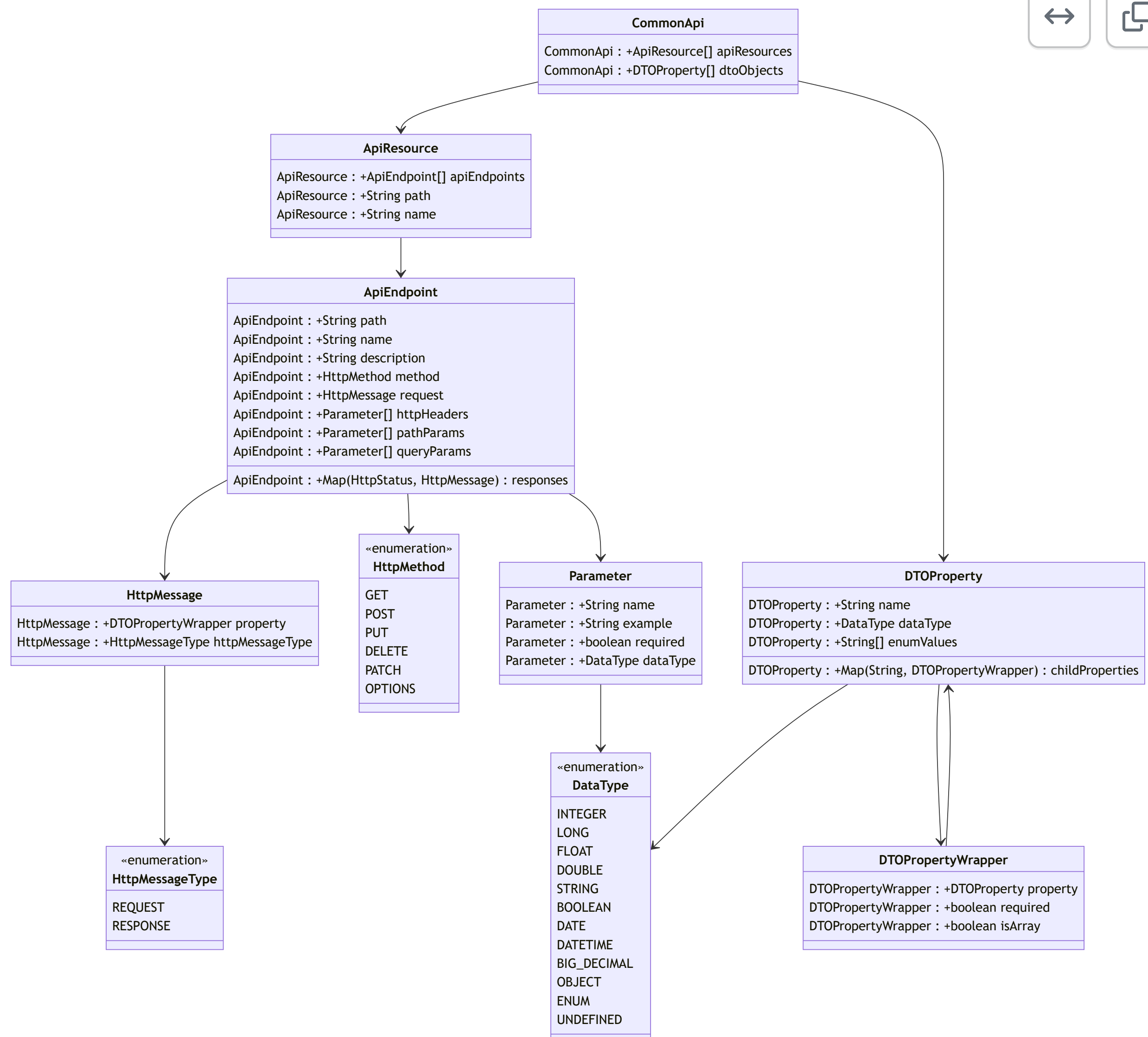
Atribut	Popis	Povinnost
databaseConnection.url	JDBC k EA databázi	✓
databaseConnection.user	username pro připojení k DB	✓
databaseConnection.password	heslo pro připojení k DB	✓
mappingConfiguration.type	typ mapování (v tuto chvíli vždy = "custom"	✓
mappingConfiguration.profile	název mapperu (odpovídá názvu v anotaci @Mapper)	✓
eaStartPackage	Umístění diagramu v EA	✓
version	Verze výstupních souborů	✓
enabledGenerators	Seznam zapnutých generátorů (odpovídá názvu v anotaci @Generator)	✓
parameters	Dodatečné parametry pro jednotlivé generátory	✗

Parameters se vztahují k jednotlivým generátorům a jejich povinnost závisí dle zapnutého seznamu generátorů.

Komponenty

Common-api

Common-api je model, který slouží jako mezivrstva mezi EA modely a profily generátoru. Jeho účel je, aby profil generátoru byl nezávislý na pravidlech modelování v EA.



Mapper

Mapper je komponenta, která se stará o zpracování dat z EA databáze do standardizované common-api. Třída musí být označena anotací a implementovat MapperHandler.

```
import com.mmasata.eagenerator.MapperHandler;
import com.mmasata.eagenerator.annotations.Mapper;

@Mapper(name = "my-mapper")
public class MyMapper implements MapperHandler {

    @Override
    public List<ApiResource> mapApiResources() {
        //implement mapping to common-api ApiResources here
    }

    @Override
    public List<DTOProperty> mapDtoObjects() {
        //implement mapping to common-api DTOProperties here
    }
}
```

Přístup do databáze lze řešit vlastní implementací, nebo využít nadefinované JPA entity ve frameworku. JPA entity se nacházejí v **"com.mmasata.eagenerator.database.entity"** a pro jejich využití je nutné si zapnout anotaci **@EnableGeneratorJpaEntities** - lze si dopsat vlastní JPA repository.

```
@SpringBootApplication
@EnableGeneratorJpaEntities
public class MyApplication {

    public static void main(String[] args) {
        SpringApplication.run(MyApplication.class, args);
    }

}
```

Generátory

Generátor je komponenta, která se stará o zpracování common-api do výsledných souborů. Třída musí být označena anotací a implementovat GeneratorHandler. K dispozici je také Beana FileProcessor, která se stará o zápis do souborů a práci s freemarker šablonami.

```
import com.mmasata.eagenerator.GeneratorHandler;
import com.mmasata.eagenerator.annotations.Generator;
import com.mmasata.eagenerator.processor.FileProcessor;
import lombok.RequiredArgsConstructor;

import java.io.FileWriter;
```

```java
import java.io.StringWriter;
import java.util.Map;

@Generator(name = "my-generator")
@RequiredArgsConstructor
public class MyGenerator implements GeneratorHandler {

    private final FileProcessor fileProcessor;

    @Override

    public void run() {
        //implement generator logic here
        Writer fileData = new StringWriter();
        fileProcessor.generate("myFile.txt", fileData);
    }

}
```

Existující generátory

Framework v sobě obsahuje několik zabudovaných generátorů, které lze spustit.

Java SpringBoot

Název	Popis
java-spring	Název generátoru
parameters.javaDistributionManagementRepoId	ID distribučního repositáře
parameters.javaDistributionManagementRepoUrl	URL adresa distribučního repositáře
parameters.javaDtoType	typ modelů (třídy s LOMBOK či RECORDS)
parameters.javaControllerType	typ Rest controllerů (REACTIVE nebo STANDARD)
parameters.javaVersion	verze javy
parameters.javaGroupId	groupId vygenerovaného projektu
parameters.javaProjectName	název vygenerovaného projektu
parameters.javaPackage	root package vygenerovaných java souborů

Swagger

Název	Popis
swagger	Název generátoru
parameters.swaggerTitle	Název vygenerovaného swaggeru dokumentu

AvroSchema

Název	Popis
avro-schema	Název generátoru