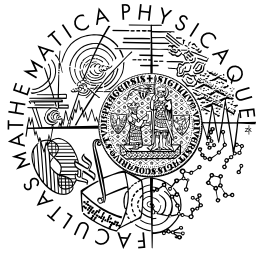


MATEMATICKO-FYZIKÁLNÍ
FAKULTA
Univerzita Karlova

BAKALÁŘSKÁ PRÁCE

Andrej Pečimúth

Datový generátor

Katedra softwarového inženýrství

Vedoucí bakalářské práce: RNDr. Michal Kopecký, Ph.D.

Studijní program: Informatika

Studijní obor: Obecná informatika

Praha 2021

Prohlašuji, že jsem tuto bakalářskou práci vypracoval(a) samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů. Tato práce nebyla využita k získání jiného nebo stejného titulu.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V ..... dne ....

Podpis autora

Rád by som sa poďakoval vedúcemu RNDr. Michalovi Kopeckému, Ph.D. za jeho podnetné pripomienky, ktorými ma usmerňoval pri tvorbe tejto práce.

Název práce: Datový generátor

Autor: Andrej Pečimúth

Katedra: Katedra softwarového inženýrství

Vedoucí bakalářské práce: RNDr. Michal Kopecký, Ph.D., Katedra softwarového inženýrství

Abstrakt: Práce se zaměřuje na návrh generátoru tabulkových dat. Naše řešení čte strukturu vstupních dat pocházejících z různých zdrojů. Toto schéma lze dále upravovat pomocí grafického prostředí. Ke každému sloupci jsou přiřazeny parametrizovatelné generátory. Generátory a jejich parametry jsou automaticky vybírány systémem tak, aby se výstupní záznamy podobaly vstupním. Výstupní sada záznamů respektuje omezení integrity. Podporováno je více výstupních formátů. Práce kombinuje výhody stávajících webových a nativních řešení. Do webového prostředí přinášíme funkce, které byly dříve dostupné pouze v nativních aplikacích. Kromě toho jsme vyřešili problém kruhových závislostí mezi tabulkami.

Klíčová slova: databáze, data, generátor

Title: Data generator

Author: Andrej Pečimúth

Department: Department of Software Engineering

Supervisor: RNDr. Michal Kopecký, Ph.D., Department of Software Engineering

Abstract: The work focuses on the design of a tabular data generator. Our solution reads the structure of input data coming from different sources. This schema can be further modified using a graphical environment. Parameterizable generators are associated with each column. The generators and their parameters are automatically selected by the system so that the output records resemble the input ones. The output set of records respects integrity constraints. Multiple output formats are supported. The work combines the advantages of existing web and native solutions. We bring functionality previously only available in native applications to the web environment. In addition, we have solved the problem of circular dependencies between tables.

Keywords: database, data, generator

Názov práce: Dátový generátor

Autor: Andrej Pečimúth

Katedra: Katedra softwarového inženýrství

Vedúci bakalárskej práce: RNDr. Michal Kopecký, Ph.D., Katedra softwarového inženýrství

Abstrakt: Práca sa zameriava na návrh generátora tabulárnych dát. Naše riešenie prečíta štruktúru vstupných dát pochádzajúcich z rôznych zdrojov. Táto schéma môže byť ďalej upravovaná pomocou grafického prostredia. S jednotlivými stĺpcami sa spájajú parametrizovateľné generátory. Generátory a ich parametre systém automaticky zvolí tak, aby sa výstupné záznamy podobali na vstupné. Výstupná sada záznamov rešpektuje integritné obmedzenia. Podporované sú viaceré formáty výstupu. Práca spája výhody existujúcich webových a natívnych riešení. Funkcionalitu doteraz dostupnú iba v natívnych aplikáciách prinášame do webového prostredia. Navyše sme vyriešili problém cirkulárnych závislostí medzi tabuľkami.

Kľúčové slová: databáza, dáta, generátor

Obsah

Kapitola 1

Úvod

V životnom cykle databázových aplikácií sa vyskytujú viaceré situácie, kedy je žiadané využiť vygenerované dáta. Pri zavádzaní nových modulov aplikácie sa často snažíme simulovať reálne využitie. Tieto dáta nám pomôžu odhaliť chyby v prvotnej fáze vývoja. Mali by byť čo najviac rôznorodé, aby pokryli aj okrajové prípady. Navyše získavame pohľad, ako bude aplikácia vypadať pri používaní, čo môže určovať aj ďalší smer vývoja. Mali by byť realistické, aby odrážali spôsob, ktorým bude aplikáciu využívať zákazník. Žiadúce môže byť aj vyskúšať stabilitu a výkon aplikácie pri väčšom objeme dát. To nám môže pomôcť pri odladovaní ešte pred nasadením do produkcie.

Produkčné dáta môžu obsahovať osobné údaje, zahešované heslá alebo údaje, ktoré by mali zostať obchodným tajomstvom. Na jednej strane môže byť výhodné, aby testeri a vývojári pracovali so skutočnými dátami. Na druhej strane chceme minimalizovať riziko, že príde k úniku týchto dát. Vhodným kompromisom v tejto situácii môže byť vygenerovať podobné dáta po vzore produkčných, ktoré spĺňajú podmienky na súkromie.

Syntetické dáta môžu byť využité aj pri regresných testoch alebo benchmarkoch. Dátové generátory nám poskytujú vysokú kontrolu nad podobou testovacej množiny.

Na úvod zadefinujeme ciele projektu. V kapitole 2 porovnáme existujúce riešenia a analyzujeme dátové zdroje. Kapitola 3 predstaví špecifikáciu nášho softvéru. V kapitole 4 zvolíme vhodné technológie a algoritmy. Kapitola 5 popisuje naše riešenie z pohľadu vývojára. Podobne v kapitole 6 poskytneme návod pre koncového užívateľa. Možnosti ďalších vylepšení softvérového riešenia ukážeme v kapitole 7. Na záver vyhodnotíme našu prácu a porovnáme ju s inými riešeniami.

1.1 Ciele práce

V tejto práci sa budeme zameriavať na riešenie vyššie predstavených problémov. Naším koncovým užívateľom budú softvéroví vývojári alebo testeri snažiaci sa vygenerovať dáta pre svoju aplikáciu. Tomu prispôsobujeme ciele projektu, ktoré sme zoradili do niekoľkých kategórií.

Automatizácia

Riešenie by si malo vyžadovať čo najmenej klikov od užívateľa. Ak je proces možné vykonať automaticky, mal by byť automatizovaný. Môžeme tým hneď dosiahnuť želaný výsledok a užívateľ ušetrí prácu. Ak je výsledok neželaný, dáme užívateľovi možnosť upraviť si nastavenia podľa vlastných predstáv. Dosiahneme to využitím referenčných dát, po vzore ktorých budú generované výstupné dáta.

Všeobecnosť

Pri návrhu aplikácie by sme sa mali pokúšať o maximálnu všeobecnosť. Referenčné dáta by sme mali byť schopní prijímať z rôznych zdrojov — vrátane databázových systémov a textových formátov. Generovanie dát by rovnako malo podporovať rôzne výstupné formáty.

Vygenerované hodnoty budú slúžiť ako testovacie dáta do aplikácií. Musíme pokrývať dostatočnú škálu dátových typov. Okrem toho by povaha dát mala byť rovnaká — ak niekde očakávame napríklad telefónne číslo, vygenerovaná hodnota by mala pripomínať telefónne číslo. Pri testovaní výkonu môže byť dôležité aj štatistické rozdelenie hodnôt. Generované dáta by teda mali byť schopné odrážať vybrané štatistické vlastnosti referenčných dát.

Multiplatformovosť

Aby sme nestrácali potenciálnych užívateľov, musí naše riešenie fungovať na každom relevantnom operačnom systéme. Rovnako by mala byť podporovaná najväčšia možná množina databázových systémov. Tiež je potrebné počítať s tým, že naše riešenie môže byť použité ako súčasť iného systému — napríklad v rámci automatizovaného procesu. Je teda žiadúce ponúknuť rozhrania, ktoré sú využiteľné z viacerých programovacích jazykov.

Otvorenosť

Predpokladáme, že akokoľvek bude naše riešenie kompletné, vždy sa nájde užívateľ, ktorému bude nejaká vlastnosť chýbať. Vzhľadom na predpokladanú tech-

nickú zdatnosť by užívateľ mohol byť schopný toto rozšírenie urobiť sám, Našou úlohou je k tomu poskytnúť dostatočne všestranné nástroje.

Tento softvér musí byť open source s permisívnou licenciou. Dovolíme využitie v rámci komerčného systému bez povinnosti zverejniť upravené zdrojové kódy. V rámci architektonického návrhu identifikujeme časti, pri ktorých budeme očakávať rozširovanie. Týmto častiam budeme venovať špeciálnu pozornosť. Rozhrania navrhneme tak, aby si zmeny vyžadovali minimálne množstvo kódu a nevyžadovali si pochopenie celej architektúry aplikácie.

Kapitola 2

Analýza

V tejto kapitole presnejšie definujeme požiadavky na dátový generátor. Prejdeme niekoľko existujúcich riešení a zhodnotíme, ako spĺňajú tieto požiadavky. Ďalej predstavíme relevantné dátové zdroje. Priblížime s nimi súvisiace problémy. Zhodnotíme, akým spôsobom sa s týmito problémami vysporiadali existujúce riešenia.

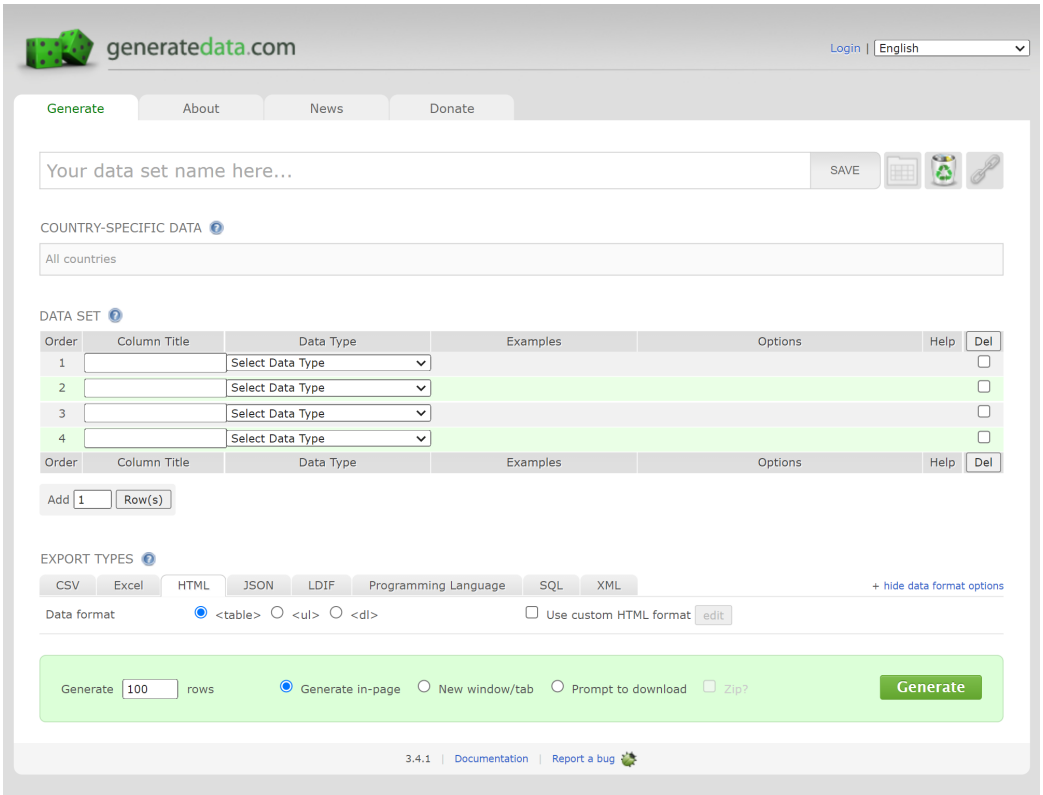
2.1 Dátové generátory

Dátové generátory môžu byť distribuované buď ako webové aplikácie alebo natívne aplikácie. Webové aplikácie sú lepšie v tom, že nevyžadujú sťahovanie a inštaláciu. Pozitívum je aj ich multiplatformovosť. Niektoré z nich okrem grafického prostredia ponúkajú aj programátorské prostredie. Môže to byť vo forme REST API alebo rozhrania príkazového riadku (CLI).

Aplikácie môžu priamo spolupracovať s databázovým systémom (DBMS). Vzhľadom k rozdielnym implementáciám databázových systémov musíme kontrolovať, či je daný softvér kompatibilný s naším DBMS. Niektoré riešenia dokážu vygenerovať iba príkazy `INSERT`, ktoré musí užívateľ preniesť do príslušného databázového systému. Ďalšie formáty výstupu vygenerovaných dát môžu zahŕňať textové formáty ako CSV, JSON či XML.

Z databáz je možné prečítať ich schému. Dátový generátor sa môže od tohto odraziť a vytvoriť sadu tabuliek a stĺpcov. Dôležité sú aj dátové typy jednotlivých stĺpcov a integritné obmedzenia. Tieto vlastnosti ovplyvňujú skutočnosť, aké dáta sú akceptovateľné pre databázový systém.

K jednotlivým stĺpcom v schéme sú priradené generátory. Výber generátorov sa líši naprieč rôznymi riešeniami. Niektoré z nich majú v grafickom prostredí integrované textové pole, do ktorého môžeme písať kód. Môže ísť o zjednodušený jazyk, v ktorom napíšeme jeden výraz. Ďalšia možnosť je napísať funkciu



Obr. 2.1 Screenshot webovej stránky `generatedata.com`.

v jazyku ako Ruby či Python. Touto funkciou sa generujú hodnoty pre vybraný stĺpec.

Dátové generátory sa líšia v tom, či zohľadňujú závislosti medzi tabuľkami — cudzie kľúče. Niektoré riešenia sa zaoberajú súčasne iba jednou tabuľkou a nie sú schopné vytvoriť žiadne závislosti tohto typu.

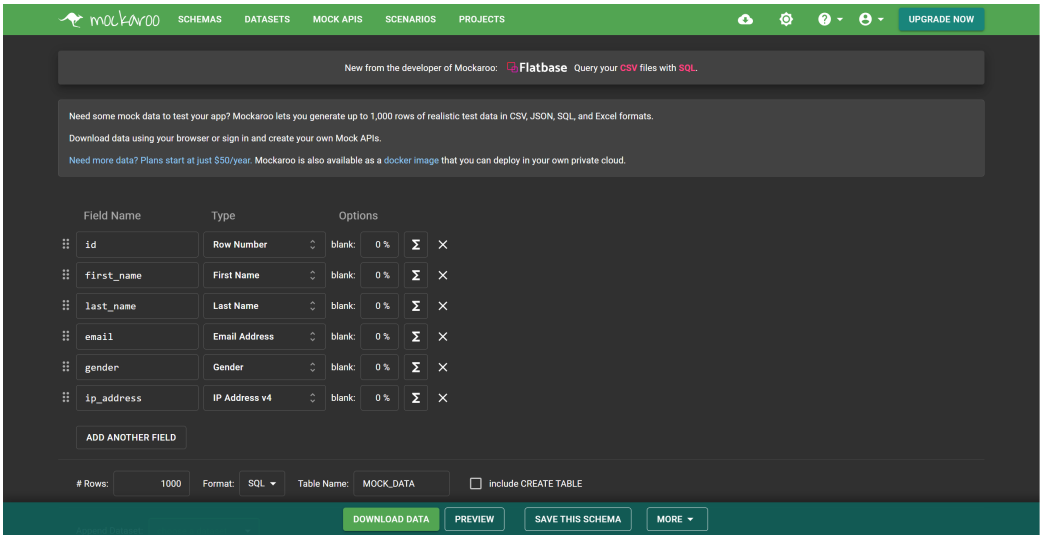
2.1.1 Analýza existujúcich riešení

V nasledujúcom texte analyzujeme štyri mierne odlišujúce sa systémy.

generatedata

Generatedata je open source dátový generátor.[1] Ide o webovú aplikáciu, ktorá je spustiteľná aj lokálne. Screenshot verejne dostupnej webovej stránky uvádzame na obrázku 2.1. Verejná verzia softvéru obsahuje značné obmedzenia — generuje

[1] `http://benkeen.github.io/generatedata/`



Obr. 2.2 Screenshot úvodnej stránky generátora Mockaroo.

fixne 100 záznamov. Z našich štyroch vybraných aplikácií obsahuje najmenej funkcií, ale je z nich jediná voľne šíriteľná.

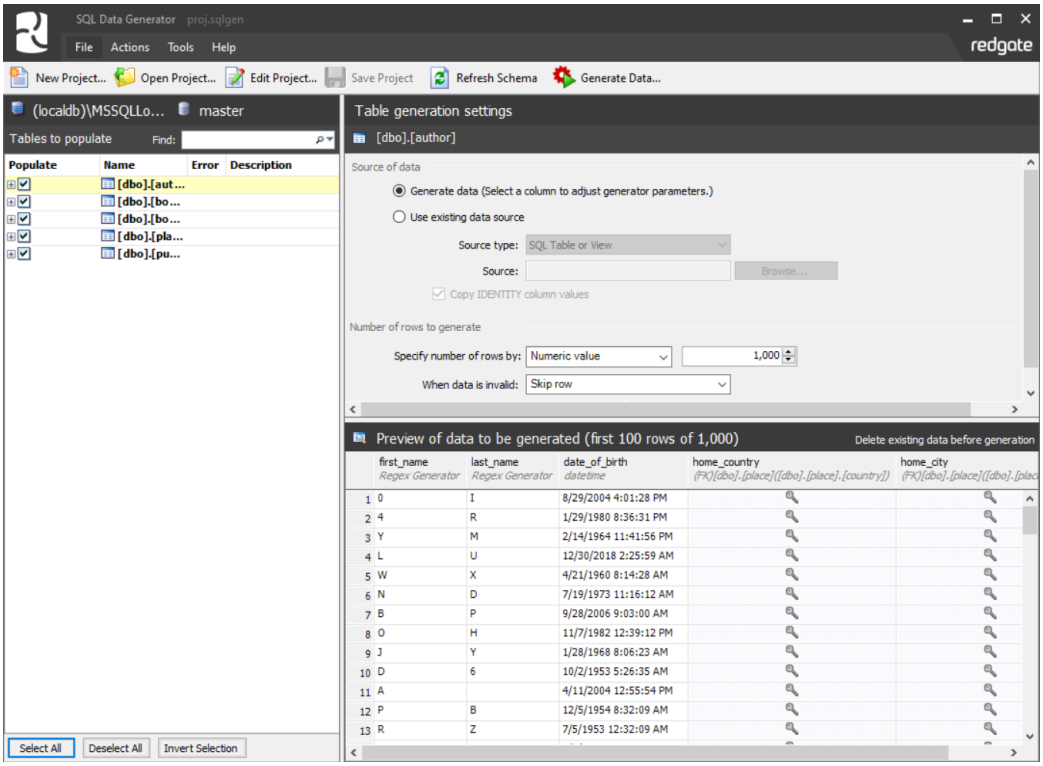
Aplikácia pracuje na jednej interaktívnej stránke. Dovolí nám zvoliť si krajinu, voči ktorej majú byť vygenerované dáta špecifické. Zadefinujeme si názvy stĺpcov tabuľky. Pre každý stĺpec si môžeme zvoliť preddefinovaný generátor. Generátory sú zotriedené do kategórií ako ľudské údaje, geografické údaje, textové, číselné a iné. Niektoré z nich majú ďalšie parametre. Napríklad pri číselnom generátore volíme jeho rozsah. Ďalej určíme počet žiadaných záznamov. Ponuka zahŕňa niekoľko formátov výstupu, vrátane CSV, JSON, XML alebo príkaz `INSERT` v jazyku SQL. Dáta sa zobrazujú v textovom poli alebo ich môžeme stiahnuť.

Mockaroo

Mockaroo je ďalší webový generátor.[2] Dostupný je v demo verzii ako webová stránka na obrázku 2.2. Rýchlosť a počet vygenerovaných riadkov je limitovaný, limity sú zvýšené v komerčnej verzii. V komerčnej verzii ponúka aj možnosť lokálneho spustenia.

Princíp fungovania spočíva v tom, že užívateľ si manuálne navolí názvy stĺpcov a k ním vyberie jeden z preddefinovaných typov. Tento typ neurčuje len dátový typ, ale aj obsah vyplneného stĺpca. Dostupných je viac ako 100 typov zotriedených do kategórií. Pri každom z nich máme možnosť dodatočnej konfigurácie. Napríklad môžeme určiť, koľko výsledných hodnôt má byť `null`. Zaujímavý je typ `Formula`, prostredníctvom ktorého je možné zapísať výraz v jazyku

[2] `https://www.mockaroo.com/api/docs`



Obr. 2.3 Screenshot ukážkového projektu v SQL Data Generator 4.

Ruby. Tento výraz sa opakovane vyhodnocuje pre každý vygenerovaný riadok. To nám umožňuje generovať dáta podľa vlastnej logiky. V rámci tohto výrazu sa môžeme odkazovať aj na hodnoty ostatných stĺpcov.

Nakoniec je možné zvoliť počet želaných riadkov a jeden z výstupných formátov. Tie zahŕňajú bežné formáty štruktúrovaných dát ako JSON, XML, CSV a ďalšie. Tiež máme možnosť vytvoriť `INSERT` príkaz. Unikátna vlastnosť je vytvorenie REST API vracajúcej generované záznamy.

Red Gate SQL Data Generator 4

SQL Data Generator 4 je komerčný generátor.[3] Funguje ako natívna aplikácia iba pre operačný systém Windows. Spolupracuje priamo s databázovým systémom SQL Server, iné systémy nie sú podporované. Na obrázku 2.3 ukazujeme otvorený ukážkový projekt.

Princíp tohto riešenia spočíva v tom, že zadáme prístupové údaje k databáze. Následne je automaticky načítaná jej schéma — tabuľky a stĺpce. Pri stĺpcoch prečíta ich názvy, dátové typy a integritné obmedzenia ako primárne kľúče, cudzie

[3] `https://documentation.red-gate.com/sdg4`

kľúče a obmedzenia na unikátnosť. Každému stĺpcu je pridelený a nakonfigurovaný nejaký z preddefinovaných generátorov. Konkrétne parametre závisia nielen na názve a dátovom type, ale aj na dátach v databáze, ku ktorej sme pripojení. Napríklad pri číselných hodnotách z nej prečíta maximum a minimum. Následne vyberá hodnoty uniformne z tohto intervalu.

Podporované sú cudzie kľúče, ktoré môžu byť kompozitné a aj prekrývajúce sa. Vždy sa najprv vyplnia všetky tabuľky, do ktorých cudzie kľúče odkazujú. Následne sa pre daný kľúč náhodne vyberajú hodnoty z množiny vygenerovaných hodnôt v cieľovej tabuľke. S cirkulárnymi závislosťami medzi tabuľkami si tento systém neporadí — nedokáže vygenerovať nič. Algoritmus výberu hodnôt pre cudzie kľúče je prispôsobiteľný v prípade nekompozitných kľúčov.

Na výber dostávame mnoho preddefinovaných procedúr pre vytváranie konkrétnych hodnôt. Ponúkajú možnosť naplniť stĺpec importovanými dátami zo stĺpca v CSV súbore alebo z textového súboru. Máme možnosť naprogramovať vlastný script pre generovanie v jazykoch Python alebo C#. Pomocou špeciálne vytvoreného jazyka je možné zapísať jednoriadkový výraz, ktorým budú generované záznamy.

Riešenie SQL Data Generator nám dáva možnosti použiť ho spolu s inými nástrojmi. Všetky naše nastavenia sú uložiteľné do projektového súboru. Spustením programu s cestou k tomuto súboru spustíme napĺňanie databázy.

Je možné pracovať s existujúcimi dátami vo formáte CSV, tabuľkou z externej databázy, view či príkazu v jazyku SQL. Hodnotami z týchto stĺpcov môžeme naplniť niektorý z cieľových stĺpcov.

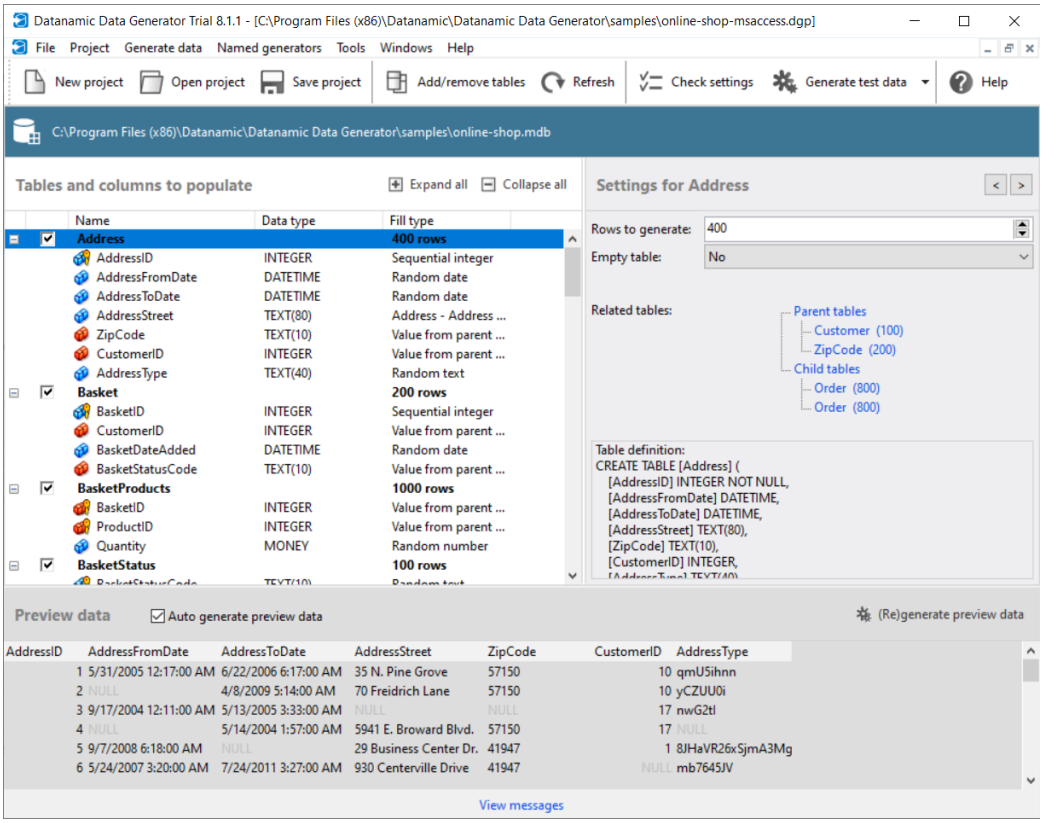
Pri načítaní schémy zohľadňuje požiadavky na unikátnosť hodnôt v stĺpci. `CHECK` obmedzenia sú zohľadnené pri automatickej voľbe parametrov generátora.

Datanamic Data Generator

Datanamic Data Generator je komerčná GUI aplikácia pre Windows.[4] Podporuje množstvo relačných databázových systémov. Ku všetkým z nich sa dokáže pripojiť a prečítať ich štruktúru. Na základe charakteristík jednotlivých stĺpcov im priradí jeden z pripravených generátorov. Zohľadňuje dátové typy a integritné obmedzenia. Stĺpec dokáže naplniť dátami z textového súboru, súboru vo formáte CSV, inej databázy alebo z príkazu v SQL. Výstupom je priame naplnenie databázy alebo script v jazyku SQL. Generovanie je spustiteľné aj prostredníctvom príkazového riadku a uložených nastavení.

Ukážkový projekt so schémou online obchodu je na obrázku 2.4. Princíp fungovania sa v mnohom podobá na vyššie analyzovaný SQL Data Generator 4. Zásadný rozdiel spočíva v širšej podpore databáz. Naopak, chýba možnosť vytvoriť

[4] `https://www.datanamic.com/datagenerator/`



Obr. 2.4 Screenshot ukážkového projektu v Datanamic Data Generator.

Názov	generatedata	Mockaroo	Red Gate	Datanamic
Licencia	GPL 3	komerčná	komerčná	komerčná
Open source	áno	nie	nie	nie
Platforma	web	web	Windows	Windows
Podpora DBMS	`INSERT`	`INSERT`	SQL Server	mnohé
Iné výstupy	mnohé	mnohé	nie	`INSERT`
API	REST	REST	CLI	CLI
Schéma z DB	nie	nie	áno	áno
Programovanie	nie	Ruby	mnohé	SQL
Cudzie kľúče	nie	nie	áno	áno

Tabuľka 2.1 Vlastnosti generátorov v skratke

vlastnú procedúru pre generovanie programovacom jazyku pre všeobecné účely.

2.1.2 Zhrnutie

Analyzovali sme štyri existujúce generátory, môžeme ich rozdeliť na dva typy. Na jednej strane máme webové technológie generatedata a Mockaroo. Webové prostredie si nevyžaduje sťahovanie a inštaláciu aplikácie. Zreteľná výhoda je krátka doba, kým začneme generovať zmysluplné dáta. Nepripájajú sa k databáze, teda nedokážu prečítať schému alebo automatizovať konfiguráciu generátorov pre jednotlivé stĺpce. Ponúkajú široký výber výstupných formátov. V kontexte databáz dokážu vygenerovať `INSERT` príkaz. Obe riešenia ponúkajú REST API ako rozhranie pre programátorov. Výstup je obmedzený na jednu tabuľku, ani v jednom prípade nám nedovolia vytvoriť viac ako pár tisíc riadkov.

Na druhej strane spoločnosti Red Gate a Datanamic ponúkajú komplexnejšie riešenia. Obe sú dostupné ako natívne aplikácie pre Windows s komerčnou licenciou. Ich prednosti spočívajú v možnostiach nastaviť generátory automaticky podľa schémy databáz. Fungujú lepšie v kontexte relačných databáz, pretože počítajú s existenciou integritných obmedzení a pracujú s konceptom cudzích kľúčov. Dáta vkladajú priamo do databázy. Prostredníctvom CLI môžu byť využité ako súčasti iných aplikácií. Tabuľka 2.1 zobrazuje zhrnutie vlastností analyzovaných riešení.

Naša aplikácia by mala kombinovať výhody oboch typov riešení. Podobne ako webové riešenia by sme mali ponúkať viaceré výstupné formáty. Natívne riešenia nás inšpirujú v úrovni automatizácie. Schému by sme mali čítať zo zdrojových dát. Je nutné dbať na integritu generovaných záznamov. Nemusíme sa však obmedzovať iba na databázové systémy — vstupné dáta môžu byť v akomkoľvek formáte. Ide o zovšeobecnenie princípov existujúcich dátových generátorov do jedného produktu. Toto všetko by sme mali ponúkať ako rozšíriteľný

systém s rovnakou dostupnosťou ako webové riešenia.

2.2 Dátové zdroje

Cieľom tejto sekcie je analyzovať rôzne podoby dátových zdrojov. Za dátové zdroje považujeme systémy na správu dát — relačné databázové systémy, ale aj textové formáty na výmenu štruktúrovaných dát.

Rovnako ako analyzované softvérové diela, budeme pracovať s tabulárnymi dátami. To znamená, že budeme mať súbor tabuliek. Tabuľka obsahuje niekoľko stĺpcov s určenými dátovými typmi. Záznam v tabuľke obsahuje pre každý stĺpec hodnotu zodpovedajúceho typu alebo prázdnu hodnotu (`null`).

Relačné databázové systémy a textové formáty sa významne líšia svojimi možnosťami a vlastnosťami. Našou úlohou bude vytvoriť zjednocujúci pohľad. Potom aplikácia, ktorú špecifikujeme v ďalšej kapitole, bude nezávislá na podobe jej vstupných a výstupných dát.

Dátový generátor prichádza do styku s dátovým zdrojom pri troch rôznych úlohách:

- čítanie referenčných záznamov,
- čítanie referenčnej schémy (metadáta),
- zapisovanie vygenerovaných dát.

Pri každom type dátového zdroja musíme vyriešiť každú z uvedených úloh. Na začiatku sekcie sa budeme venovať relačným databázovým systémom. Tu sa budeme zameriavať najmä na problémy súvisiace s integritnými obmedzeniami. Zhodnotíme, ako sa nimi vysporiadali ostatné softvérové riešenia.

V ďalšej podsekcii rozoberieme textové formáty. Pri nich sa budeme zameriavať na určenie metadát, ktoré z nich vieme vyčítať. Niektoré formáty bude nutné obmedziť, pretože ich dáta nemajú len tabuľkovú formu. Týmto sa nesnažíme vymenovať kompletnú sadu podporovaných formátov. Náš finálny systém bude musieť byť v súlade s cieľmi všeobecný. Základný zoznam textových formátov nám však pomôže definovať spoločnú reprezentáciu dát v aplikácii.

2.2.1 Relačné databázové systémy

V nasledujúcej sekcii rozoberieme vlastnosti relačných databáz týkajúce sa našej úlohy. Vymenujeme problémy, s ktorými sa stretneme pri vývoji a načrtneme možné riešenia.

Výpis kódu 1 Čítanie štruktúry databáz. Porovnanie SQL Server a PostgreSQL. Formáty sa líšia medzi rôznymi databázovými systémami.

```
-- SQL Server 2019
SELECT * FROM sys.tables
SELECT * FROM sys.columns
-- sys.foreign_keys, sys.indexes, sys.key_constraints, ...

-- PostgreSQL 13
SELECT * FROM tables
SELECT * FROM columns
-- table_constraints, constraint_column_usage, ...
```

Štruktúra

Podobne ako iné softvérové riešenia budeme čítať štruktúru databázy. Jednotlivé databázové systémy nám ponúkajú navzájom nekompatibilné možnosti, ako k týmto informáciám pristupovať. SQL Server[5] a PostgreSQL[6] sprístupňujú objekty, z ktorých prečítame informácie o tabuľkách, stĺpcoch a integritných obmedzeniach. Príklady použitia uvádzame vo výpise 1.

SQL Data Generator 4 podporuje iba SQL Server. Takýto prístup umožňuje detailne prispôsobiť správanie generátora jedinému databázovému systému. Pri tvorbe multiplatformového riešenia stojí za zváženie výber knižnice, ktorá nám ponúkne jednotné rozhranie na čítanie databázovej štruktúry.

Primárne kľúče

Primárny kľúč jednoznačne určuje záznam v tabuľke. Z pohľadu dátového generátora dát musíme zabezpečiť, aby vygenerovaná hodnota bola unikátna. Žiadna jeho časť nesmie mať hodnotu `null` [1, s. 76].

V špeciálnom prípade sa môže jednať o *surrogate key*, ktorý je typicky automaticky zvyšujúce sa číslo pridelené databázou [1, s. 179]. Pri vkladaní do databázy nie je žiadúce pre takýto stĺpec vygenerovať hodnotu. Nastanú však situácie, kedy budeme potrebovať získať jeho databázou vygenerovanú hodnotu. Vo všeobecnom prípade sa môže jednať o primárny kľúč skladajúci sa z jedného alebo viacerých stĺpcov. Jednotlivé jeho stĺpce môžu mať rôzne dátové typy.

Existujúce webové riešenia nepracujú so zdrojovými schémami databáz. Riešenie primárnych kľúčov zostáva na užívateľovi, ako si nastaví definíciu tabuľky.

[5] `https://docs.microsoft.com/en-us/sql/relational-databases/system-catalog-views/object-catalog-views-transact-sql`

[6] `https://www.postgresql.org/docs/13/information-schema.html`

SQL Data Generator 4 podporuje iba priamy výstup do databázy. V prípade surrogate key necháva hodnotu prideliť SQL Serverom. Konkrétne čísla nezobrazuje ani v náhľade generovaných dát — vidíme iba ikonku naznačujúcu, že hodnota bude pochádzať z databázy.

Datanamic Data Generator dokáže v prípade surrogate key identifikovať, že ide o sekvenčne zvyšujúcu sa hodnotu. Toto riešenie umožňuje výstup do súboru, v tom prípade generuje aj hodnoty primárnych kľúčov. Problém je, že sa snaží do databázy priamo vkladať konkrétne kľúče. Ak databáza už obsahuje záznamy s takými kľúčmi, vygenerované hodnoty neprijme.

Cudzie kľúče

Vo výstupe generátora musíme zabezpečiť dodržanie referenčnej integrity. To znamená, že každý cudzí kľúč musí mať buď hodnotu `null` (ak nie je z iného dôvodu zakázaná) alebo musí mať hodnotu primárneho kľúča v referencovanej tabuľke.

Vzniká tu mnoho komplikácií. Cudzie kľúče môžu byť kompozitné — skladajúce sa z viacerých stĺpcov. Počítať musíme aj s tým, že sa budú prekrývať. Môžu sa dokonca prekrývať s primárnym kľúčom.

Webové riešenia sa cudzími kľúčmi nezaoberajú. Natívne riešenia cudzie kľúče identifikujú a tieto stĺpce riešia špeciálnymi generátormi. SQL Data Generator 4 vyberá hodnoty cudzích kľúčov náhodne z hodnôt v referencovanej tabuľke. Týmto zabezpečuje integritu generovaných hodnôt. Toto správanie nám dovoľuje prispôsobiť výberom jednej z troch možností.

- „all key values unique“ — každý cudzí kľúč sa odkazuje na iný záznam,
- „repeat key values“ — viacero záznamov v tabuľke môže mať zhodný cudzí kľúč,
- „repeat key values between“ — počet záznamov so zhodným cudzím kľúčom je obmedzený intervalom.

Druhé natívne riešenie ponúka rovnaké možnosti, akurát sú inak pomenované. Obe riešenia nám dovoľujú využiť aj iný typ generátora pre cudzie kľúče (napr. náhodné číslo z rozsahu).

Zjavný problém pre takýto generátor môže byť, keď medzi tabuľkami existuje cyklická závislosť. Vo výpise 2 definujeme dve navzájom závislé tabuľky. Keď zadáme takéto tabuľky do systému SQL Data Generator 4, varuje nás, že nie je schopný generovať žiadne záznamy.

Výpis kódu 2 Minimálna definícia tabuliek s cyklickými závislosťami v PostgreSQL.

```
CREATE TABLE "TableA" (
    id SERIAL NOT NULL,
    fk INTEGER,
    PRIMARY KEY (id)
);
CREATE TABLE "TableB" (
    id SERIAL NOT NULL,
    fk INTEGER,
    PRIMARY KEY (id)
);
ALTER TABLE "TableB" ADD FOREIGN KEY(fk)
REFERENCES "TableA" (id);
ALTER TABLE "TableA" ADD FOREIGN KEY(fk)
REFERENCES "TableB" (id);
```

Výpis kódu 3 Generovaný SQL script pre dve cyklicky závislé tabuľky softvérom Datanamic Data Generator. Databáza neprijme ani jeden záznam.

```
INSERT INTO "public"."TableA" ("id","fk") VALUES (1,3);
INSERT INTO "public"."TableA" ("id","fk") VALUES (2,3);
INSERT INTO "public"."TableA" ("id","fk") VALUES (3,1);

INSERT INTO "public"."TableB" ("id","fk") VALUES (1,3);
INSERT INTO "public"."TableB" ("id","fk") VALUES (2,3);
INSERT INTO "public"."TableB" ("id","fk") VALUES (3,3);
```

Výpis kódu 4 Odložená kontrola obmedzení v transakcii.

```
BEGIN TRANSACTION;
SET CONSTRAINTS ALL DEFERRED;
-- INSERT ...
COMMIT;
```

Výpis 3 ukazuje výstup z druhého testovaného softvéru, ktorý nie je prípustný. Záznamy sa odkazujú na neexistujúce cudzie kľúče. Tento výstup v našom teste nebol prijatý databázovým systémom. Nepomôže ani zmena poradia — žiadny zo záznamov nemôže byť vložený ako prvý.

Dátovému generátoru zostáva ešte niekoľko možností, ako túto situáciu vyriešiť. Pomohla by „odložená kontrola integrity“ [2, ss. 315–317]. Integrita je v takom nastavení kontrolovaná až na konci transakcie. Príkazy `INSERT` zoskupíme do transakcie, pričom na začiatku nastavíme odloženie kontroly. Nastavenie môže byť cielené na obmedzenia podľa mena. Systém PostgreSQL dovoľuje vypnúť všetky obmedzenia.[7] Tento postup ukazujeme vo výpise 4.

Nevýhoda je, že obmedzenie musí byť nastavené ako odložiteľné. SQL Server odloženú kontrolu nepodporuje.[8] Umožňuje však vypnúť kontrolovanie obmedzenia prostredníctvom úpravy tabuľky.[9] Po vložení záznamov by bolo možné kontrolu naspäť zapnúť. Odloženie kontroly aj vypínanie obmedzení sú citlivé na podobu schém a rozdiely medzi implementáciami databázových systémov.

Ďalšia alternatíva je v systéme Datanamic Data Generator nastaviť tak, aby všetky cudzie kľúče dostali hodnotu `null`. V takom prípadne by sme vygenerovali prípustné záznamy, ale nie užitočné.

Žiadnu z vyššie prezentovaných alternatív nepovažujeme za vhodnú. Pri návrhu nášho generátora sa budeme snažiť vytvárať záznamy v takom poradí a s takým obsahom, aby boli prijaté databázovým systémom. Návrhu konkrétneho algoritmu sa budeme venovať v kapitole 4.

Integritné obmedzenia

Ďalšie integritné obmedzenia pre jednotlivé stĺpce zahŕňajú [1, ss. 370–371]:

1. `NOT NULL` — stĺpec nesmie obsahovať hodnotu `null`,

[7] `https://www.postgresql.org/docs/13/sql-set-constraints.html`

[8] `https://docs.microsoft.com/en-us/sql/t-sql/statements/alter-table-transact-sql?view=sql-server-ver15`

[9] `https://docs.microsoft.com/en-us/sql/relational-databases/tables/disable-check-constraints-with-insert-and-update-statements?view=sql-server-ver15`

2. `UNIQUE` — hodnota stĺpca v každom zázname musí byť unikátna,

3. `CHECK` — zadaný výraz musí platiť.

Unikátnosť n-tice hodnôt sa dá vyžiadať vytvorením unikátneho indexu nad danými stĺpcami [1, s. 374]. Primárne kľúče sa z nášho pohľadu skladajú z dvoch obmedzení — `NOT NULL` pre jednotlivé stĺpce a `UNIQUE` pre celý kľúč. Obe tieto obmedzenia vieme priamočiaro kontrolovať, aby sme zabezpečili integritu výstupu.

Možnosť `CHECK` je zložitejšia. Spája sa s výrazom v jazyku SQL a hranice toho, čo je dovolené, závisia od implementácie databázového systému. Ak by sme chceli kontrolovať integritu aj pre tento typ obmedzenia, museli by sme tieto výrazy vedieť interpretovať.

2.2.2 Textové formáty štruktúrovaných dát

Náš systém bude prijímať dáta v rôznych formátoch. Štruktúry, ktoré dokážu reprezentovať, sa medzi nimi môžu líšiť. Hiearchické formáty ako JSON, YAML alebo XML nekódujú iba vzťahy tabulárneho významu. Z toho vyplýva, že je nutné obmedziť množinu prijímaných dát v týchto formátoch. V nasledujúcich častiach tieto formáty presnejšie rozoberieme. Zadefinujeme sady určitých obmedzení. Tieto obmedzenia sa však môžu zdať ako arbitrárne. Ak by to užívateľovi nevyhovovalo, ponúkneme spôsob, ako systém jednoducho rozšíriť.

CSV

CSV je jednoduchý formát na reprezentáciu tabulárnych dát. Hodnoty sú oddelené čiarkou, môžu byť ohraničené úvodzovkami. Záznamy sú oddelené znakom nového riadku. Prvý záznam môže definovať názvy stĺpcov [3].

Jeden CSV súbor predstavuje iba jedinú tabuľku. Neobsahuje žiadne ďalšie metadáta. Prvý riadok budeme interpretovať vždy ako názvy stĺpcov. Ak by sme všetky hodnoty považovali za reťazec, nebol by náš dátový generátor príliš užitočný. Rozhodli sme sa teda vyžadovať od reťazcov, aby boli ohraničené úvodzovkami. Ostatné hodnoty sú interpretované ako čísla s desatinnou časťou. Ak-

Výpis kódu 5 Príklad akceptovaného súboru vo formáte CSV.

```
"Name","Age","Height"
"Alica",23,1.85
"Robert",21,1.77
"Cyril",22,1.92
```

ceptovaný formát uvádzame vo výpise 5. Ide o jednoduchý spôsob ako efektívne rozlíšiť reťazce od čísel bez nutnosti zavádzania komplikovaných pravidiel.

Budeme musieť zaviesť ešte jednu požiadavku. Typy hodnôt v rámci jedného stĺpca musia byť zhodné. To znamená, že stĺpec obsahuje buď reťazce alebo čísla. Každý súbor v takto definovanom formáte je validný CSV súbor. Pri tvorbe formátu sme sa inšpirovali štandardnou knižnicou jazyka Python.[10]

Pri číslach ešte môžeme rozlíšiť celočíselnosť. Zavedieme jednoduché pravidlo — ak sú všetky hodnoty v stĺpci celočíselné, ide o stĺpec s celými číslami. V opačnom prípade je jeho typ číslo s desatinnou časťou. V príklade 5 je stĺpec `Age` celočíselný. Ak by sme však pridali osobu s vekom `0.5`, stĺpec by mal typ desatinného čísla.

Existuje štandardný spôsob ako pridať ďalšie informácie pomocou metadátového súboru vo formáte JSON [4]. Je možné definovať primárne kľúče, cudzie kľúče naprieč súbormi, určenie reprezentácie hodnoty `null`, reprezentáciu pravdivostných hodnôt, obmedzenie na unikátnosť hodnoty v stĺpci a ďalšie integritné obmedzenia ako formáty alebo minimálne a maximálne hodnoty.

V týchto schémach vidíme jasné paralely so schémami relačných databáz. Môžeme sa tým inšpirovať pri návrhu všeobecných schém, spoločných pre databázy aj textové formáty.

Nevýhoda schém je, že ich musí niekto vytvoriť. Ak užívateľ neobdržal dáta spolu so schémou, nie je užívateľsky prívetivé žiadať ho, aby ju dodatočne napísal. Lepšou alternatívou by bolo ponúknuť systém úpravy schémy v grafickom užívateľskom prostredí, čo by bolo aj všeobecne využiteľné — nielen pre formát CSV.

JSON

JSON je textový formát pre výmenu štruktúrovaných dát medzi programovacími jazykmi. *JSON hodnoty* sú objekty, polia, čísla, reťazce, a tokeny `true`, `false`, `null`. Objekt je sada párov kľúča a JSON hodnoty. Pole je zoznam JSON hodnôt [5].

Zadefinujme jednoduchú podmnožinu dát v tomto formáte. Jeden riadok v tabuľke nech je reprezentovaný jedným objektom. Názvy atribútov zodpovedajú názvom stĺpcov tabuľky, ich hodnoty sú potom konkrétne hodnoty daného záznamu — reťazec, číslo, `true`, `false` a `null`. Očakávame, že tieto objekty sú organizované v poli. Uvádzame príklad spĺňajúci definíciu vo výpise 6.

Tento spôsob považujeme za prirodzený spôsob zachytávania tabulárnych záznamov. Nie je to síce jediná možnosť, neobsahuje však žiadne prekvapenia. Jeden takýto súbor nám dá záznamy tabuľky, názvy jej stĺpcov, čiastočne aj ich

[10] `https://docs.python.org/3/library/csv.html#csv.QUOTE_NONNUMERIC`

Výpis kódu 6 Definícia jednej tabuľky pomocou obmedzeného formátu JSON. Príklad obsahuje 3 konzistentné záznamy — majú zhodné názvy stĺpcov a kompatibilné typy hodnôt.

```
[
    {"name": "Alica",  "age": 23,   "likesIceCream": true},
    {"name": "Robert", "age": null, "likesIceCream": false},
    {"name": "Cyril",  "age": 22,   "likesIceCream": null}
]
```

typy. Podľa špecifikácie [5] nedokážeme pri `age` rozlíšiť, či ide o celé číslo alebo desatinné. Rovnako ako v prípade súboru CSV určíme celočíselnosť podľa toho, či sú všetky čísla v danom stĺpci celočíselné. V prípade stĺpca `age` teda ide o celočíselný stĺpec.

Budeme vyžadovať ešte ďalšie obmedzenia, ktoré sú prirodzené pre tabulárnu reprezentáciu dát.

- konzistentnosť stĺpcov — v každom zázname musí byť rovnaká množina kľúčov,
- konzistentnosť typov — hodnoty v rámci jedného stĺpca musia byť rovnakého typu

Požiadavka na konzistentnosť typov má niekoľko okrajových prípadov:

- typ hodnoty `null` je konzistentný s každým iným typom,
- `true` a `false` považujeme za hodnoty rovnakého typu,
- celé a desatinné čísla sú navzájom konzistentné.

V jednom súbore dovolíme definovať aj viacero tabuliek. JSON hodnota na najvyššej úrovni je objekt a jeho kľúče interpretujeme ako názvy tabuliek. Náležiace hodnoty budú záznamy príslušnej tabuľky spĺňajúce vyššie vymenované obmedzenia. Vzorové dáta uvádzame vo výpise 7. Tento formát ponúkneme aj ako formát vygenerovaných dát. Ako vstup budeme prijímať oba formáty a automaticky medzi nimi rozlíšime.

2.2.3 Spoločná reprezentácia

Ukázali sme vybrané typy dátových zdrojov, ktoré implementujeme v tejto práci. Budú súčasťou systému nezávislého na fungovaní konkrétneho zdroja. To umožní pridať ďalšie podporované formáty alebo iné typy databáz. Aby sme

Výpis kódu 7 Príklad definície viacerých tabuliek v jednom JSON súbore.

```
{
    "flight": [
        {"from": "VIE", "to": "BCN"},
        {"from": "BCN", "to": "VIE"}
    ],
    "airport": [
        {"code": "VIE", "city": "Vienna"},
        {"code": "BCN", "city": "Barcelona"}
    ]
}
```

tento cieľ dosiahli, potrebujeme zvoliť spoločnú reprezentáciu schém a dát, s ktorým bude pracovať zvyšok systému.

Na začiatku sekcie sme určili, že pracujeme s tabulárnymi dátami — schémy pozostávajú z tabuliek a ich stĺpcov. V nasledujúcom texte vytvoríme delenie dátových typov. Tiež je nutné zovšeobecniť typy integritných obmedzení.

Dátové typy

Prezentované formáty sa líšia delením dátových typov. CSV nerobí žiadne delenie. JSON pozná okrem reťazcov aj desatinné čísla a pravdivostné hodnoty. Najjemnejšie delenie ponúkajú databázové systémy, hoci medzi implementáciami existujú rozdiely [1, s. 363].

Vybrali sme nasledujúcu sadu dátových typov:

- celé čísla — `INTEGER` v databázach, stĺpce zo súborov JSON a CSV s výlučne celočíselnými hodnotami,
- čísla so zlomkovou časťou — typ ostatných číselných hodnôt v JSON a CSV, `NUMBER` a `DECIMAL` v databázach,
- pravdivostné hodnoty — typ hodnôt `true` a `false` vo formáte JSON,
- reťazce — reťazce z JSON a CSV, reťazce fixnej a premennej dĺžky v databázových systémoch,
- dátumy — dátum a čas, zodpovedá typu `DATETIME` alebo `DATE` [1, s. 364].

V prípade formátu CSV sme určili, že hodnoty v úvodzovkách sú reťazce a všetko ostatné sú celé alebo desatinné čísla. Formát JSON tiež nerozlišuje celočíselnosť [5]. V oboch prípadoch určíme typ stĺpca ako celé číslo, ak všetky číselné hodnoty v ňom majú zlomkovú časť vynechanú alebo nulovú.

Pri číslach so zlomkovou časťou sa nebudeme zaoberať presnosťou reprezentácie. V kontexte dátového generátora by to bolo zbytočné, keďže generované dáta sú náhodné a stratová reprezentácia nie je závadou. Pravdivostné hodnoty môžu mať svoj typ aj v databázových systémoch.[11]

V určitých situáciách budeme musieť formátovať dátum ako reťazec. Pri komunikácii s užívateľom využijeme formát `YYYY-MM-DD HH:MM:SS` alebo `YYYY-MM-DD`, keď je časová zložka irelevantná. Tento formát je inšpirovaný štandardnou reprezentáciou dátumu a času.[12] Využijeme ho aj pri výstupe do súborov CSV a JSON. Pri vkladaní do databázy je formátovanie v kompentencii adaptéra príslušného databázového systému.

Naša dátová sada zjavne nie je vyčerpávajúca. Vynechali sme napríklad reprezentáciu dátumu (bez časovej zložky) a času (bez dátumovej zložky). Zaujímavé by bolo tiež pridať JSON ako ďalší dátový typ. Vynahradíme to však rozšíriteľnou implementáciou, takže pridať ďalší typ bude priamočiare.

Integritné obmedzenia

V časti 2.2.1 sme predstavili primárne kľúče, cudzie kľúče a ďalšie typy integritných obmedzení. Nadviazali sme na to v časti 2.2.2, keď sme ukázali podobné typy obmedzení definované pre formát CSV. Tieto obmedzenia musia byť zohľadnené dátovým generátorom usilujúcim sa o zabezpečenie integrity výstupných záznamov.

Zvolili sme nasledujúce typy integritných obmedzení nezávislých na spôsobe reprezentácie dát:

- primárny kľúč (`PRIMARY KEY`),
- cudzí kľúč (`FOREIGN KEY`),
- unikátnosť n-tice stĺpcov v tabuľke (`UNIQUE`),
- stĺpec bez prázdnej hodnoty (`NOT NULL`).

Každé z týchto obmedzení má svoj ekvivalent v relačných databázových systémoch. V metadátach CSV súboru [4] neexistuje obmedzenie na unikátnosť n-tice, ale zložený primárny kľúč má podobný efekt. Ako sme už skôr naznačili, tieto obmedzenia sú priamočiaro kontrolovateľné. Vynechali sme obmedzenie `CHECK` z databázových systémov — práve kvôli náročnosti jeho kontroly.

[11] `https://www.postgresql.org/docs/current/datatype-boolean.html`
[12] `https://en.wikipedia.org/wiki/ISO_8601`

Kapitola 3

Špecifikácia

V tejto kapitole na základe analýzy zadefinujeme požiadavky kladené na našu aplikáciu. Pre maximálnu pohodlnosť chceme užívateľovi dať viac možností, ako s aplikáciou interagovať. Tieto možnosti budú zahŕňať grafické a programátorské prostredia. Tým pokryjeme viacero možností použitia. Potenciálni užívatelia ich zrejme budú aj kombinovať.

Grafické rozhranie by malo byť centrom našej aplikácie. Jeho výhoda je, že nie je nutné vytvárať špeciálne manuály, aby s ním vedel byť užívateľ produktívny. Rola ďalších rozhraní ako webové API a CLI bude najmä pri automatizácii.

Na základe analýzy ostatných softvérových riešení sme sa rozhodli grafické prostredie umiestniť na web. Skombinujeme v ňom výhody existujúcich webových a natívnych riešení. Z webových riešení prevezmeme jednoduchosť a nízku bariéru k začatiu používania. Užívateľ sa okamžite ocitne na stránke, kde môže byť produktívny. Podobne ako natívne aplikácie budeme načítavať štruktúru databázy. Náš nástroj bude navyše schopný čítať štruktúru tabulárnych dát v textovom formáte. Rovnako ponúkneme viacero výstupných formátov.

3.1 Webová aplikácia

Voľba webového rozhrania má niekoľko zásadných implikácií. Zjavnými výhodami sú, že užívateľ si nebude musieť nič sťahovať. Minimalizujeme tým bariéru začatia používania nášho systému. Zachytíme tým najviac potencionálnych používateľov. Dostávame tým aj multiplatformové riešenie — konečnému používateľovi stačí iba webový prehliadač.

Nevýhodou webového riešenia je, že sa z nášho backendu nedokážeme pripojiť do databáz, ktoré nie sú verejne prístupné. Tiež je možné, že užívateľ s nami nechce zdieľať citlivé informácie. Kvôli tomu musíme poskytnúť možnosť, aby si užívateľ spustil systém na vlastnom hardvéri.

Máme na výber medzi multi-page aplikáciou alebo single-page aplikáciou.[1] V prvom prípade by sme používali template engine.[2] Kód medzi backendom a frontendom by nebol jasne oddelený. Každá užívateľská akcia by si vyžadovala nové načítanie stránky. Webovú aplikáciu preto postavíme ako single-page. Vďaka tejto technológii ľahšie vytvoríme plynule pôsobiacu stránku. Vytvorenie API, cez ktorú budeme sprostredkovávať všetky užívateľské požiadavky, si zrejme bude žiadať viac práce. Vznikne nám však jasne definované rozhranie medzi webovým serverom a klientom. Toto rozhranie bude využiteľné aj inými aplikáciami.

3.1.1 Užívateľ a projekt

Žiadame, aby naše riešenie zvládalo obslúžiť viacero užívateľov súčasne. Keď sa užívateľ ocitne na našej stránke, potrebujeme ho vedieť identifikovať. Ďalšia požiadavka je, aby sa užívateľ mohol ku svojej práci vrátiť aj po zatvorení prehliadača. Tiež by mal mať možnosť pokračovať vo svojej práci na inom zariadení. Musíme teda vytvoriť systém registrácie a prihlasovania.

Pre rýchle testovanie by nemuselo byť ideálne nútiť nového návštevníka o vyplnenie údajov. Mali by sme ponúknuť spôsob, ako sa hneď dostať do vnútra systému. Vo vnútri to môžeme implementovať ako registráciu užívateľa bez zadaných prihlasovacích údajov. Počas evaluácie by návštevník mal mať možnosť stať sa riadnym užívateľom s ponechaním vykonaných zmien.

Užívateľ by mal mať tiež možnosť si oddeliť svoju prácu. Inštanciu práce užívateľa sme nazvali projekt. Každý projekt bude označený názvom a bude obsahovať vlastnú sadu dátových zdrojov a tabuliek. Budeme potrebovať komponent zobrazujúci zoznam projektov patriacich užívateľovi. Súčasťou by mala byť možnosť vytvoriť nový projekt, premenovať ho alebo zmazať. Pri vyššie popísanej rýchlej registrácii by jeden projekt mal byť vytvorený automaticky.

V ďalších častiach budeme opisovať požiadavky na funkcionalitu týkajúcu sa jedného projektu.

3.1.2 Dátové zdroje

Na začiatku prijmeme od užívateľa zdroje referenčných dáta, ktoré určujú vlastnosti generovaných dát. Prvý komponent týkajúci sa projektu bude preto spravovať dátové zdroje. Základné operácie sú pridanie a zmazanie. Z dátových zdrojov čerpáme referenčné dáta, respektíve ich schémy. Na pokyn užívateľa načítame z dátového zdroja tabuľky, stĺpce a integritné obmedzenia. Po načítaní tabuliek by

[1] `https://en.wikipedia.org/wiki/Single-page_application`

[2] `https://en.wikipedia.org/wiki/Template_processor`

sme mali dostať takú konfiguráciu, aby bolo okamžite možné generovať zmysluplné dáta.

Analyzované natívne riešenia umožňujú pridať práve jednu databázu. Webové riešenia s databázovým systémom nespolupracujú. Náš softvér by mal byť všeobecnejší. Nebude povinné pridávať žiadny zdroj a dovolíme ich pridať aj viac ako jeden. Navyše sa nebudeme obmedzovať iba na relačné databázy. Budeme podporovať niekoľko typov dátových zdrojov:

- relačná databáza — napríklad PostgreSQL a ďalšie,
- textový súbor — CSV, JSON a ďalšie,
- relačná databáza v súbore — SQLite.[3]

Relačnú databázu pridáme vyplnením prístupových údajov. Súbory, vrátane SQLite, musia byť nahraté na server. Užívateľovi by sme mali dať šancu ich stiahnuť naspäť. Súbory obsahujúce dáta vo formátoch CSV alebo JSON musia spĺňať požiadavky určené v časti 2.2.2.

Mali by sme myslieť aj na situácie, keď už v projekte existujú nejaké tabuľky a snažíme sa importovať schému. Podporovať budeme kombinovanie tabuliek načítaných z viacerých zdrojov. Zmysel dáva aj viacnásobný import z jedného zdroja — najmä ak sa zmenila pôvodná schéma. V takom prípade musíme zapracovať zmeny do projektovej schémy.

V prípade databáz bude môcť byť dátový zdroj využitý aj ako cieľ výstupu. Tento prístup je úplne iný ako čokoľvek, čo sme videli u iných existujúcich riešení. Natívne riešenia sa obmedzovali na jednu zdrojovú a zároveň cieľovú databázu v jednom projekte. Nami navrhnuté riešenie umožňuje kombinovať mnohé zdroje vstupu a ciele výstupu.

Môže sa naskytnúť situácia, že užívateľ má jednu databázu s produkčnými dátami. Chce naplniť testovaciu databázu syntetickými dátami po vzore produkčných. V takom prípade pridá obe databázy do projektu ako dátové zdroje. Z produkčnej importuje schému a vygenerovanými dátami naplní testovaciu databázu. Existujúce aplikácie robia tento proces zbytočne komplikovaný.

3.1.3 Tabuľky, stĺpce a generátory

Na jednej stránke by sme mali mať zoznam tabuliek v projekte. Toto je súčasťou každého dátového generátora, hoci niektoré neoddeľujú projekty alebo pracujú len s jedinou tabuľkou. V našom prípade môžu tabuľky vzniknúť importom z dátového zdroja. Tiež by sme mali mať možnosť tabuľky vytvárať manuálne. K tomu

[3] `https://www.sqlite.org/index.html`

patrí aj editácia a odoberanie tabuliek. Týmto prístupom kombinujeme vlastnosti analyzovaných natívnych a webových riešení. Prvá menovaná kategória umožňovala iba import tabuliek, druhá kategória iba manuálnu správu.

Tabuľke prináleží zoznam integritných obmedzení, ktoré načítavame z dátových zdrojov. Tieto obmedzenia by mali byť zobrazené spolu tabuľkou. Pre užívateľa sú to relevantné informácie, pretože môžu ovplyvniť jeho voľbu generátora.

Súčasťou tabuľky sú stĺpce. Pri manuálnej tvorbe schémy by sme mali mať možnosť ich pridávať, editovať a mazať. Dôležitá informácia pri každom stĺpci je nielen jeho názov, ale aj dátový typ či obmedzenie `NOT NULL`. Toto všetko by malo byť zobrazené spolu s tabuľkou.

Stĺpcom prináleží generátor. Na tejto stránke by sme mali mať možnosť zmeniť voľbu generátora pre stĺpec. Zobrazovať by sa mali iba generátory relevantné pre daný dátový typ, zotriedené do kategórií. Ku generátoru sa viaže sada parametrov. Tieto parametre môžu byť rôznych dátových typov. Hodnoty parametrov musia byť po zmene ukladané.

Niektoré softvérové riešenia umožňujú napísať výraz alebo krátky kus kódu, ktorý generuje hodnoty závisiace na hodnotách iných stĺpcov. Aby sme toto podporovali, museli by sme buď implementovať vlastný jazyk a interpreter či implementovať existujúci jazyk a vyriešiť možné bezpečnostné riziká. Toto by bola veľká úloha pre prácu nášho rozsahu. Miesto toho sa budeme sústrediť na zjednodušenie vnútorných rozhraní, aby bolo možné pridávať vlastné generátory. Dosiahnuť sa tým dá podobný výsledok, akurát užívateľ si bude musieť spustiť vlastný server.

Každému importovanému stĺpcu automaticky pridelíme stĺpcový generátor. Toto je v súlade s cieľom maximálnej automatizácie. Generátor bude nakonfigurovaný v závislosti od vstupných dát, aby sa jeho výstup podobal vzoru. Musíme pokryť dostatočné množstvo generátorov, pokrývajúce každý zo základných dátových typov. Na rovnakom princípe fungujú analyzované natívne riešenia.

3.1.4 Náhľad

Užívateľovi ukážeme náhľad generovaných dát. Na základe tohto výstupu môže užívateľ ďalej prispôsobovať jednotlivé stĺpcové generátory. Táto vlastnosť bola implementovaná v oboch natívnych a jednom webovom riešení, ktoré sme analyzovali. V prípade SQL Data Generator 4 neboli viditeľné dáta generované databázovým systémom. Vždy bola viditeľná iba jedna tabuľka súčasne. Za lepší prístup považujeme, keď každý stĺpec v náhľade obsahuje reálne dáta.

Náhľad môžeme implementovať ako samostatnú stránku. Nevýhoda je, že zobrazenie náhľadu si bude žiadať kliknutie od užívateľa. Ušetríme tým však priestor na stránke s tabuľkami a stĺpcami. Tiež je potrebné zabezpečiť obnovenie náhľadu po zmene nastavení. Vďaka tomu, že to bude samostatná stránka, stačí nám

dáta generovať práve vtedy, keď užívateľ má záujem ich vidieť. Nebudeme tým zahlcovať server s požiadavkami pri každej zmene parametrov. Užívateľ získa prehľad o celej databáze a nie len o jednej tabuľke.

3.1.5 Export

Keď sú tabuľky a generátory nastavené k spokojnosti užívateľa, ponúkneme niekoľko možností výstupu. Užívateľ si vyberie, pre ktoré tabuľky si želá vytvárať dáta a zadá želaný objem dát. Možnosti výstupu zahŕňajú vypĺňanie dát priamo do niektorej z pripojených databáz, súbor s príkazmi `INSERT` alebo dáta vo formáte CSV či JSON. Podrobnejšia podoba výstupných CSV a JSON súborov bola opísaná v časti 2.2.2. Ďalšia možnosť je vytvoriť nový súbor s databázou SQLite, kde sa systém postará o vytvorenie schémy a následne do nej vloží záznamy. Musíme navyše zabezpečiť, aby generovanie bolo replikovateľné. Vykonávame pritom kontrolu integritných obmedzení.

Širokým spektrom možností výstupu sa odlišujeme od existujúcich softvérových riešení. Každé z nich ponúkalo buď výstup do databázy alebo rôzne textové formáty. My ponúkneme kombináciu oboch.

Konfiguračný súbor pre CLI

Ako ďalšiu možnosť ponúkneme export konfigurácie do konfiguračného súboru. V takom prípade generovanie dát neprebehne. Pomocou programu, ktorý vyvinieme, bude možné z príkazového riadku vygenerovať dáta vo všetkých formátoch ponúkaných v grafickom prostredí. Ide o podobný princíp, ako sme videli u natívnych riešení.

V časti 3.1.2 sme špecifikovali, že naše riešenie spolupracuje aj s viacerými databázami. Toto by mala podporovať aj CLI aplikácia. Jedna možná alternatíva by bola zahrnúť prístupové údaje k pridaným dátovým zdrojom do konfiguračného súboru. Užívateľ by si pri generovaní z CLI vyberal jeden z týchto zdrojov.

Existuje všeobecnejší a bezpečnejší prístup. Konfiguračný súbor nebude obsahovať žiadne prístupy k databázam. Miesto toho budeme prístupové údaje do výstupnej databázy prijímať pri spustení generovania. Týmto dosiahneme, že prístupy do výstupnej databázy nemusia byť zadávané vo webovom prostredí. Navyše môžeme z jedného konfiguračného súboru napĺňať akúkoľvek databázu.

3.2 Programátorské rozhrania

Musíme pokryť použitia, keď užívateľ chce zapojiť naše riešenie do inej aplikácie alebo spúšťať ho z iného programu. Obe webové riešenia ponúkali REST API. My

Výpis kódu 8 Príklady použitia rozhrania príkazového riadku. Prvý argument určuje formát výstupu. Ako druhý argument prijmeme cestu ku konfiguračnému súboru. Tretí argument závisí od akcie. V prípade databázy je to prístupový reťazec, v prípade súborov cesta k výstupnému súboru.

```
# gen.py action configuration [destination]
gen.py json project.json
gen.py json project.json output.json
gen.py zip project.json output.zip
gen.py script project.json output.sql
gen.py insert project.json sqlite:///database.sqlite
gen.py sqlite project.json output.db
```

môžeme využiť rozhranie API, ktoré vyvinieme pre pokrytie potrieb grafického rozhrania.

V časti 3.1.5 sme opísali princíp vytvárania konfiguračného súboru v grafickom prostredí. Umožní nám to vyvinúť CLI aplikáciu generujúcu záznamy na základe tohto súboru. Tým umožníme rovnaké spôsoby použitia ako ponúkali obe natívne riešenia.

Venujme sa teraz špecifikácii rozhrania našej CLI aplikácie. Nemá zmysel poskytovať celú funkcionalitu systému v CLI. Nerobili to ani ostatné analyzované riešenia. Pre projekt nášho rozsahu by bolo zbytočné venovať sa dvom rozhraniam (REST a CLI), ktoré ponúkajú zhodnú funkcionalitu. Ak má užívateľ takéto špeciálne použitie, že potrebuje z nejakého programovacieho jazyka meniť konfiguráciu, môže použiť API. Otvorená je aj možnosť použiť naše riešenie ako knižnicu.

Úloha CLI aplikácie je prečítať konfiguračný súbor a podľa neho generovať dáta. Mali by sme ponúkať všetky možné formy výstupu ako grafické prostredie. Tento formát nech je určený prvým pozičným argumentom. Ako druhý argument prijmeme cestu ku konfiguračnému súboru. Ďalej môžeme dostať špecifikáciu, kam má byť výsledok vypísaný — teda cestu k výstupnému súboru alebo prístup do databázy. Prístup do databázy môžeme elegantne podať vo forme URL.[4] Ak nedostaneme špecifikáciu výstupu, budeme vypisovať na štandardný výstup. Príklady použitia tohto rozhrania uvádzame vo výpise 8.

[4] `https://docs.sqlalchemy.org/en/14/core/engines.html`

Kapitola 4

Design

Na implementáciu aplikácie podľa špecifikácie zvolíme vhodné programovacie jazyky a knižnice. Prejdeme dostupnými alternatívami a podrobne zdôvodníme naše voľby. Každú z vybraných knižníc krátko predstavíme a vysvetlíme, ako bude zapadať do zvyšku systému. Potom analyzujeme problémy súvisiace s generovaním dát. Porovnáme možnosti riešenia a popíšeme vybrané algoritmy.

4.1 Voľba technológie

Logiku generátora implementujeme v jazyku Python. Nad týmto modulom bude stáť tenká vrstva sprístupňujúca celú funkcionalitu ako REST API. Časť funkcionality bude sprístupnená cez CLI. Vyrobíme webovú aplikáciu na platforme Angular, ktorá bude nasaditeľná ako webová služba.

4.1.1 Programovací jazyk

Python sme zvolili na implementáciu jadra a webového servera. Očakávame od neho najmä vysokú produktivitu a široký výber kvalitných nástrojov.

Medzi najväčšie prednosti Pythonu patrí množstvo knižníc v rôznych oblastiach. Vieme si vybrať z niekoľkých vyspelých webových frameworkov (Django, Flask), knižníc pre správu databáz (SQLAlchemy, peewee, Django ORM) či nástroje k testovaniu (pytest, unittest) Môžu nám pomôcť aj nástroje na spracovanie dát (numpy, pandas), prípadne by sme mohli zapojiť strojové učenie (sklearn, tensorflow, pytorch). Veľkou výhodou je aj bohatá štandardná knižnica, ktorá nám poskytne napríklad nástroje na bezpečné hešovanie hesiel (hashlib) alebo prostriedky na prácu so súbormi vo formátoch json, csv a zip.

Jazyky s manuálnou správou pamäti ako C, C++ alebo Rust nepovažujeme za vhodný nástroj pre náš problém. Predpokladáme, že úzke hrdlo vo výkone

Výpis kódu 9 Príklad generickej triedy s typovými anotáciami.

```
from abc import ABC, abstractmethod
from typing import Generic, TypeVar

OutputType = TypeVar('OutputType')

class Generator(Generic[OutputType], ABC):
    @abstractmethod
    def make_value(self) -> OutputType:
        pass

class HelloGenerator(Generator[str]):
    def make_value(self) -> str:
        return 'Hello'
```

nášho systému bude rýchlosť databázového systému, disku alebo siete. Z väčšej kontroly nad hardvérom by sme teda nemali výrazné výhody, naopak by sme nechtiac mohli spôsobiť viac chýb.

Medzi ďalšie významné výhody Pythonu patrí multiplatformovosť. Keďže je to jazyk interpretovaný, nie je potrebná kompilácia. Významnou súčasťou nášho riešenia bude, že pokročilý užívateľ si bude môcť doplniť svoje generátory. To bude spočívať dopísaním kódu, ktorý sa automaticky a okamžite prejaví. Oproti kompilovaným jazykom je to flexibilnejšie.

Nevýhodou je slabý typový systém. Táto vlastnosť nám zhoršuje čitateľnosť a udržiavateľnosť celého riešenia. Tento problém čiastočne minimalizujeme pridávaním typových anotácií. Hoci nám zo strany interpreteru nie sú poskytované žiadne garancie navyše, typovú kontrolu vieme spustiť pomocou externých nástrojov (mypy).

Python nám dáva možnosť pomocou kódu získavať informácie o objektoch a modifikovať ich. Príkladom využitia týchto vlastností sú dekorátory. Nájdeme aj ďalšie využitia. Umožňujú nám písať ľahko čítateľný, expresívny kód.

Typové anotácie

Typové anotácie umožňujú vytvárať aj zložitejšie konštrukcie bežné v staticky typovaných jazykoch. Pomocou štandardného modulu *typing* dokážeme definovať napríklad generické typy.

V príklade 9 sme vytvorili generickú triedu `Generator` s typovou premennou `OutputType`. Má jednu abstraktnú metódu, vracajúcu hodnotu typu `OutputType`. Z triedy `Generator[str]` sme odvodili triedu `HelloGenerator`.

Užitočná vlastnosť, ktorú využijeme pri budovaní rozhrania pre generátory je

Výpis kódu 10 Čítanie typovej anotácie v REPL prostredí. Nadväzuje na predchádzajúci príklad.

```
>>> from inspect import signature
>>> sig = signature(HelloGenerator.make_value)
>>> sig.return_annotation
<class 'str'>
```

Výpis kódu 11 Jednoduchý endpoint pomocou dekorátora vo Flasku. Nachádza sa na adrese `/user` a odpovedá na HTTP GET požiadavku. Vrátený objekt bude serializovaný do formátu JSON. Neskôr tento príklad rozšírime o API dokumentáciu a pravidlá serializácie.

```
from flask import Flask
app = Flask(__name__)

@app.route('/user')
def get_user():
    return {'id': 1, 'email': 'foo@bar.com'}
```

možnosť čítať typové anotácie. Túto funkcionalitu zabezpečuje štandardný modul `inspect`. Uvádzame príklad použitia vo výpise 10.

4.1.2 Jadro a webový server

Jadrom označíme kód sprístupňujúci vlastnosti dátového generátora. Túto časť aplikácie vyrobíme nezávislú na tom, z akého prostredia bude používaná. Ako dôsledok bude tento modul využiteľný aj ako súčasť inej aplikácie, ľahšie testovateľný a nebude zložité nad ním postaviť ďalšie rozhrania.

Nad jadrom bude stáť webový server. Potrebujeme od neho, aby HTTP požiadavky prekladal na volania do jadra. Obľúbenými možnosťami medzi profesionálnymi vývojármi sú *Flask*[1] a *Django*.[2][3] Django je komplexnejšie riešenie, ktoré ponúka napríklad aj vlastnú knižnicu pre prácu s databázou. Naopak Flask je minimalistický framework, ktorý za nás túto voľbu nespravil. Pri našom riešení potrebujeme vysokú kontrolu nad databázovými systémami. Za tento aspekt bude zodpovedné jadro, ktoré má byť nezávislé na webovej technológii nad ním. Preto volíme Flask.

[1] `https://flask.palletsprojects.com/en/1.1.x/`

[2] `https://www.djangoproject.com/`

[3] `https://insights.stackoverflow.com/survey/2020#technology-web-frameworks-professional-developers2`

Výpis kódu 12 Príklad definície schémy v marshmallow. Schému vytvoríme odvodením z triedy `Schema`. Dekorátor `@post_load` označuje metódu, ktorá sa zavolá po deserializácii. Využijeme ju na vytvorenie objektu, s ktorým pracujeme v Pythone.

```
from dataclasses import dataclass
from marshmallow import Schema, post_load
from marshmallow.fields import Int, Str
from marshmallow.validate import Range

@dataclass
class ExportRequisitionRow:
    table_name: str
    row_count: int
    seed: int

class ExportRequisitionRowView(Schema):
    table_name = Str()
    row_count = Int(validate=Range(min=1))
    seed = Int()
    @post_load
    def make_row(self, data, **kwargs):
        return ExportRequisitionRow(**data)
```

Vo Flasku vytvárame endpointy pomocou dekorovaných funkcií. Vo výpise 11 uvádzame minimalistický príklad. Týmto spôsobom postavíme celú aplikáciu. Každá akcia bude mať svoj endpoint.

Serializácia a deserializácia

Formátom výmeny dát medzi webovým serverom a klientom bude JSON. Potrebujeme presne zadefinovať, aká štruktúra týchto objektov je jednotlivými endpointami prijímaná, aké objekty vracajú a tiež musíme vedieť overiť správnosť týchto objektov. Pomôže nám s tým knižnica *marshmallow*.

Výpis 12 uvádza reálny príklad, keď od užívateľa spracovávame požiadavku na generovanie dát. Táto požiadavka je reprezentovaná na strane Pythonu objektom triedy `ExportRequisitionRow`. Definovaná schéma určuje, ktoré jeho atribúty sa majú ocitnúť v serializovanom objekte. Rovnako túto schému využijeme na deserializáciu. V takom prípade marshmallow kontroluje zhodu dátových typov. Pridali sme validačné pravidlo, že užívateľ musí žiadať o aspoň jeden riadok. Ak vstup od užívateľa nevyhovuje schéme, marshmallow naformátuje zrozumiteľnú chybovú hlášku.

Po deserializácii vráti marshmallow objekt typu `dict`. V príklade 12 sme tiež pridali metódu, ktorá na konci deserializácie zo slovníka vytvorí inštanciu

Výpis kódu 13 Validácia, deserializácia a serializácia pomocou marshmallow schémy. Príklad je v REPL prostredí a nadväzuje na výpis 12.

```
>>> obj = {'table_name': None, 'row_count': 0, 'seed': 123}
>>> ExportRequisitionRowView().validate(obj)
{'row_count': ['Must be greater than or equal to 1.'],
'table_name': ['Field may not be null.']}
>>> obj = {'table_name': 'author', 'row_count': 9, 'seed': 3}
>>> ExportRequisitionRowView().load(obj)
ExportRequisitionRow(table_name='author',
                     row_count=9,
                     seed=3)
>>> server_obj = ExportRequisitionRow(table_name='book',
                                      row_count=100,
                                      seed=42)
>>> ExportRequisitionRowView().dump(server_obj)
{'seed': 42, 'row_count': 100, 'table_name': 'book'}
```

`ExportRequisitionRow`.

Validáciu spustíme metódou `validate` na inštancii schémy. Táto metóda vráti zoznamy chybových hlášok. Metódou `load` deserializujeme objekt prijatý od klienta. Metódou `dump` naopak serializujeme objekt, ktorý vrátime klientovi. Použitie týchto metód uvádzame vo výpise 13.

Špecifikácia API

Možnosti a vlastnosti nášho webového rozhrania popíšeme pomocou štandardu OpenAPI.[4] Tento popis bude slúžiť nielen ako dokumentácia, ale využijeme ju aj na automatické generovanie kódu. Knižnica *flasgger* nám umožní priamo k endpointom pridať definície parametrov a odpovedí.

Vo výpise 14 rozširujeme endpoint z príkladu 11. Pomocou dekorátora `swag_from` sme definovali možnú odpoveď s kódom 200, ktorá vracia informácie o prihlásenom užívateľovi. Na definíciu schémy užívateľa využijeme marshmallow schému. Rovnakú schému využijeme aj v tele funkcie na serializáciu skutočného objektu `g.user` držiacom všetky informácie o užívateľovi. Metóda `dump` vyberie z objektu žiadané atribúty, nevráti teda užívateľove (zahešované) heslo.

Po lokálnom spustení serveru dostávame interaktívne prostredie dokumentujúce aplikáciu na adrese `http://localhost:5000/apidocs/`. Strojovo čitateľná schéma je dostupná z `http://localhost:5000/apispec_1.json`. Druhú menovanú schému môžu využiť knižnice ako *ng-swagger-gen*.[5] Táto knižnica vy-

[4] `https://swagger.io/specification/v2/`

[5] `https://www.npmjs.com/package/ng-swagger-gen`

Výpis kódu 14 Rozšírenie endpointu vo Flasku z výpisu 11 o dokumentáciu a serializáciu. Objekt g môžeme využiť ako slovník, ktorého životnosť trvá počas spracovania jednej HTTP požiadavky. Vhodný je napríklad na uloženie inštancie prihláseného užívateľa. Pre stručnosť sú vynechané importy knižníc.

```
class UserView(Schema):
    id = Int()
    email = Str()

@app.route('/user')
@swag_from({
    'tags': ['Auth'],
    'responses': {
        200: {
            'description': 'Logged in user',
            'schema': UserView
        }
    }
})
def get_user():
    return UserView().dump(g.user)
```

generuje zo schém typovo anotované triedy, prostredníctvom ktorých budeme volať funkcie API.

4.1.3 Interakcia s databázou

V rámci nášho softvérového riešenia budeme využívať databázový systém na uloženie užívateľských nastavení. Pri správe databázy budeme teda riešiť niekoľko bežných problémov:

- definícia schémy databázy (tabuľky, stĺpce, kľúče, obmedzenia, indexy),
- mapovanie tabuliek na objekty v Pythone,
- čítanie, vkladanie, upravovanie a mazanie záznamov.

V doméne dátového generátora nám vznikajú ďalšie požiadavky. Vonkajšie databázové systémy, ku ktorým sa budeme dynamicky pripájať na pokyn užívateľa, budú slúžiť ako zdroj schém a dát. Tieto databázy môžu byť tiež cieľovým úložiskom generovaných dát. Potrebujeme pokryť nasledujúce operácie:

- čítanie schémy databázy,
- čítanie záznamov z databázy s ľubovoľnou schémou,

Výpis kódu 15 Príklad definície mapovaných objektov v SQL Alchemy.

```
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy import Column, Integer, String, ForeignKey
from sqlalchemy.orm import relationship

base = declarative_base()

class Project(base):
    __tablename__ = 'project'
    id = Column(Integer, primary_key=True)
    name = Column(String)
    user_id = Column(Integer, ForeignKey('user.id'))
    user = relationship('User', back_populates='projects')

class User(base):
    __tablename__ = 'user'
    id = Column(Integer, primary_key=True)
    email = Column(String, unique=True)
    projects = relationship('Project',
                            order_by=Project.id,
                            back_populates='user')
```

- vkladanie záznamov do databázy s ľubovoľnou schémou,
- zostavenie príkazov `INSERT`,
- vyššie definované operácie musia byť vykonateľné v rôznych databázových systémoch.

Vyberieme knižnicu, od ktorej očakávame, že nám čo najviac uľahčí vyššie zmienené problémy. Zjavne nie je výhodné pracovať priamo s databázovými adaptérmi pre jednotlivé databázové systémy. V takom prípade by sme mali veľa zduplikovaného kódu.

V jazyku Python prichádzajú do úvahy dve knižnice, ktoré podporujú viaceré databázové systémy a zároveň ponúkajú objektovo-relačné mapovanie — *peewee*[6] a *SQLAlchemy* [6]. SQLAlchemy je pre nás jasná voľba, pretože je to vyspelý projekt s omnoho väčším množstvom podporovaných databáz.

Objektovo-relačné mapovanie v SQLAlchemy

Vo výpise 15 zadefinujeme triedy dediace zo spoločnej bázy. Každá z nich zodpovedá jednej tabuľke v databáze. Triedové atribúty typu `Column` zodpovedajú stĺpcom. Entita projekt obsahuje cudzí kľúč odkazujúci na entitu užívateľa. Funkciou

[6] `https://github.com/coleifer/peewee`

Výpis kódu 16 Pripojenie k SQLite databáze v pamäti a vytvorenie schémy definovej vo výpise 15.

```
from sqlalchemy import create_engine
engine = create_engine('sqlite:///:memory:')
base.metadata.create_all(engine)
```

Výpis kódu 17 Príklad práce s ORM. Vytvárame užívateľa, viažeme s ním projekt a potvrdzujeme transakciu. Nadväzuje na príklad 16.

```
from sqlalchemy.orm import Session
session = Session(engine)
user = User(email='foo@bar.com')
session.add(user)
project = Project(name='Baz', user=user)
session.commit()
```

`relationship` zabezpečíme, že `project.user` bude inštanciou triedy `User`. Naopak `user.projects` bude zoznam užívateľských projektov.

Na ukážku sa môžeme pripojiť do databázy SQLite v pamäti a vytvoriť v nej vyššie definované tabuľky. Príklad ukazujeme vo výpise 16.

Pri práci s ORM potrebujeme vytvoriť objekt typu `Session` zviazaný s vyššie vytvoreným `engine`. `Session` v sebe drží mapované entity a otvorenú transakciu. Ukončíme ju volaním `commit` alebo `rollback` [6]. V príklade 17 vytvárame nového užívateľa a jeden s ním zviazaný projekt.

Čítanie z databázy prevedieme prostredníctvom objektu `Session`. Uvádzame príklad použitia vo výpise 18. Operácia `query` otvorí novú transakciu. SQLAlchemy sa postará o to, aby z databázy bol dočítaný asociovaný užívateľ. Keďže sme otvorili ďalšiu transakciu, musíme ju aj ukončiť príkazom `commit`.

Výpis kódu 18 Príklad čítania projektu z databázy. Nadväzuje na príklad 17.

```
project = session.query(Project).\
            filter(Project.name == 'Baz').\
            one()
print(project.user.email) # 'foo@bar.com'
session.commit()
```

Výber databázového systému

Venujme sa teraz voľbe vhodného databázového systému na ukladanie užívateľských dát. Vďaka knižnici SQLAlchemy môžeme zvoliť akýkoľvek z podporovaných systémov.[7]

SQLite má odlišné ciele v porovnaní s ostatnými podporovanými dialektami.[8] Jeho prednosti spočívajú v tom, že je „serverless“ a databáza sa nachádza v jedinom súbore. Výhodou je aj podpora na strane štandardnej knižnice Pythonu,[9] vďaka čomu nie je potrebné inštalovať ďalší ovládač. Ideálne využitie SQLite je pri integračných testoch. Na začiatku testu vytvoríme novú databázu (súbor) a po ukončení testu ju zmažeme.[10]

Keďže naša aplikácia bude podľa špecifikácie (3.1.1) obsluhovať viacerých užívateľov súčasne, lepšie riešenie je využiť systém s architektúrou client/server.[11] Rozhodli sme sa využiť PostgreSQL, pretože je open source.[12] Podobný výsledok by však bolo možné dosiahnuť aj s ostatnými podporovanými systémami.

4.1.4 Webová aplikácia

Webovú aplikáciu postavíme pomocou frameworku. Ten dá nášmu kódu štruktúru, ktorá bude zrozumiteľná pre každého programátora ovládajúceho danú technológiu. Vyhneme sa vymýšľaniu existujúcich riešení odznova a použijeme praxou overené knižnice.

Najvýznamnejšie technológie v tejto oblasti sú platforma *Angular*, knižnica *React* a framework *Vue*.[13] Všetky tri fungujú na princípe znovu-použiteľných komponentov s vnútorným stavom.[14] V kontexte nášho problému ich považujeme za ekvivalentné. Vyberáme si Angular, lebo s ním máme skúsenosti.

Komponenty

Komponent v Angulari definujeme dekorovanou triedou. Skrátený príklad komponentu predstavujúci hlavičku stránky uvádzame vo výpise 19. Táto hlavička bude zobrazovať parametrizovaný titulok a voliteľne môže obsahovať tlačidlo

[7] `https://docs.sqlalchemy.org/en/14/dialects/`

[8] `https://www.sqlite.org/about.html`

[9] `https://docs.python.org/3/library/sqlite3.html`

[10] `https://flask.palletsprojects.com/en/1.1.x/testing/`

[11] `https://www.sqlite.org/whentouse.html`

[12] `https://www.postgresql.org/`

[13] `https://insights.stackoverflow.com/survey/2020#technology-web-frameworks-professional-developers2`

[14] `https://angular.io/`, `https://reactjs.org/`, `https://vuejs.org/`

Výpis kódu 19 Príklad komponentu v Angulari. Importy sú pre stručnosť vynechané.

```
@Component({
    selector: 'app-page-header',
    templateUrl: './page-header.component.html',
    styleUrls: ['./page-header.component.scss']
})
export class PageHeaderComponent {
    @Input() title: string;
    @Input() sidenavButton = false;
    @Output() toggleSidenav = new EventEmitter();

    toggle() {
        this.toggleSidenav.emit();
    }
}
```

Výpis kódu 20 Implementácia šablóny pre komponent z výpisu 19.

```
<div *ngIf="sidenavButton" class="sidenav-button">
    <button type="button" (click)="toggle()">
        {{ title }}
    </button>
</div>
<h1 *ngIf="!sidenavButton" class="page-title">
    {{ title }}
</h1>
```

prepínajúce viditeľnosť menu. Dekorátor `@Input()` označuje vstupné vlastnosti. V našom prípade je to text v titulku a príznak určujúci, či má byť zobrazené tlačidlo. Tieto hodnoty môžeme využiť v šablóne komponentu. Dekorátor `@Output()` označuje výstupnú udalosť. Vytvorili sme ním objekt, prostredníctvom ktorého je možné reagovať na stlačenia tlačidla.

Kód vo výpise 20 ilustruje implementáciu šablóny. Vlastnosť `sidenavButton` sme využili v štruktúrnej direktíve `*ngIf`, ktorá zobrazí tlačidlo práve vtedy, keď je daná podmienka splnená. Text titulku čerpáme z vlastnosti `title`. Po kliknutí na tlačidlo je zavolaná metóda `toggle`. Táto metóda vygeneruje udalosť, na ktorú môže reagovať rodičovský komponent.

CSS pre triedy `sidenav-button` a `page-title` definujeme buď globálne alebo výlučne pre daný komponent. V druhom prípade Angular štýly zapúzdri tak, aby neovplyvňovali elementy mimo určeného komponentu.

Angular sleduje zmeny hodnôt vlastností komponentu a stará sa o jeho prekresľovanie. Tento komponent môžeme vložiť do šablóny iného komponentu po-

Výpis kódu 21 Inštanciácia komponentu v šablóne. Predávanie hodnôt do vstupných vlastností realizujeme atribútami so zhodnými názvami. Keď je názov atribútu v hranatých zátvorkách, pravá časť sa vyhodnotí ako výraz. Bez hranatých zátvoriek by sa pravá strana vyhodnotila ako reťazec. Oblými zátvorkami ohraničíme názov výstupnej vlastnosti. Do hodnoty atribútu dáme volanie, ktoré sa vykoná pri prijatí danej udalosti.

```
<app-page-header
    [title]="project?.name || 'Project'"
    [sidenavButton]="false"
    (toggleSidenav)="drawer.toggle()"></app-page-header>
```

Výpis kódu 22 Príklad na použitie dependency injection v komponente. O vytvorenie inštancie `AlertService` sa postará Angular (presnejšie `Injector`). Kľúčové slovo `public` je vlastnosť jazyka TypeScript. V tomto prípade je použité ako syntaktická skratka, keď zároveň definuje argument konštruktoru a vytvára verejnú vlastnosť objektu, ktorej priradí hodnotu parametru.

```
@Component({
    selector: 'app-hello',
    templateUrl: './hello.component.html'
})
export class HelloComponent {
    constructor(public alertService: AlertService) {}
    displayMessage() {
        this.alertService.display('Hello World!');
    }
}
```

mocou selectoru `app-page-header`, ako vo výpise 21. Hranatými zátvorkami nastavíme hodnoty vstupných vlastností a oblými zátvorkami definujeme reakciu na výstupnú udalosť.

Bežné komponenty ako prvky formulárov, tabuľky a dialógy nemusíme implementovať sami. Môžeme využiť knižnicu ako napríklad *bootstrap*[15] alebo *Angular Material*.[16] Druhá menovaná je od vývojárov Angularu priamo pre Angular. Ponúka napríklad aj jednoduchšie rozhranie pre testovanie. Preto sme sa rozhodli využiť práve Angular Material.

Výpis kódu 23 Služba je dekorovaná trieda. V dekorátore definujeme koreňový `Injector`. Tým sprístupňujeme jednu spoločnú inštanciu tejto služby pre všetky ostatné komponenty a služby.

```
@Injectable({providedIn: 'root'})
export class AlertService {
    display(message: string) {
        alert(message);
    }
}
```

Dependency injection

Angular sa automaticky postará o vytvorenie inštancie triedy komponentu. Mechanizmom *dependency injection (DI)* vytvorí inštancie služieb, na ktorých je náš komponent závislý. Stačí ich pridať s typovou anotáciou do konštruktoru. Príklad komponentu využívajúceho DI ukazujeme vo výpise 22.

Službu definujeme ako dekorovanú triedu. Príklad je vo výpise 23. Táto trieda môže mať rovnako vlastný konštruktor s inými závislosťami, o ktoré je automaticky postarané.

Dependency injection nám pomáha s organizáciou kódu. Keď využívame službu, nechceme sa zaoberať jej závislosťami a ich konštrukciou. Argument `{providedIn: 'root'}` určuje, že celý náš aplikačný kód bude mať dostupnú jednu zdieľanú inštanciu danej služby. Neskôr ukážeme implementáciu podobného, ale zjednodušeného, mechanizmu v Pythone.

V príkladoch vyššie sme využívali jazyk *TypeScript*. Tento jazyk rozširuje JavaScript o statickú typovú kontrolu. Podobne ako typové anotácie v Pythone bude náš kód prehľadnejší, ľahšie udržiavateľný, dostaneme viac pomoci od textového editora a predídeme mnohým chybám. TypeScript sa kompiluje do JavaScriptu zrozumiteľného pre webový prehliadač. Dovoľuje nám využívať aj nové funkcie, ktoré sú prekompilované do staršej verzie, čím sa zaručuje kompatibilita s väčším množstvom reálnych zariadení.[17]

Udalosťami riadené programovanie

Súčasťou projektu založeného v Angulari[18] je TypeScript, konfigurácia testov, a ďalsie knižnice tretích strán vrátane *RxJS*. RxJS implementuje návrhový vzor

[15] `https://themes.getbootstrap.com/`
[16] `https://material.angular.io/`
[17] `https://www.typescriptlang.org/`
[18] `https://angular.io/guide/file-structure`

Výpis kódu 24 Príklad práce s triedou `Subject`.

```
import { Subject } from 'rxjs';
import { map } from 'rxjs/operators';

const subject = new Subject<string>();
subject.pipe(
            map((msg) => `${msg}!`)
        )
        .subscribe((msg) => console.log(msg));

subject.next('Hello'); // 'Hello!'
subject.next('Hello World'); // 'Hello World!'
subject.complete();
```

observer, pomocou ktorého budeme posielať správy medzi súčasťami aplikácie.[19]

Základná rozhranie, s ktorým pracujeme, je `Observable`. Obsahuje metódu `subscribe`, ktorou zaregistrujeme volanie vykonané pri prijatí správy. Jedna z implementácií `Observable` je `Subject`.[20] Ten si uchováva zoznam pozorovateľov. Volaním metódy `next` im pošleme správu.

Vo výpise 24 sme ilustrovali použitie metódy `pipe` objektu `Observable`. Týmto spôsobom dokážeme aplikovať niekoľko operátorov transformujúcich tok dát. Medzi operátory patrí aj `map`, ktorým sme k správe pripojili výkričník. Volaním `complete` avizujeme, že už nebudeme posielať ďalšie správy.

Pri práci v Angulari sa viackrát stretneme s tým, že ako návratovú hodnotu dostaneme objekt typu `Observable`. Obdržíme tak napríklad výsledok HTTP požiadavky. V inej situácii budeme rovnakým mechanizmom reagovať na zmeny hodnôt vo formulári.

V prípade, že o príjmanie správ už nemáme záujem, musíme sa odhlásiť. Jedna možnosť je zavolať `unsubscribe` na objekte vrátenom metódou `subscribe`. Druhá možnosť je využiť operátor `takeUntil`, ktorý berie `Observable` ako argument, a odhlási nás v momente ako prijme správu z argumentu.

V príklade z výpisu 25 sa odhlásime pri vykonaní predposledného príkazu. Správu `'Pong'` už nikto neprevezme, takže sa ani nič nevypíše. Tento prístup je mnohokrát výhodné použiť, keď v jednom komponente odoberáme správy z viacerých inštancií `Observable`. Na konci životnosti komponentu potrebujeme zrušiť všetky odbery a nechať garbage collector uvoľniť zdroje. Správne miesto, kde je to vhodné urobiť, sa nachádza v metóde `ngOnDestroy` volanej na konci životnosti komponentu. Ilustrujeme to skráteným príkladom vo výpise 26. Tento

[19] `https://angular.io/guide/observables`
[20] `https://rxjs-dev.firebaseapp.com/guide/subject`

Výpis kódu 25 Riadenie odhlásenia odberu pomocou ďalšej inštancie `Subject`.

```
import { Subject } from 'rxjs';
import { takeUntil } from 'rxjs/operators';

const unsubscriber = new Subject();
const subject = new Subject<string>();

subject.pipe(takeUntil(unsubscriber))
        .subscribe(console.log);

subject.next('Ping'); // 'Ping'
unsubscriber.next();
subject.next('Pong');
```

Výpis kódu 26 Príklad komponentu odoberajúceho centrálne riadený stav.

```
export class ProjectComponent implements OnInit, OnDestroy {
    project: ProjectView;
    private unsubscribe$ = new Subject();

    constructor(
        private activeProject: ActiveProjectService
    ) { }

    ngOnInit() {
        this.activeProject.project$
            .pipe(takeUntil(this.unsubscribe$))
            .subscribe((project) => this.project = project);
    }

    ngOnDestroy() {
        this.unsubscribe$.next();
        this.unsubscribe$.complete();
    }
}
```

komponent odoberá centrálne spravovaný stav zo služby kým nie je zničený.

4.2 Princípy generovania dát

V tejto sekcii sa budeme zaoberať úlohou generovania dát. Na začiatku zavedieme objekt *generátor*. Navrhneme štruktúru, funkcionalitu a rozhranie tohto objektu. Ďalej popíšeme, ako objekt generátora zapadá do širšej štruktúry aplikácie. Navrhneme algoritmy schopné prostredníctvom generátorov vytvárať dáta určenej štruktúry, pričom tieto dáta musia spĺňať integritné obmedzenia.

4.2.1 Generátor

Generátor je základným pojmom v rámci úlohy generovania dát. Tým myslíme objekt, ktorý na základe štatistického rozdelenia alebo algoritmu vytvára dáta. Od generátora vyžadujeme nasledujúce vlastnosti:

- Generátor by mal byť parametrizovateľný. Generátoru uniformne rozdelených čísel by sme napríklad mohli určiť rozsah hodnôt. Tieto parametre budú určené používateľom alebo odhadnuté.
- S generátorom sa viaže algoritmus, ktorým odhaduje vlastné parametre zo zdrojových dát.
- Dáta musia byť generované deterministicky. Tým vznikne možnosť spustiť procedúru generovania viackrát s rovnakým výstupom.

Vzťah generátorov a stĺpcov

Porovnáme dva prístupy k viazaniu stĺpcov s generátormi. Ponúkajú sa tieto možnosti:

1. stĺpec je zviazaný s práve jedným generátorom,
2. generátor je zviazaný s n-ticou stĺpcov.

V analyzovaných riešeniach sa využíva najmä prvý prístup. Funguje to dobre z pohľadu vnútornej reprezentácie aj užívateľského prostredia. Niektoré generátory umožňujú odkazovať sa na hodnotu vygenerovanú iným generátorom. To sa využije v situácii, keď v jednom stĺpci náhodne vyberieme dátum narodenia osoby a v ďalšom stĺpci z toho vypočítame súčasný vek. Takéto prípady však nepovažujeme za dobrý návrh štruktúry databáz.

Slabina prvého prístupu sa objaví, keď potrebujeme generovať závislé hodnoty. Z pohľadu štatistických rozdelení môže byť žiadúce, aby hodnoty dvoch

alebo viacerých stĺpcov neboli nezávislé, ale pochádzali z určeného viacrozmerného rozdelenia. Príkladom je generovanie dvojíc stĺpcov štát a mesto, pričom vyberáme dvojice z reálneho sveta. Navyše keď generujeme hodnoty pre stĺpce obmedzené kompozitným cudzím kľúčom, tiež môže byť výhodnejšie uvažovať nad nimi ako n-ticou stĺpcov.

Pre zlepšenie tejto situácie volíme druhý prístup, v ktorom je výstupom generátora celá navzájom závislá n-tica. Z pohľadu užívateľa generátora je irelevantné, či daný generátor jednotlivé zložky generuje v nejakom konkrétnom poradí alebo súčasne.

Každý generátor bude jedna nezávislá jednotka. Závislosti medzi stĺpcami, za ktoré je zodpovedný, budú riešené v rámci jeho vnútornej implementácie a oddelené od zvyšku aplikácie. Žiadny generátor nebude zasahovať do práce iného generátora. Dostaneme tým lepšie riešenie z pohľadu softvérového návrhu. SQL Data Generator 4 nám neumožňuje upraviť generátor pre stĺpce s kompozitným cudzím kľúčom.[21] Naopak v našom riešení budú generátory pre stĺpce s cudzím kľúčom úplne prirodzenou súčasťou návrhu. Kompletná logika bude zapúzdrená vo vnútri tohto generátora.

Odhad parametrov generátorov

Parametre generátorov budeme čerpať zo zdrojových dát. Všeobecným princípom je predpoklad, že zdrojové dáta boli vygenerované rovnakým generátorom s neznámymi hodnotami parametrov. Zo zdrojových dát teda odhadneme hodnoty parametrov a našim generátorom nastavíme parametre na tieto hodnoty. Tieto hodnoty môžu byť dodatočne prispôsobené užívateľom.

V prípade uniformného rozdelenia vypočítame v zdrojovom stĺpci najmenšiu a najväčšiu hodnotu, to nastavíme ako parametre daného generátora. Analogicky v prípade normálneho rozdelenia spočítame strednú hodnotu a rozptyl zdrojových hodnôt.

Jedná možná implementácia by ku každému stĺpcu vypočítavala metadáta ako minimum, maximum, stredná hodnota a ďalšie. Generátor by si následne vyberal parametre, ktoré by preň boli relevantné. Tento prístup by bol neefektívny, pretože by sme počítali mnoho zbytočných odhadov. Navyše v prípade parametrov komplikovanejších modelov, ako napríklad váh neurónovej siete, je interpretácia parametrov úzko zviazaná s konkrétnym modelom.

Z pohľadu softvérového návrhu je preto výhodnejšie držať metódy výpočtu parametrov v blízkosti metód, ktoré dané parametre používajú. V našej aplikácii bude odhad parametrov súčasťou implementácie generátora. Bude to teda metóda, ktorá bude cez jednotné rozhranie pristupovať k zdrojovým dátam a ľubo-

[21] `https://documentation.red-gate.com/sdg4/using-generators`

voľným algoritmom počítať hodnoty parametrov. Každému stĺpcu je pridelený jeden generátor. Nenastane preto situácia, keď by viacero generátorov počítalo rovnaké parametre pre rovnaký stĺpec. V súlade s cieľmi práce dostávame vďaka tejto voľbe lepšiu rozšíriteľnosť. Pridanie nového generátora s vlastnými parametrami si vyžaduje pridanie kódu iba do jediného modulu.

4.2.2 Analýza základných generátorov

Implementujeme niekoľko široko využiteľných generátorov. Táto množina generátorov by mala byť schopná vygenerovať dáta, ktoré budú spĺňať aspoň základné požiadavky na dátové typy a integritu v bežných databázových aplikáciách. Presnejšie teda chceme, aby tieto dáta boli aspoň prijaté daným databázovým (alebo iným) systémom.

Neočakávame však, že sa nám podarí odraziť všetky zásadné rysy pôvodných dát. Tento cieľ by mal byť dosiahnuteľný pomocou programovania vlastných generátorov, špecifických pre potreby danej aplikácie. Vymenovanie a implementácia základnej sady generátorov nám pomôže pochopiť požiadavky pre voľbu vhodných rozhraní.

Jednostĺpcové generátory

V najjednoduchšom prípade budeme vytvárať hodnoty len pre jeden stĺpec. Ukážeme niekoľko generátorov rozdelených podľa dátových typov. V prípade reťazcov vytvoríme generátory vyberajúce mená ľudí, geografické označenia, prípadne termíny z rôznych oblastí. Na toto použijeme knižnicu tretej strany. Zvolili sme balíček *Faker*.[22] pretože ide o populárny open source projekt. Všeobecnejší generátor bude skladať reťazce náhodnej dĺžky z písmen a čísel, parametrizovaný minimálnou a maximálnou dĺžkou.

Pre desatinné čísla ponúkneme generátory nasledujúce niektoré štatistické rozloženia. Implementujeme uniformné rozloženie parametrizované minimom a maximom. Ďalej ponúkneme Gaussovo rozloženie parametrizované strednou hodnotou a rozptylom. Tieto rozloženia môžeme podobne implementovať aj pre celé čísla, akurát výslednú hodnotu zaokrúhlime. Vystačíme si pritom so štandardnou knižnicou Pythonu.[23]

Pre typ `boolean` budeme vyberať hodnoty z Bernoulliho rozdelenia. Dátumy budú pochádzať z uniformného rozdelenia určeného rozsahu.

[22] `https://faker.readthedocs.io/en/master/`
[23] `https://docs.python.org/3/library/random.html`

Prázdne hodnoty

Jednostĺpcové generátory by mali byť schopné vygenerovať `null`. V základe to môžeme riešiť jednotným spôsobom — pridáme parameter interpretovaný ako želaný podiel `null` hodnôt. Pri každom vzorkovaní potom tento podiel interpretujeme ako pravdepodobnosť, že vrátime `null`. Inak vrátime hodnotu vytvorenú generátorom.

Tento jednoduchý mechanizmus využijeme pri všetkých našich základných generátoroch. V záujme všeobecnosti systému toto správanie necháme preťažiteľné. Vďaka tomu bude možné implementovať generátor, ktorý bude generovanie `null` riešiť svojím vlastným spôsobom.

Surrogate key

Surrogate key je špeciálny prípad primárneho kľúča, ktorý sme definovali v časti 2.2.1. Aby boli výstupné záznamy prijaté cieľovou databázou, surrogate key si žiada špeciálne ošetrenie. Vyrobíme preto preň samostatný generátor. Keď vkladáme do súboru, takýto generátor by mal vracať čísla postupne od jednotky. Keď vkladáme do databázy, nesmieme generovať nič a nechať databázový systém, aby vytvorenému záznamu priradil primárny kľúč.

Vyriešime to tak, že generátor bude mať príznak určujúci, či jeho hodnota má byť vygenerovaná databázou. Modul využívajúci túto API sa už musí postarať o to, aby hodnotu získal z generátora alebo z databázy.

Cudzie kľúče

V porovnaní s inými softvérovými riešeniami sme sa rozhodli nerobiť rozdiely medzi obyčajným stĺpcom a stĺpcom, ktorý je obmedzený cudzím kľúčom. Užívateľ teda bude môcť priradiť aj obmedzenému stĺpcu ľubovoľný generátor vhodný pre daný dátový typ. Ako sme už spomínali, dosahujeme tým širšie možnosti prispôsobenia systému.

Nejde však o dostačujúce riešenie v súvislosti so stĺpcami obmedzenými cudzím kľúčom. Kontrola integritných obmedzení sa postará o to, že keď by záznam vygenerovaný pre tabuľku porušoval cudzí kľúč, tento záznam bude zahodený. Ak by sme pre stĺpec obmedzený cudzím kľúčom generovali hodnoty naivne všeobecným generátorom, mohlo by byť príliš veľa hodnôt zahodených.

V tejto časti navrhneme spečiálny generátor cudzích kľúčov. Cieľom je vytvárať hodnoty pre stĺpce obmedzené cudzím kľúčom tak, aby boli prijaté kontrolou integritných obmedzení. Využívať pritom bude rovnaké rozhranie ako každý iný generátor. Základná myšlienka je vyberať existujúce hodnoty zo stĺpcov, na ktoré sa odkazujeme. Tento princíp implikuje, že:

- generátor by mal mať prístup k vygenerovaným hodnotám,
- záznamy pre tabuľku, na ktorú sa odkazujeme, musíme vygenerovať skôr.

Prístup k vygenerovaným hodnotám umožníme tak, že v rámci generovacej procedúry budeme držať v dátovej štruktúre všetky doposiaľ vytvorené záznamy. Nevýhoda uchovávania tejto štruktúry je, že pamäťové nároky budú rásť úmerne objemu generovaných dát. Viaceré časti nášho systému — napríklad kontrola obmedzení — sú však nastavené tak, že sa tomu nedokážeme vyhnúť.

O generovanie záznamov pre tabuľky v poradí závislom na cudzích kľúčoch sa budeme musieť v každom prípade postarať. Cudzie kľúče určujú poradie generovania aj z dôvodu poradia vkladania do databázy, pretože najprv musíme vložiť dáta, na ktoré sa chystáme odkazovať. Využíva to aj kontrola obmedzení. Tento problém presnejšie analyzujeme v časti 4.2.4.

Cudzí kľúč môže referencovať aj hodnoty v rámci jednej tabuľky. Takto zvolené riešenie si s tým poradí. Prvá vygenerovaná hodnota takého kľúča musí byť `null`, pri každej ďalšej vyberieme buď `null` alebo uniformne z vygenerovaných hodnôt v referencovanom stĺpci (stĺpcoch).

Celý tento generátor implementujeme ako multistĺpcový, bude teda vracať n-ticu. Fungovať to bude samozrejme aj v prípade, keď nejde o kompozitný cudzí kľúč. Ide len o špeciálny prípad $n = 1$, ktorý nie je potrebné osobitne riešiť.

Zostáva vyriešiť, akým spôsobom budeme generátoru určovať, z ktorých stĺpcov má čerpať hodnoty. Necháme to na generátor, aby si našiel svoj zodpovedajúci cudzí kľúč. Generátor cudzieho kľúča si prečíta zoznam stĺpcov, ktoré sú mu pridelené a pokúsi sa nájsť adekvátne integritné obmedzenie. Adekvátne je také, že množina stĺpcov priradených generátoru je podmnožina množiny stĺpcov obmedzených cudzím kľúčom. V prípade, že sa mu adekvátny kľúč nájsť nepodarí, vráti chybu.

Automatické vyhľadanie adekvátneho integritného obmedzenia si vyžaduje, aby implementácia generátora mala prístup k integritným obmedzeniam danej tabuľky. Ide o prirodzenú požiadavku — inteligentný generátor by mohol prispôsobovať svoje správanie napríklad aj v závislosti od `CHECK` obmedzení.

4.2.3 Procedúra generovania dát

V tejto časti navrhneme postup, ako pomocou generátorov vytvoriť dáta vo výstupnom formáte. Rozoberieme voľbu poradia tabuliek, v ktorom do nich budeme vytvárať dáta.

Algoritmus 1 načrtá vkladanie dát do jednej tabuľky. Táto procedúra dostane ako argument výstupný ovládač. Ten predstavuje rozhranie pred ľubovoľným výstupným formátom. Od výstupného ovládača očakávame nasledujúce funkcie:

- vloženie záznamu do tabuľky,
- v prípade, že je interaktívny — vrátenie ovládačom pridelených hodnôt.

Interaktívny ovládač bude v našej implementácii iba ovládač stojaci pred relačnou databázou. Je to práve kvôli primárnym kľúčom priradených databázovým systémom. Pre prípad, že by v budúcnosti bol pridaný ďalší výstup schopný priradiť niektoré hodnoty, sme interaktivitu ovládača navrhli ako všeobecné rozhranie.

V ďalšom parametre do procedúry generovania dát predávame počet želaných riadkov pre danú tabuľku. Tento počet nemusí byť naplnený. Príčinou môže byť:

- porušenie integrity — odhalené kontrolou integrity (zahŕňa unikátnosť hodnôt, primárne a cudzie kľúče),
- neprijatie záznamov výstupným ovládačom.

Jediný z našich implementovaných ovládačov schopných neprijať je zase iba relačná databáza. V budúcnosti by tam mohli patriť ďalšie. Databázový systém môže nájsť porušenie integrity, ktoré sa našou kontrolou nepodarilo odchytiť. Nevieme kontrolovať kompletne všetky obmedzenia (napríklad `CHECK`). Cieľová databáza môže v čase spustenia obsahovať už iné dáta, s ktorými nepočítame, a práve tieto dáta spôsobia porušenie integrity. Trigger môže mať tiež zásadný vplyv na vkladanie dát. Okrem toho sa môžu vyskytnúť iné problémy:

- nesprávne prihlasovacie údaje alebo nedostatočné práva,
- strata spojenia s databázou,
- nesúhlas našej schémy a reálnej schémy.

V takom prípade je nepravdepodobné, že sa to opraví vygenerovaním ďalšie záznamu. Preto musíme prerušiť celú procedúru.

Keď je vygenerovaný riadok odmietnutý, máme tieto možností ako sa s tým vysporiadať:

1. vo výstupe necháme menej riadkov ako bolo žiadané,
2. budeme generovať až kým sa žiadaný počet nenaplní,
3. kompromis medzi dvomi predchádzajúcimi bodmi.

Prvé riešenie by malo za následok zbytočne časté nenaplnenie kvóty. Napríklad pri obmedzení na unikátnosť hodnoty je pravdepodobnosť kolízií vyššia ako sa môže zdať.[24] V druhom prípade by sme dostali algoritmus, ktorý nie je konečný. Taká konfigurácia napríklad nastáva, keď generujeme čísla z rozsahu, ale chceme dostať viac riadkov, ako je veľkosť rozsahu. Kompromis medzi týmito možnosťami dostaneme tak, že zavedieme číselný parameter $tolerance$. Riadky generujeme kým ich nemáme dosť úspešne vložených alebo kým počet pokusov nedosiahol $tolerance$-násobok želaného počtu. Pre hodnotu $tolerance = 1$ by sme dostali práve prvé riešenie a pre $tolerance = \infty$ práve druhé. Rozumné voľby sú teda $tolerance \in [1, \infty)$.

Algoritmus 1 Pre danú tabuľku, výstupný ovládač, počet žiadaných riadkov a zadanú toleranciu generujeme dáta. Cyklus opakujeme, kým nemáme dosť vložených riadkov alebo kým počet pokusov neprevýšil počet určený toleranciou. V cykle vygenerujeme riadok, skontrolujeme integritu a pokúsime sa o vloženie. Postaráme sa o prípadné načítanie hodnôt vygenerovaných ovládačom a kompletný riadok uložíme.

procedure GenerateDataForTable($table$, $driver$, $requested$, $tolerance$)
  $generators \leftarrow$ vytvor inštancie generátorov pre tabuľku $table$
  $inserted \leftarrow 0$
  $attempted \leftarrow 0$
  while $inserted < requested$ **and** $attempted < tolerance \cdot requested$ **do**
    $row \leftarrow$ generuj prostredníctvom $generators$
    $attempted \leftarrow attempted + 1$
    if not integrita je neporušená pre row **then continue**
    end if
    $success \leftarrow$ pokús sa vložiť riadok row pomocou ovládača $driver$
    if not $success$ **then continue**
    end if
    $inserted \leftarrow inserted + 1$
    if $driver$ je interaktívny **then**
      zapíš do row hodnoty načítané ovládačom $driver$
    end if
    ulož row do vnútornej dátovej štruktúry
  end while
end procedure

[24] `https://en.wikipedia.org/wiki/Birthday_problem`

4.2.4 Poradie tabuliek

Potrebujeme zvoliť poradie tabuliek, v ktorom ich budeme napĺňať. Musíme sa tým zaoberať kvôli závislostiam vytvorených cudzími kľúčmi. Na správnom poradí závisia nasledujúce časti aplikácie:

- priamy výstup do databázy,
- výstup do `INSERT` scriptu,
- kontrola integrity,
- generátory cudzích kľúčov.

Pri vložení záznamu do tabuľky s cudzím kľúčom databázový systém kontroluje integritu. Musíme zabezpečiť, aby v tom momente už existoval záznam, na ktorý sa vkladaný záznam odkazuje. Rovnako to funguje v prípade `INSERT` scriptu. Našu kontrolu integrity navrhneme po vzore databázovej kontroly — bude spracovávať záznamy v rovnakom poradí, ako by sme ich vkladali do databázy. Náš navrhovaný generátor pre cudzie kľúče funguje na princípe náhodnej voľby z existujúcich hodnôt v referencovanej tabuľke. To znamená, že referencovaná tabuľka musí už byť naplnená v momente, keď generujeme tabuľku obmedzenú týmto cudzím kľúčom.

Pre účely ilustrácie sme navrhli schému tabuliek 27. Skúsme teraz vytvoriť nejaké poradie tabuliek, v akom by sme ich mohli napĺňať. Tabuľky reprezentujme ako vrcholy a cudzie kľúče nech sú orientované hrany, smerujúce od obmedzenej tabuľky k referencovanej. Túto štruktúru môžeme vidieť na obrázku 4.1.

Vytvorením hrán z cudzích kľúčov medzi tabuľkami dostávame orientovaný multigraf. Slučky môžeme bezpečne odstrániť. Ich interpretácia je, že hodnoty vygenerované pre cudzí kľúč v danej tabuľke musia pochádzať zo skôr vygenerovaných hodnôt v referencovanom stĺpci (stĺpcoch) v rovnakej tabuľke. Tento fakt neovplyvňuje poradie tabuliek.

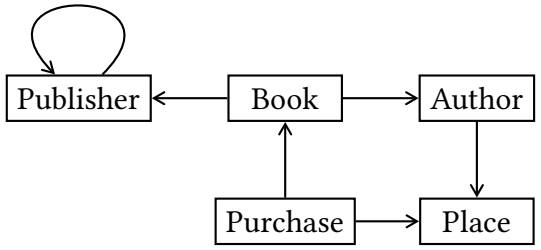
Násobné hrany nemajú v tomto kontexte iný význam ako obyčajné hrany. Odstránením slučiek a zjednotením násobných hrán nám zostáva iba orientovaný graf. Budeme hľadať poradie, v ktorom každá hrana ukazuje na predchádzajúci vrchol. To je *opačné topologické poradie*.

Opačné topologické poradie orientovaného grafu dostaneme ako poradie, v ktorom vyhľadávanie do hĺbky opúšťa vrcholy. Topologické poradie existuje práve vtedy, keď je graf acyklický [7, ss. 128-129].

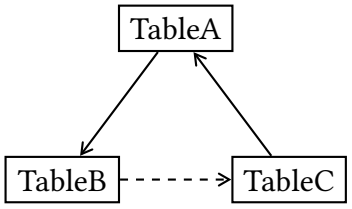
Zaoberajme sa teraz prípadom, keď topologické poradie neexistuje. Na ilustráciu problému majme tabuľky `A`, `B` a `C`, ktorých závislosti tvoria cyklus. Zobrazané sú na obrázku 4.2. Možné riešenie je niektorý z kľúčov ignorovať, čím

Výpis kódu 27 Relačná schéma predstavujúca entity autor, kniha, vydavateľ, miesto a kúpa knihy. Príklad je síce umelý, ale obsahuje aj zložitejšie situácie, ktoré v praxi môžu nastať. Špeciálnym javom je štruktúra vydavateľstiev — vydavatelia môžu mať rodičovskú spoločnosť. V prípade knihy máme kompozitný kľúč, ktorý v sebe zahŕňa aj pole obmedzené cudzím kľúčom. Kúpa knihy obsahuje dva kompozitné cudzie kľúče.

```
Author(firstName, lastName, dateOfBirth, homeCountry, homeCity),
homeCountry, homeCity ⊆ Place.country, city

Book(publisher, title, edition, authorFirstName, authorLastName,
numberOfPages),
publisher ⊆ Publisher.companyName,
authorFirstName, authorLastName ⊆ Author.firstName, lastName,

Publisher(companyName, parentCompany),
parentCompany ⊆ Publisher.companyName

Place(country, city)

Purchase(purchaseId, publisher, title, edition, quantity
placeCountry, placeCity),
publisher, title, edition ⊆ Book.publisher, title, edition
placeCountry, placeCity ⊆ Place.country, city
```



Obr. 4.1 Zobrazenie závislostí medzi tabuľkami zo schémy 27. Vrcholy sú tabuľky. Pre každý cudzí kľúč máme jednu hranu vedúcu od obmedzenej tabuľky k referencovanej.



Obr. 4.2 Tri tabuľky s cyklickými závislosťami danými cudzími kľúčmi. Hrana smeruje od obmedzenej tabuľky k referencovanej. Plná hrana predstavuje cudzí kľúč s obmedzením `NOT NULL`, čiarkovaná hrana znázorňuje kľúč s povolenými `NULL` hodnotami.

zmažeme jednu z hrán a graf sa môže stať opäť acyklickým. V závislosti od konkrétnej schémy a konfigurácie generátorov to môže skončiť zahodením všetkých vygenerovaných záznamov kontrolou integrity.

Ako naznačuje obrázok 4.2, najvýhodnejšie by bolo ignorovať závislosť, kde stĺpce s cudzím kľúčom môžu nadobúdať hodnotu `null`. Algoritmus sa teda bude najprv pokúšať vytvoriť poradie so zohľadnením všetkých závislostí. Ak to nevyjde, zmaže cudzie kľúče s povolenou hodnotou `null` a znova sa pokúsi o zotriedenie tabuliek.

V situácii na obrázku by sme zvolili opačné topologické poradie `B`, `A`, `C`. S tým by čiastočne fungoval nami navrhnutý generátor cudzích kľúčov. V tabuľke `B` by však pre stĺpce s cudzím kľúčom boli vygenerované iba hodnoty `null`, takže ani to nie je ešte konečné riešenie. Tento problém vyrieši striedavé generovanie.

Striedavé generovanie

V časti 2.2.1 sme prezentovali, ako sa s cyklickými závislosťami vysporiadavajú existujúce riešenia. Snažia sa najprv naplniť tabuľku kompletne a potom prejsť na ďalšiu tabuľku a znova ju naplniť kompletne. Ako sme už naznačili, takto to nemôže fungovať správne. Jedno riešenie hlási chybu, druhé generuje neprípustné záznamy.

Inšpirujeme sa tým, ako by bola napĺňaná reálna databáza. Nadväzujúc na schému 27, autora musíme vložiť do databázy pred tým, ako vložíme jeho knihu. Musíme teda dodržiavať závislosti medzi tabuľkami. Po vložení knihy však môžeme znova vložiť ďalšieho autora.

V konečnom riešení vytvoríme poradie tabuliek podľa závislostí medzi nimi. Najprv sa pokúsime rešpektovať všetky závislosti. Ak je taký graf cyklický, zanedbáme cudzie kľúče s povolenými `null` hodnotami.

Tabuľky napĺňame striedavo. Prechádzame po tabuľkách v opačnom topologickom poradí. Do každej vložíme jeden záznam a posunieme sa na ďalšiu. Po vložení záznamu do poslednej prechádzame zase od začiatku, vynechávajúc tabuľky s dostatočným počtom záznamov.

Výpis kódu 28 Generovaný SQL script pre dve cyklicky závislé tabuľky naším softvérom. Záznamy sú prijaté databázou.

```
INSERT INTO "TableA" (id, fk) VALUES (1, NULL)
INSERT INTO "TableB" (id, fk) VALUES (1, 1)
INSERT INTO "TableA" (id, fk) VALUES (2, 1)
INSERT INTO "TableB" (id, fk) VALUES (2, 2)
INSERT INTO "TableA" (id, fk) VALUES (3, 1)
INSERT INTO "TableB" (id, fk) VALUES (3, 3)
```

Vo výpise 28 uvádzame výsledok tohto spôsobu generovania. Tento script bol vygenerovaný naším kompletným riešením. Pri vkladaní prvého záznamu je `null` jediná akceptovateľná hodnota pre cudzí kľúč. Cudzie kľúče v každom zázname sa odkazujú iba na skôr vygenerované záznamy.

Kontrola integritných obmedzení

Pri vkladaní do databázy dostávame spätnú väzbu, či sa záznam podarilo vložiť. Žiadnu odozvu nedostaneme pri vytváraní `INSERT` scriptu alebo iného súboru. Navrhneme preto algoritmy, ako zabezpečiť integritu dát nezávisle od výstupného média.

Integritné obmedzenia budeme kontrolovať pri jednotlivých záznamov v poradí generovania. Nevyhovujúci záznam nezaradíme do výstupu a pokračujeme generovaním ďalšieho záznamu. Malo by nám to zaručiť, že výsledok nielen spĺňa požiadavky integrity, ale je aj vložiteľný do databázy v poradí generovania.

Týmto prístupom dostaneme iný výsledok, ako keby sme najprv spustili celé generovanie a až potom overovali integritu. Je to spôsobené tým, že keď je integrita porušená cudzím kľúčom, dá sa opraviť vložením ďalšieho záznamu.

Priamočiaro kontrolujeme `NOT NULL`. Integritné obmedzenia typu `UNIQUE`, `PRIMARY KEY`, `FOREIGN KEY` vieme v zásade vyriešiť udržiavaním si množín vygenerovaných n-tíc:

- v prípade `PRIMARY KEY` a `UNIQUE` záznam prijmeme, ak n-tica obmedzených stĺpcov neexistuje v danej tabuľke,
- v prípade `FOREIGN KEY` záznam prijememe, ak n-tica obmedzených stĺpcov už existuje v množine n-tíc referencových stĺpcov.

Zložitejšia situácia nastáva pri `CHECK`. Ako sme už naznačili v časti 2.2.1, kontrola by bola náročná. V našej práci sa tým nebudeme zaoberať, pretože je to problém špecifický pre jediný zdroj vstupu (databázu) a bolo by ťažké dosiahnuť uspokojivý výsledok.

4.2.5 Automatické priraďovanie generátorov

Pri načítaní schémy z databázy, súboru alebo iného zdroja priraďujeme vhodný generátor k novým stĺpcom. Cieľom je ušetriť užívateľovi prácu. Určíme si presnejšie požiadavky:

1. stĺpcu by mal byť v každom prípade pridelený niektorý z generátorov — nesmie zostať bez generátora,
2. po pridelení generátorov by sme mali dostať takú konfiguráciu, ktorá je schopná generovať dáta prijaté kontrolou integrity,
3. logika priraďovania by mala byť čo najviac prispôsobiteľná autorom generátora,
4. musíme rešpektovať multistĺpcové generátory; náš návrh by mal byť schopný priradiť viacerým stĺpcom jeden generátor.

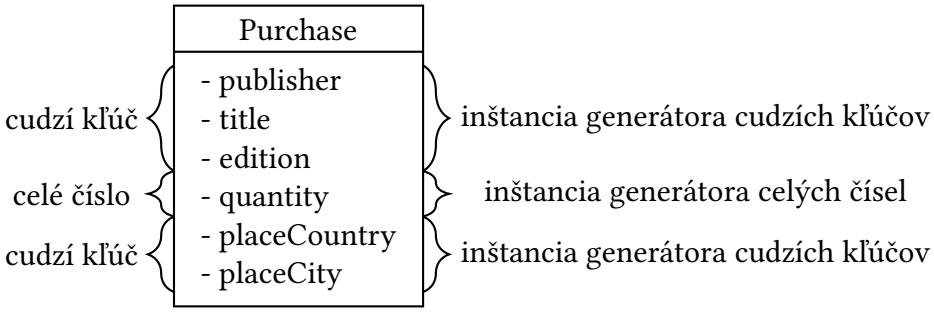
Začneme prvými dvoma bodmi. Naše základné generátory by sme mohli roztriediť na tieto kategórie:

1. zabezpečujúce integritu — primárne a cudzie kľúče,
2. generátory dát z nejakej špecifickej domény ako názvy štátov,
3. všeobecné generátory určené pre jeden dátový typ — náhodné čísla z intervalu, náhodné reťazce,
4. generátory poskytujúce dáta z iných štatistických rozdelení.

Idea algoritmu priradenia spočíva v tom, že prechádzame dostupné generátory v poradí týchto kategórií. Keď narazíme na *odporúčaný* generátor, pridelíme ho danému stĺpcu. Určenie, či je generátor odporúčaný pre konkrétny stĺpec, je výlučne záležitosťou implementácie tohto generátora.

Uveďme si na príklade, ako by to mohlo fungovať. Majme stĺpec, ktorý je súčasťou cudzieho kľúča. Začneme prechádzať generátory a ako prvý odporúčaný sa prihlási generátor cudzích kľúčov. To je aj najlepšia možnosť, pretože v súlade s druhou požiadavkou bude vytvárať dáta spĺňajúce integritu.

V inom príklade majme stĺpec s názvom `country` držiaci reťazce. V prvej kategórii preň nenájdeme odporúčaný generátor, pretože nie je súčasťou žiadneho kľúča. Pridelíme mu generátor vytvárajúci názvy štátov, ktorý signalizuje svoju vhodnosť práve kvôli názvu stĺpcu. Toto určenie nemusí byť nutne správne, ale užívateľ má vždy možnosť vymeniť automatický generátor za iný.



Obr. 4.3 Tabuľka zo schémy 27 a žiadané priradenie generátorov. Každý z kompozitných kľúčov by mal zdieľať jednu inštanciu generátora cudzích kľúčov.

Keď pre stĺpec nenájdeme žiadny vhodný generátor v prvej ani druhej kategórii, určite ho zachytíme v tretej kategórii. Tretia kategória by mala pokrývať každý podporovaný dátový typ. Jednotlivé generátory sa v nej bezpodmienečne budú vyhlasovať za odporúčané. Tým splníme prvú požiadavku, aby každý stĺpec dostal generátor.

Rožšírenie na multistĺpcové generátory

V nasledujúcom texte je dôležité rozlišovať *typ* generátora a *inštanciu* generátora.

- typ generátora — napríklad generátor celých čísel z intervalu alebo generátor cudzích kľúčov,
- inštancia generátora — inštancii sú pridelené konkrétne stĺpce v tabuľke a obsahuje konkrétne hodnoty parametrov.

Na diagrame 4.3 máme tabuľku s dvomi kompozitnými cudzími kľúčmi a jedným číslom. Každý stĺpec obmedzený cudzím kľúčom by mal dostať generátor typu „generátor cudzích kľúčov“, pretože ten zabezpečuje integritu výstupných dát. Inštanciu tohto typu generátora by mali zdieľať však len stĺpce obmedzené rovnakým cudzím kľúčom. Celkovo tak máme v našej tabuľke dve inštancie generátora cudzích kľúčov. Prvá inštancia je pridelená svojim trom obmedzeným stĺpcom a druhá inštancia je pridelená svojim dvom stĺpcom.

Nejde o jediný možný spôsob ako navrhnúť princíp fungovania generátora cudzích kľúčov. Mohli by sme každému stĺpcu obmedzenému cudzím kľúčom priradiť rovnakú inštanciu generátora. Tým by sme sa však problémom nevyhli, iba by sme ich presunuli do vnútra implementácie generátora. Tento generátor by musel vedieť nájsť všetky cudzie kľúče svojich stĺpcov a pre každý cudzí kľúč by vyberal hodnoty zvlášť. Na opačnej strane extrému je možnosť pridelenia vlastnej inštancie generátora každému stĺpcu. Potom by generátory museli navzájom nejakým spôsobom komunikovať, čo tiež nie je dobrým riešením.

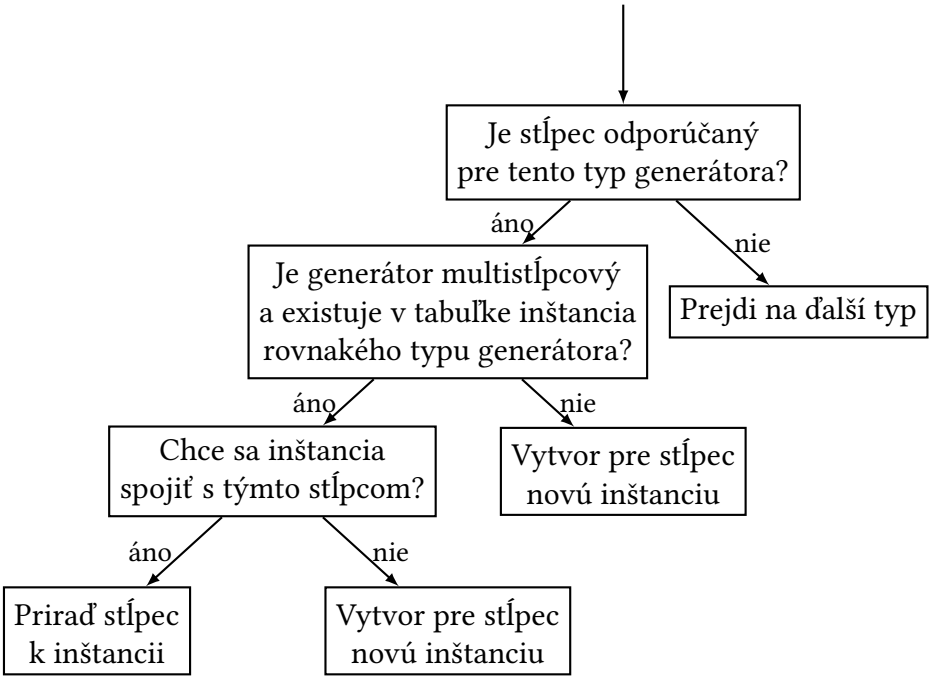
Navrhneme všeobecný algoritmus, kde je prideľovanie inštancie generátora stĺpcom riadené implementáciou generátora. Náš generátor cudzích kľúčov bude prideľovaný princípom ilustrovanom na obrázku 4.3. Pomocou algoritmu popísaného v nasledujúcom texte by bolo možné implementovať generátor využívajúci alternatívne spôsoby pridelenia. Navyše tento algoritmus nemusí byť využívaný len generátorom cudzích kľúčov. Využitie všeobecného algoritmu prideľovania je otvorené každému multistĺpcovému generátoru, ktorý chce ovplyvniť, akým spôsobom bude automaticky priraďovaný.

Fungovanie algoritmu vysvetlíme najprv na príklade tabuľky z diagramu 4.3, neskôr opíšeme všeobecný postup. Stĺpcu `publisher` nájdeme a priradíme odporúčaný generátor ako bolo popísané v minulej podsekcii, bude to generátor cudzích kľúčov. Pre stĺpec `title` bude odporúčaný tiež generátor cudzích kľúčov. Ide o multistĺpcový generátor, z ktorého už máme jednu inštanciu. Tejto inštancie sa spýtame (volaním metódy), či by jej mal byť priradený aj stĺpec `title`. Inštancia generátora cudzích kľúčov nájde cudzí kľúč, ktorý obmedzuje oba stĺpce. Stĺpec `title` kvôli tomu priradíme k tejto inštancii. Podobne to dopadne pre stĺpec `edition`.

Stĺpcu `quantity` je odporúčaný generátor celých čísel. Ten nie je multistĺpcový, takže stĺpcu je vytvorená nová inštancia. Pre stĺpec `placeCountry` je odporúčaný znova generátor cudzích kľúčov. Najprv sa dopytujeme existujúcej inštancie, či jej má byť tento stĺpec priradený. So stĺpcami tejto inštancie nezdieľa `placeCountry` cudzí kľúč, preto mu vytvoríme novú inštanciu. Stĺpcu `placeCity` je znova odporúčaný generátor cudzích kľúčov. S prvou vytvorenou inštanciou nezdieľa spoločný cudzí kľúč. Bude pridelený druhej inštancii, vytvorenej pre stĺpec `placeCountry`.

Všeobecný algoritmus prechádza postupne stĺpcami a dostupnými typmi generátorov, v poradí ako sme definovali kategórie. Pre dvojicu stĺpec a typ generátora urobí rozhodnutie podľa stromu 4.4. Ak typ generátora nie je odporúčaný pre daný stĺpec, skúsi ďalší typ. Ak je odporúčaný, tak mu buď priradí existujúcu inštanciu alebo vytvorí novú. O tom, či mu bude jedna z existujúcich inštancií priradená, rozhoduje samotná inštancia. V prípade generátora cudzích kľúčov toto rozhodnutie vyplýva z podoby cudzích kľúčov v danej tabuľke.

V implementačnej časti navrhneme na základe popísaných postupov konkrétne rozhrania. Spájanie stĺpcov s generátormi bude plne prispôsobiteľné daným generátorom. To, či je generátor odporúčaný pre stĺpec alebo či sa inštancia má priradiť k ďalšiemu stĺpcu, bude implementované ako sada metód. Týmto metódam bude poskytnutý dostatočný kontext, podľa ktorého urobia rozhodnutie.



Obr. 4.4 Rozhodovací strom popisujúci algoritmus automatického priraďovania stĺpcov ku generátorom.

Kapitola 5

Vývojová dokumentácia

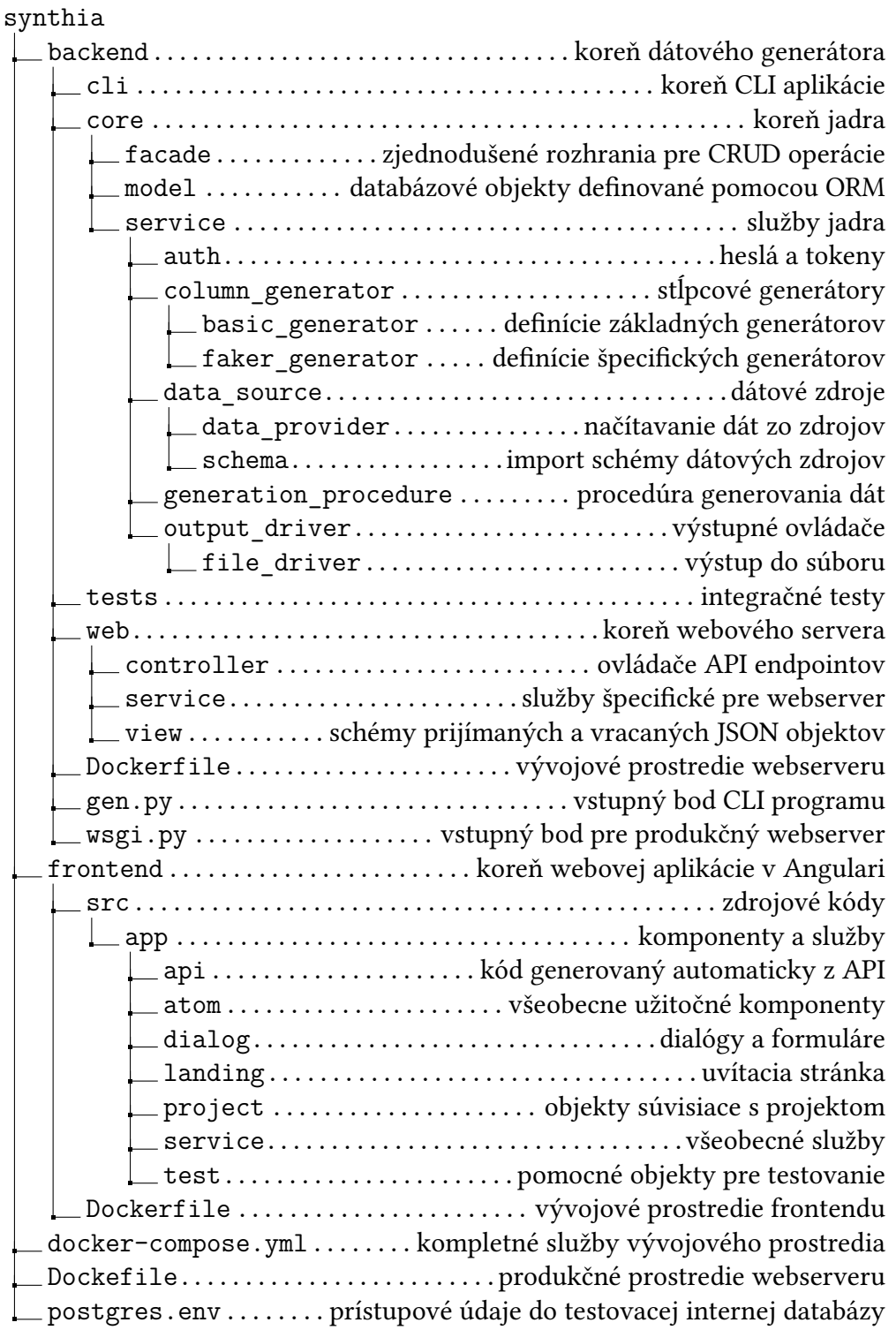
V tejto kapitole opíšeme našu implementáciu aplikácie. Na začiatku predstavíme jednotlivé vrstvy a vzťahy medzi nimi. Potom prejdeme na detailný popis častí architektúry v samostných sekciách v poradí zdola nahor. V sekcii 5.2 popíšeme tabuľky v internej databáze. Zadefinujeme tým objekty, s ktorými pracujeme. Väčšinu funkcionality dátového generátora sa snažíme implementovať v jadre opísaného v sekcii 5.3. Nad funkciami jadra stavia svoje rozhranie webový server popísaný v sekcii 5.4. Nakoniec sa budeme venovať webovej aplikácií v časti 5.5.

5.1 Architektúra

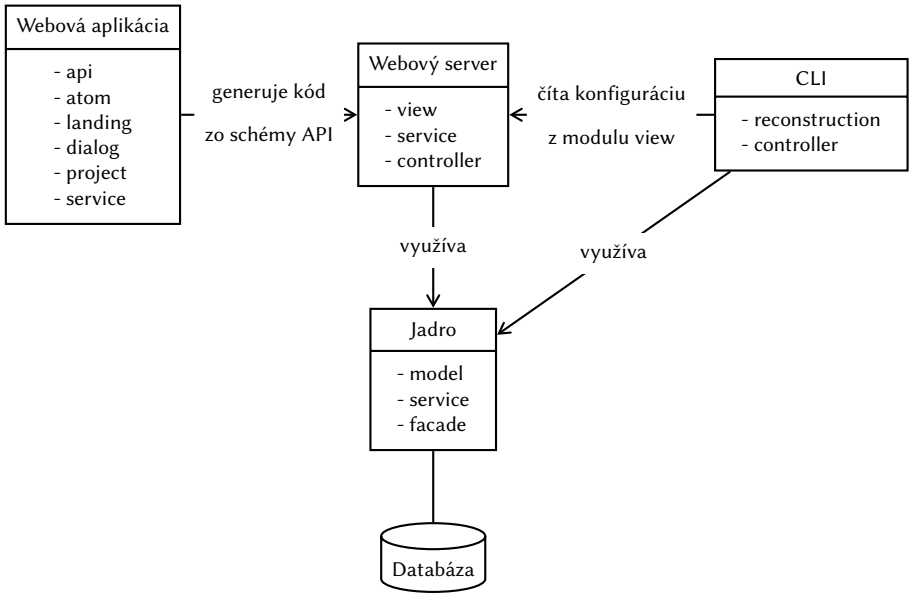
Jadro je Pythonový balíček obsahujúci základnú funkcionalitu dátového generátora predstavený v časti 4.1.2. Importujú ho dva ďalšie Pythonové balíčky — webový server a CLI. Webová aplikácia je oddelená od webového serveru. V časti 4.1.4 sme pre ňu vybrali a predstavili framework Angular. S webovým serverom komunikuje prostredníctvom REST API. Kód webovej aplikácie využívajúci API je automaticky vygenerovaný pomocou nástroja ng-swagger-gen. Časť CLI prijíma konfiguračný súbor vygenerovaný webovým serverom. Tento princíp bol špecifikovaný v časti 3.2.

Na obrázku 5.2 zobrazujeme jednotlivé vrstvy a vzťahy medzi nimi popísané vyššie. Pri každej funkčnej jednotke sme v odrážkach vypísali zoznam jej modulov na najvyššej úrovni. Vrstvy a ich moduly sa priamo viažu na podstromy v súborovej štruktúre. Prehľad najdôležitejších adresárov a súborov uvádzame na obrázku 5.1.

Pythonové balíčky sídlia v adresári `backend`. CLI, jadro a webový server sídlia v podadresároch `cli`, `core` a `web` respektíve. Kód webovej aplikácie sa nachádza v podadresári `frontend`. Relatívna cesta k jej základným modulom má prefix `frontend/src/app`.



Obr. 5.1 Stručný prehľad súborovej štruktúry práce.



Obr. 5.2 Prehľad štruktúry aplikácie a základných modulov jednotlivých vrstiev.

Vynechali sme popis menej dôležitých súborov a adresárov. V koreni webovej aplikácie je mnoho konfiguračných súborov bežných pre všetky Angularové aplikácie. Ich význam je dokumentuje sprievodca Angularu.[1]

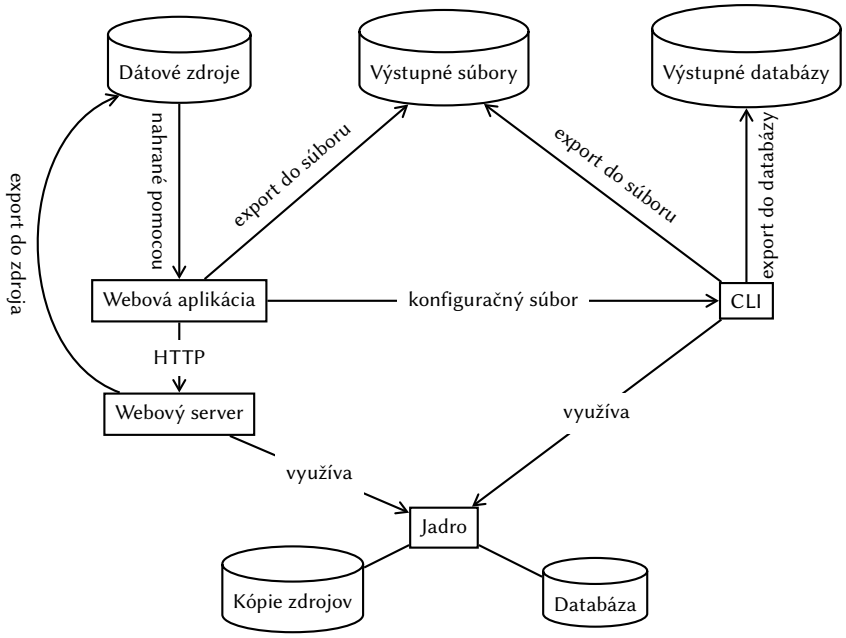
Integrácia vrstiev a dátových zdrojov

V časti 3.1.2 sme popísali akceptované typy dátových zdrojov. Dostaneme ich buď ako súbor alebo prístupové údaje k databáze. Špecifikovali sme, že užívateľ bude môcť po pridaní zdroja z neho kedykoľvek importovať schému. To znamená, že webový server bude musieť mať prístup k týmto súborom či databázam. Nestačí teda súbor spracovať len pri nahraní, musíme si vytvoriť lokálnu kópiu. Z rovnakého dôvodu ukladáme do našej internej databázy prístupové údaje do externej databázy. Tieto vzťahy naznačuje obrázok 5.3.

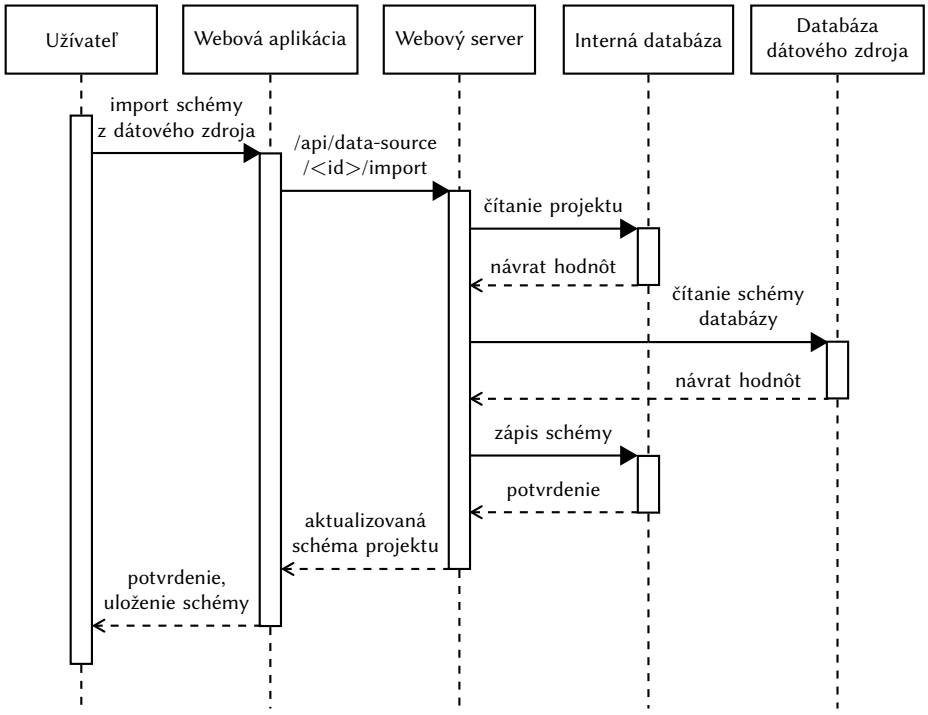
Z pridaného dátového zdroja môže užívateľ spustiť import schémy. Tento scenár zobrazuje diagram 5.4. Popisom tohto konkrétneho scenára ukážeme ako spolupracujú jednotlivé vrstvy s dátovými zdrojmi aj pri plnení iných požiadaviek. Detaily schválne vynechávame, tým sa budeme venovať v jednotlivých sekciách.

Užívateľ iniciuje proces stlačením tlačidla pri konkrétnom dátovom zdroji. Mohlo by ísť o súbor alebo databázu, na diagrame ukazujeme databázu. Webová aplikácia vygeneruje požiadavku na API.

[1] `https://angular.io/guide/file-structure`



Obr. 5.3 Prehľad štruktúry aplikácie a komunikácie s dátovými zdrojmi.



Obr. 5.4 Sekvenčný diagram znázorňujúci import schémy z dátového zdroja.

Webový server spraví niekoľko čítaní z internej databázy — potrebuje najmä informácie o dátovom zdroji a schéme projektu. Vďaka týmto informáciám sa dokáže pripojiť na externú databázu, z ktorej načíta jej schému. Ak by išlo o súbor, bola by načítaná schéma z lokálneho súboru. Následne prebehne zlúčenie projektovej schémy a načítanej schémy. To spočíva z pridania všetkých nových tabuliek do projektu. Tieto zmeny sú zapísané do internej databázy. Webový server vráti aplikácii aktualizovanú projektovú schému. Aplikácia dá užívateľovi vedieť o úspešnom vybavení požiadavky a aktualizuje si svoj interný stav — detaily rozoberieme v sekcii 5.5.

Generovanie dát môže byť z grafického prostredia spustené dvojakým spôsobom, ako sme to definovali v časti 3.1.5. Môžeme napĺňať niektorú zo zdrojových databáz. K tej sa pripája webový server s užívateľom zadanými prístupovými údajmi. Druhý spôsob je vygenerovať súbor, ktorý si užívateľ stiahne prostredníctvom webového prehliadača.

Exportovaná konfigurácia projektu je čitateľná CLI programom. Časť CLI využíva funkcionalitu jadra zaoberujúcu sa generovaním dát. Nepracuje s internou databázou ani nepotrebuje prístup k dátovým zdrojom. Všetky relevantné informácie sú ukryté v konfiguračnom súbore. Možnosti výstupu sú výpis do súboru alebo do databázy podľa zadaných prístupových údajov.

5.2 Databáza

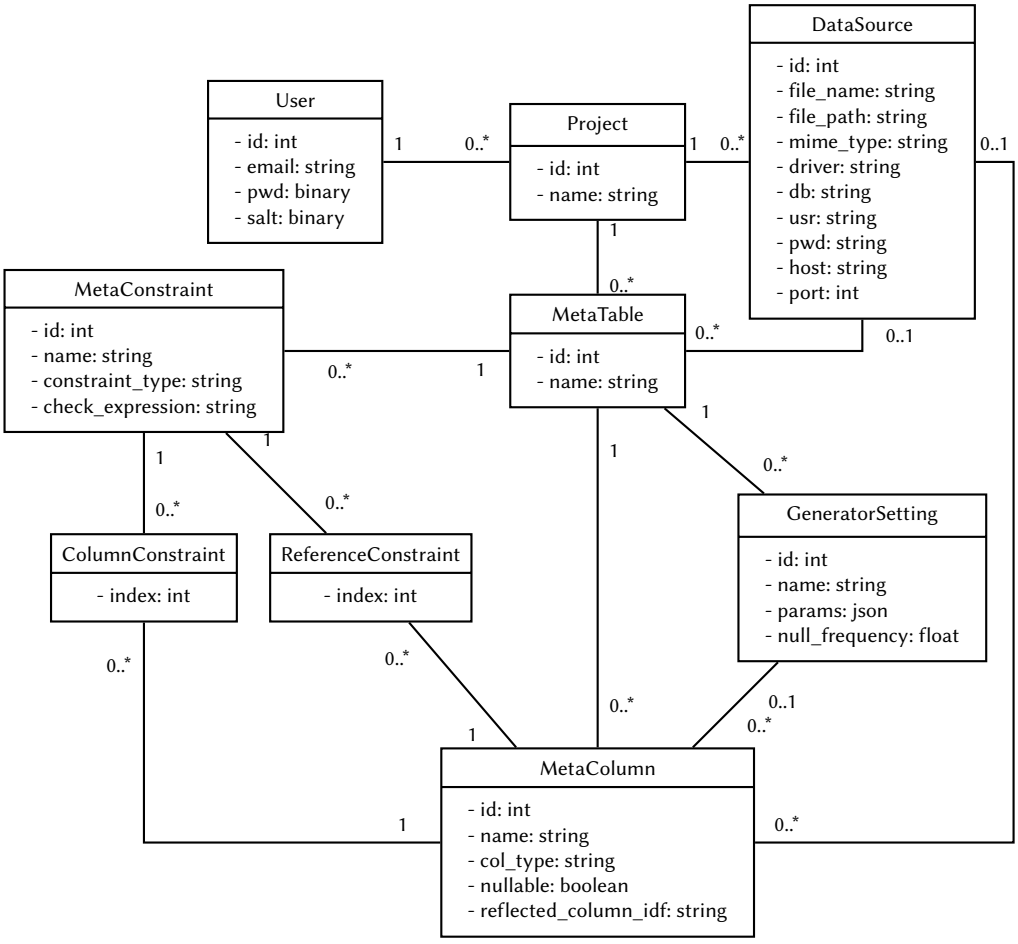
Vysvetlíme, s akými objektami pracujeme a definujeme vzťahy medzi nimi. Štruktúra databázy je znázornená na diagrame 5.5.

Začneme od užívateľa. Tabuľka `User` obsahuje unikátny email, zahešované heslo a hodnotu `salt` používanú pri hešovaní. V časti 3.1.1 sme žiadali možnosť rýchlej registrácie, bez uvádzania emailu a hesla — nazývame to anonymný užívateľ. Pre takého užívateľa zostávajú stĺpce email a heslo prázdne. Každý užívateľ, vrátane anonymného, je jednoznačne identifikovaný primárnym kľúčom `id` prideleným databázovým systémom.

Bolo špecifikované, že užívateľ si môže vytvoriť niekoľko projektov. Projekt má názov a obsahuje tabuľky a dátové zdroje.

Dátový zdroj

Každý nahraný dátový zdroj bude mať jeden záznam v tabuľke `DataSource`. Ako to bolo určené v časti 3.1.2, dátový zdroj je súbor alebo prístup k databáze. V prípade SQLite databázy tento záznam reprezentuje súbor a zároveň databázu. Pri súbore sú relevantné polia `file_name`, `file_path` a `mime_type`. Prvé pole obsahuje pôvodný názov súboru. Ten je dôležitý najmä pri súbore CSV, pretože



Obr. 5.5 Databázovy diagram.

Stĺpec	Databáza	Súbor	SQLite
`id`	1	2	3
`file_name`		airplanes.csv	books.db
`file_path`		gPTcN_airplanes.csv	rzElh_books.db
`mime_type`		text/csv	application/vnd.sqlite3
`driver`	postgresql		sqlite
`db`	postgres		rzElh_books.db
`usr`	postgres		
`pwd`	secret		
`host`	localhost		
`port`	5432		
`project_id`	1	1	1

Tabuľka 5.1 Príklad troch rôznych typov dátových zdrojov zaznamenaných v tabuľke.

nám udáva názov tabuľky. V ďalšom poli ukladáme cestu k súboru v úložisku. Nakoniec ukladáme typ dokumentu vo formáte MIME.[2]

Ďalšie polia sú relevantné pre databázy. Pole `driver` indikuje typ databázového systému, napr. `sqlite` alebo `postgresql`. Vo zvyšných poliach ukladáme prístupové údaje — názov databázy, užívateľ, heslo, adresa a port respektíve.

Príklady záznamov pre každý z troch typov dátových zdrojov uvádzame v tabuľke 5.1. Cesty k súboru sme zjednodušili — v skutočnosti je v nich uložená absolútna cesta. K pôvodnému názvu súboru pripájame náhodný reťazec, aby sme predišli kolíziám. Je zrejmé, že pri každom type záznamu máme veľa prázdnych hodnôt. Považujeme to za lepšie riešenie, ako mať rôzne tabuľky pre každý typ. Vo vyšších vrstvách nám to umožňuje s dátovými zdrojmi zachádzať jednotne.

Tabuľky a stĺpce

V časti 3.1.3 sme určili, že tabuľka môže pochádzať z dátového zdroja alebo byť zadaná užívateľom. V prvom prípade sa viaže na dátový zdroj jej pôvodu. Názov tabuľky musí byť unikátny v rámci projektu. Nedávalo by totiž zmysel generovať viac tabuliek s rovnakým názvom.

Tabuľka obsahuje niekoľko stĺpcov. Stĺpce majú unikátny názov v rámci tabuľky. Potrebujeme ukladať ich dátový typ — uložíme ho ako reťazec `integer`, `float`, `bool`, `string` alebo `datetime`. Pole `nullable` určuje, či stĺpec môže nadobúdať hodnotu `null`. Toto sú kľúčové informácie pre správne pridelenie generátora popísané v časti 4.2.5.

[2] `https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/MIME_types`

Stĺpce môžu pochádzať z dátového zdroja. V tom prípade sa viažu na pôvodný zdroj. Pole `reflected_column_idf` slúži na konkrétne určenie stĺpca pôvodu. Obsahuje presný identifikátor vo formáte `table.column`. Väzba tabuľky na dátový zdroj na tento účel nepostačuje, pretože aj do importovanej tabuľky môžeme manuálne pridávať stĺpce.

Generátor

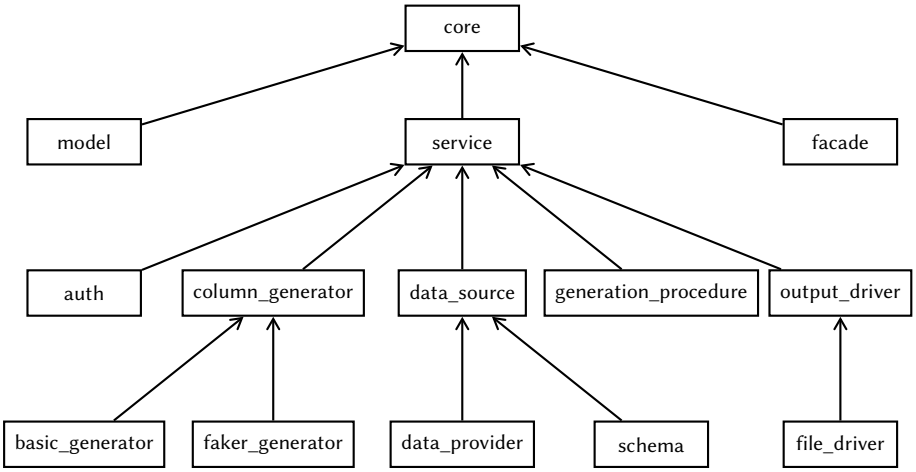
Generátor s jeho nastaveniami reprezentuje tabuľka `GeneratorSetting`. Držíme sa pritom návrhu z časti 4.2.1. Pole `name` určuje typ generátora. Množina povolených hodnôt záleží od sady definovaných generátorov. V poli `params` ukladáme parametre generátora vo formáte JSON. Každý generátor si sám určuje sadu, dátové typy a povolené hodnoty parametrov. Vo všeobecnosti je to JSON objekt, ktorého kľúče sú názvy parametrov. Jeden parameter spoločný pre mnoho generátorov je `null_frequency` určujúci frekvenciu hodnôt `null` vo výstupe.

V časti 4.2.1 sme popísali vzťahy medzi stĺpcami a generátormi. Z toho vyplýva charakter väzby tabuliek `GeneratorSetting` a `MetaColumn`. Navyše sme pridali väzbu nastavení generátora k tabuľke. Keď od generátora užívateľ odhlási všetky stĺpce, stále ho potrebujeme vedieť niekam priradiť. Prirodzene žiadame, že tabuľka priradená nastaveniu generátora je rovná tabuľke každého jeho stĺpca.

Integritné obmedzenia

Možné typy integritných obmedzení sme popísali v časti 2.2.1 a spôsob ich kontroly bol navrhnutý v časti 4.2.4. Integritné obmedzenia ukladáme do tabuľky `MetaConstraint`. Názov integritného obmedzenia je v poli `name`. Typ obmedzenia ukladáme do poľa `constraint_typ` ako jeden z reťazcov `primary`, `foreign`, `unique` alebo `check`. V prípade obmedzenia typu `CHECK` vložíme výraz do poľa `check_expression`.

Obmedzenia sa viažu so stĺpcami. Keďže jeden stĺpec môže mať viac obmedzení a jedno obmedzenie môže ovplyvňovať viacero stĺpcov, ide o vzťah many-to-many. Cudzie kľúče sa viažu s dvoma množinami stĺpcov — obmedzenými stĺpcami a referencovaný stĺpcami. Tiež musíme zachovávať poradie jednotlivých stĺpcov, čo je dôležité najmä v kontexte kompozitných cudzích kľúčov. Tieto vzťahy teda reprezentujeme dvoma tabuľkami. Obmedzenie stĺpca vyjadruje tabuľka `ColumnConstraint`. Číslo `index` značí poradie stĺpca v rámci sady obmedzených stĺpcov. Referencovaný stĺpec vyjadruje tabuľka `ReferenceConstraint`.



Obr. 5.6 Hierarchia modulov jadra. Koncové moduly boli vynechané v záujme prehľadnosti.

5.3 Jadro

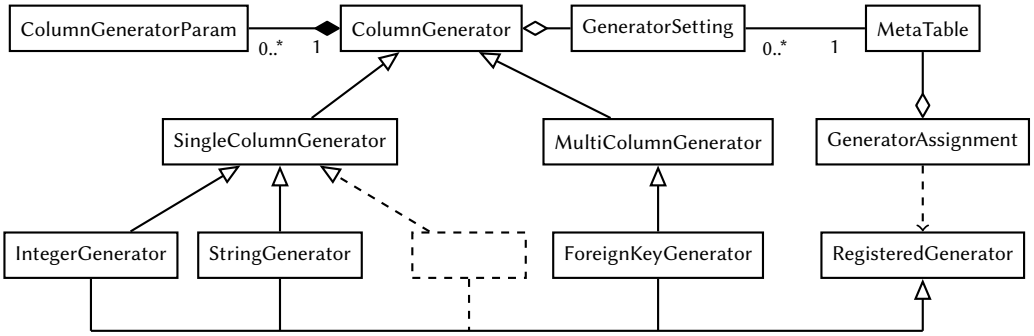
Jadro (`core`) je Pythonový balíček, ktorý poskytuje absolútnu väčšinu funkcionality dátového generátora ako bola popísaná v špecifikácii. Závisia na ňom balíčky `web` a `cli` implementujúce webové rozhranie a rozhranie v príkazovom riadku. Toto rozdelenie umožňuje využívať jadro ako samostatnú knižnicu.

Hierarchiu modulov jadra zobrazuje diagram 5.6. Modul `core.model` obsahuje objektové mapovanie databázových tabuliek popísaných v sekcii 5.2. Do modulu `core.service` sme zaradili procedúry spracovania, generovania a vypisovania dát. Nakoniec modul `core.facade` zjednodušuje rozhranie k operáciám poskytnutými predchádzajúcimi dvoma modulmi pre potreby webového servera.

V nasledujúcom texte sa budeme venovať podmodulom `core.service`. Vytváranie a overovanie hesiel a tokenov je zabezpečené modulom `auth`. Modulu `column_generator` implementujúcemu stĺpcové generátory sa venujeme v časti 5.3.1. Generovaniu dát využitím stĺpcových generátorov sa zaoberá modul `generation_procedure`. Využíva pritom výstupný ovládač z modulu `output_driver`. Tento mechanizmus je pokrytý časťou 5.3.2. Čítanie schém a záznamov dátových zdrojov poskytuje modul `data_source` opísaný v časti 5.3.3.

5.3.1 Generátory

Generátory sú implementované v module `core.service.column_generator`. Implementujeme objekty popísané v častiach 4.2.1 a 4.2.2. Obrázok 5.7 zobrazuje hierarchiu tried v tomto module.



Obr. 5.7 Diagram tried hierarchie stĺpcových generátorov

V strede stojí abstraktná trieda `ColumnGenerator`. V konštruktore prijíma inštanciu `GeneratorSetting` definujúceho nastavenia generátora. Bližšie sme ho opísali v sekcii 5.2. Stĺpcový generátor je zodpovedný za generovanie dát práve pre stĺpce naviazané na túto inštanciu `GeneratorSetting`.

Stĺpcový generátor obsahuje sadu definícií parametrov. Každý z nich je reprezentovaný dátovou triedou `ColumnGeneratorParam`. Parameter má názov, dátový typ a môže obsahovať obmedzenia množiny akceptovaných hodnôt. Parametrom určíme východiskovú hodnotu. Môžeme definovať metódu, ktorá odhadne hodnotu parametra zo vstupných dát.

S generátorom sa viaže niekoľko triednych atribútov. Každý z nich obsahuje:

- názov — automaticky odvodzujeme z názvu triedy metódou `name`,
- voliteľné obmedzenie pre jediný dátový typ metódou `only_for_type`,
- príznak `is_database_generated` určujúci, či je hodnota generovaná databázou (napr. surrogate key),
- príznak `supports_null` určujúci, či sa majú generovať `null` hodnoty,
- kategóriu `category` kvôli roztriedeniu na strane webovej aplikácie.

Kvôli deterministickému generovaniu vytvára základný generátor inštanciu štandardnej triedy `random.Random` uloženej v atribúte `_random`. Konfigurovaná je v metóde `seed`. Implementácie generátorov by mali pri náhodných javoch využívať túto inštanciu alebo preťažiť metódu `seed`.

Podľa množstva stĺpcov, za ktoré je generátor zodpovedný, sme ich rozdelili na dva typy. `SingleColumnGenerator` zjednodušuje rozhranie pre jednostĺpcové generátory. Stačí preťažiť abstraktnú metódu `make_scalar` vracajúcu vygenerovanú hodnotu. Ponúkame tiež automatickú podporu generovania `null` hodnôt — `make_scalar` sa sústredí iba na generovanie skutočných hodnôt.

Multistĺpcový generátor dediaci z triedy `MultiColumnGenerator` musí preťažiť abstraktnú metódu `make_dict`. Vygenerované hodnoty musia byť usporiadané v slovníku, kde kľúče sú názvy stĺpcov. Za generovanie `null` je zodpovedná trieda implementujúca túto metódu.

Priraďovanie generátorov

V sekcii 4.2.5 sme predstavili algoritmus priraďovania generátorov stĺpcom. Túto logiku implementujeme v triede `GeneratorAssignment`. Jej metóda `assign` pridelí stĺpcom v tabuľke inštancie `GeneratorSetting` v závislosti od dostupných generátorov.

Rozhranie generátora pozostáva zo statickej metódy `is_recommended_for`. Ako parameter dostane inštanciu `MetaColumn` a odpovie, či je daný generátor odporúčaný pre stĺpec. Východisková implementácia metódy hľadá zhodu v názve generátora a názve stĺpca. Ak je generátor vhodný iba pre jediný dátový typ, ktorý indikuje metódou `only_for_type`, toto obmedzenie je rešpektované implementáciou `GeneratorAssignment`.

Rozhranie multistĺpcového generátora si vyžaduje ďalšie metódy pre pokrytie algoritmu na obrázku 4.4. Jeho metóda `should_unite_with` dostane parameter typu `MetaColumn`. Vráti pravdivostnú hodnotu označujúcu, či má byť táto inštancia spojená s ďalším stĺpcom, ktorému je odporúčaný rovnaký typ generátora. Ak má byť spojená, zavolá sa metóda `unite_with`.

`RegisteredGenerator` by mal byť priamym rodičom každého generátora. Dedením z tejto triedy sa generátora zaregistruje do zoznamu generátorov, ktoré sa zvažujú pri automatickom prideľovaní. Generátor sa týmto tiež stane dostupným z webovej aplikácie.

Implementácia generátora

V module `core.service.column_generator.basic_generator` sme implementovali generátory pre základné dátové typy a niektoré štatistické rozdelenia. Je dôležité, aby boli importované v správnom poradí, pretože v tom poradí sa registrujú a ovplyvňuje automatické priraďovanie. Pre doménové generátory sme v časti 4.2.2 zvolili knižnicu Faker a implementácie generátorov sa nachádzajú v module `core.service.column_generator.faker_generator`.

Rozhranie generátora považujeme za najdôležitejšie rozhranie v aplikácii. Dôležitosť vychádza z toho, že my sami implementujeme mnoho základných generátorov a očakávame tu rozšírenia od užívateľov. Venovali sme mu preto najviac pozornosti, aby bolo ľahko čitateľné a bez zbytočného kódu navyše.

Minimalistickú implementáciu uvádzame vo výpise 29. Táto trieda sa musí nachádzať v ľubovoľnom importovanom module. Generátor sa objaví vo výbere

Výpis kódu 29 Implementácia generátora vypisujúca reťazec.

```
class HelloGenerator(RegisteredGenerator,
                     SingleColumnGenerator[str]):
    def make_scalar(self, db: GeneratedDatabase) -> str:
        return 'hello'
```

Výpis kódu 30 Implementácia generátora vypisujúca reťazec z parametra.

```
class GreetingGenerator(RegisteredGenerator,
                        SingleColumnGenerator[str]):
    @parameter
    def greeting(self) -> str:
        return 'hi'

    def make_scalar(self, db: GeneratedDatabase) -> str:
        return self.greeting
```

generátorov vo webovej aplikácii vďaka dedeniu z triedy `RegisteredGenerator` (je potrebné obnoviť stránku). Nachádza sa v kategórii `General` pre stĺpce s dátovým typom reťazec. Pomenovanie triedy mu dáva meno `Hello`.

Metóda `make_scalar` iba vracia reťazec `'hello'`. `GeneratedDatabase` obsahuje doteraz vygenerovanú sadu dát. Užitočné je to najmä pri implementácií cudzích kľúčov. Typová anotácia výstupnej hodnoty je veľmi dôležitá — tento generátor je vhodný iba pre stĺpce typu reťazec. Trieda `SingleColumnGenerator` sa automaticky postará o zabezpečenie tohto obmedzenia vďaka uvedenej anotácii.

Parametrizovaný generátor

Rozšírme predchádzajúci príklad, aby vypisoval reťazec zadaný užívateľom miesto fixného reťazca. Tento príklad uvádzame vo výpise 30. Rozhranie je inšpirované vstavaným dekorátorom `@property` v štandardnej knižnici Pythonu. Parameter definujeme dekorátorom `@parameter`. Dekorovaná metóda určuje názov parametra. Návratová hodnota bude východisková hodnota parametra.

Hodnota parametra je dostupná z atribútu so zhodným menom ako názov parametra. Tento atribút je naviazaný na hodnotu parametra v tabuľke `GeneratorSetting`. Je teda možné nielen čítanie, ako to robíme v metóde `make_scalar`, ale aj písanie.

Anotácia návratovej hodnoty je znova veľmi dôležitá. Určuje dátový typ parametra. Zvyšok systému garantuje, že hodnota parametra prijatá od užívateľa bude tohto typu. Navyše to môže ovplyvniť typ vstupného komponentu na strane

Kľúčové slovo	Typ hodnoty	Význam
`allowed_values`	`list`	zoznam povolených hodnôt
`min_value`	zhodný s parametrom	minimálna hodnota
`max_value`	zhodný s parametrom	maximálna hodnota
`greater_equal_than`	`str`	nerovnosť medzi hodnotami parametrov

Tabuľka 5.2 Dostupné validačné pravidlá pre parametre stĺpcových generátorov.

webovej aplikácie.

Podporujeme dátové typy `int`, `float`, `bool`, `str` a `datetime`. Sú to rovnaké typy ako podporované typy stĺpcov. Takto to bolo zvolené v časti 2.2.3. Pridaním typovej definície do modulu `core.types` je možné pridať podporu ďalším typom.

Implementovali sme možnosť obmedziť množinu prijímaných hodnôt parametrov. Obmedzenie pridáme ako pomenovaný argument do dekorátora `@parameter` Ak by sme chceli vo výpise 30 dovoliť užívateľovi zadať len jeden z preddefinovaných reťazcov, pridáme do dekorátora validačné pravidlo.

```
@parameter(allowed_values=['hi', 'yo', 'sup'])
```

Systém zaručuje vynútenie tohto pravidla. V kóde generátora validáciu nemusíme riešiť. Validáciu zabezpečuje modul `core.column_generator.params`. Toto pravidlo má vplyv aj na webovú aplikáciu — zobrazí sa zoznam s povolenými hodnotami. Zoznam všetkých validačných pravidiel uvádzame v tabuľke 5.2. Miesto toho, aby sme zamietali nevyhovujúce hodnoty, snažíme sa ich opravovať. Napríklad ak je hodnota parametra zhora obmedzená a užívateľom zadaná hodnota prekročí maximum, znížime ju na toto maximum.

Odhad hodnoty parametra

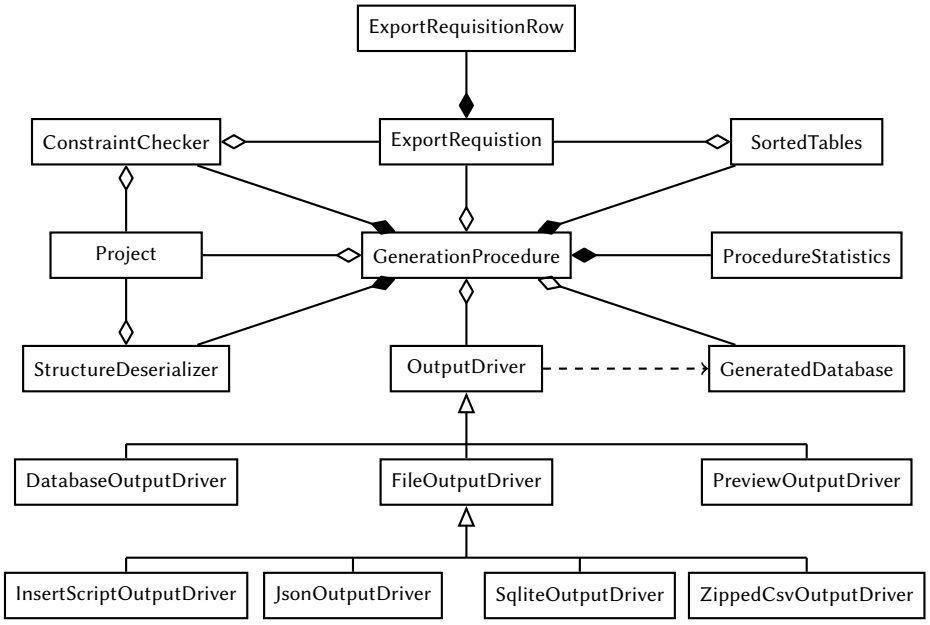
Dekorátor `@parameter` sme prirovnali k vstavenému dekorátoru `@property`. Podobným spôsobom ako funguje `@property.setter` priradíme k parametru metódu odhadujúcu jeho hodnotu. Táto metóda musí mať rovnaký názov ako názov parametra, rovnakú anotáciu návratovej hodnotu. Ako argument dostane inštanciu triedy `DataProvider` a vráti odhadnutú hodnotu zodpovedajúceho parametra. Ide o implementáciu mechanizmu navrhnutého v časti 4.2.1.

Vo výpise 31 uvádzame kompletný generátor celých čísel z uzavretého intervalu. Má dva parametre, `min` a `max`, určujúce rozsah intervalu. Validačným pravidlom sme zabezpečili, že dostaneme korektný interval. Dokumentačný reťazec má význam nie len pre programátora, ale zobrazíme ho aj v grafickom prostredí.

Minimálnu hodnotu odhadujeme jednoducho tak, že nájdeme minimum v zdrojovom stĺpci. Trieda `DataProvider` nám ponúka iterátor po hodnotách v

Výpis kódu 31 Implementácia generátora celých čísel z intervalu s odhadom parametrov.

```
class IntegerGenerator(RegisteredGenerator,
                       SingleColumnGenerator[int]):
    """Generate integers uniformly from a closed interval."""

    @parameter
    def min(self) -> int:
        return -100

    @parameter(greater_equal_than='min')
    def max(self) -> int:
        return 100

    @min.estimator
    def min(self, provider: DataProvider) -> int:
        return provider.estimate_min()

    @max.estimator
    def max(self, provider: DataProvider) -> int:
        return provider.estimate_max()

    def make_scalar(self, db: GeneratedDatabase) -> int:
        return self._random.randint(self.min, self.max)
```



Obr. 5.8 Diagram tried generačnej procedúry.

zdrojovom stĺpci metódou `scalar_data`. Naivná implementácia by mohla hľadať minimum prostredníctvom tohto iterátora. Efektívnejšie je nechať jednotlivé dátové zdroje preťažiť bežné operácie ako hľadanie minima. V prípade, že čerpáme dáta z databázy, metóda `estimate_min` vyprodukuje dotaz priamo dopytujúci sa po minime v rámci databázového systému.

```
SELECT MIN(source_column_name) FROM source_table_name
```

Maximálnu hodnotu odhadujeme analogicky. Dátovým zdrojom sa budeme venovať podrobnejšie v časti 5.3.3.

5.3.2 Generačná procedúra

Základné princípy generovania dát sme rozoberali v časti 4.2.3. Implemetuje ich trieda `GenerationProcedure` zobrazená v strede diagramu tried 5.8. Niektoré časti funkcionality sme rozdelili do oddelených tried. `GenerationProcedure` ich využíva ako svoje zložky.

Procedúra beží v kontexte jedného projektu, ktorý obsahuje sadu tabuliek. Vo vstupe od užívateľa obdržíme niekoľko záznamov (`ExportRequisitionRow`), z ktorých každý obsahuje názov tabuľky, počet želaných riadkov a seed. Tieto záznamy sú zoskupené v štruktúre `ExportRequisition`. Do výstupu nemusia byť zahrnuté všetky tabuľky projektu.

V sekcii 4.2.4 sme popísali spôsob zoraďovania tabuliek. Toto triedenie imple-

mentuje trieda `SortedTables`. V konštruktore dostane zoznam tabuliek projektu (`MetaTable`) a vstupné požiadavky (`ExportRequistion`). Z tabuliek projektu vyberá iba relevantné — iba tie, pre ktoré sa majú generovať záznamy. Metóda `get_order` vráti zoradený zoznam tabuliek.

Trieda `ConstraintChecker` implementuje kontrolu integrity popísanú v časti 4.2.4. V konštruktore dostane inštanciu projektu. Jej verejné rozhranie pozostáva z metódy `check_row`, ktorá pre riadok odpovie, či spĺňa požiadavky integrity. Ak áno, riadok je vložený prostredníctvom výstupného ovládača. Ak je riadok akceptovaný aj výstupným ovládačom, generačná procedúra zavolá s výsledným riadkom metódu `register_row`. Táto interakcia bola naznačená v algoritme 1.

Algoritmus 1 tiež naznačoval, že musíme počítať celkový počty generovaných riadkov a počty úspešne vložených riadkov. Na základe tolerancie určujeme, kedy generovanie ukončíme. Tieto princípy sú zapúzdrené v triede `ProcedureStatistics`.

V dátovej štruktúre `GeneratedDatabase` priebežne ukladáme vygenerované riadky. Túto štruktúru predávame stĺpcovým generátorom, ktoré ju môžu využívať napríklad na efektívne generovanie cudzích kľúčov. Pre niektoré výstupné ovládače môže byť tiež výhodné použiť túto dátovú štruktúru na vytvorenie výstupu.

Výstupné ovládače

Výstupný ovládač predstavuje jednotné rozhranie rôznych foriem výstupu. Základné požiadavky naň sme opísali v časti 4.2.3. Obsahuje dva triedne atribúty:

- `is_interactive` — príznak označujúci či ovládač generuje hodnoty ako surrogate key,
- `cli_command` — reťazec, ktorým je ovládač využiteľný z príkazového riadku.

Implementácia ovládača musí preťažiť nasledujúce abstraktné metódy:

- `start_run` — volaná pred začatím generovania, je to miesto na alokovanie zdrojov (napr. zahájenie spojenia s databázou),
- `end_run` — volaná na konci generovania, v argumente sa podáva inštancia `GeneratedDatabase`,
- `switch_table` — príprava na vklad do tabuľky, v argumente sú podané inštancie SQLAlchemy `Table` a `MetaTable`,

- `insert_row` — vloženie riadku, pri úspechu vráti ovládač výsledný riadok a pri neúspechu vráti `None`.

Metóda `insert_row` je miesto nielen na vkladanie do výstupnej databázy, ale aj na dočítanie databázou vygenerovaných hodnôt. Z tejto metódy musí byť vrátený kompletný riadok. V prípade neinteraktívnych ovládačov je dostačujúce vrátiť pôvodný riadok z argumentu.

Najjednoduchší ovládač je `PreviewOutputDriver`. Nemá žiadny účinok, akceptuje každý riadok. Použijeme ho, keď sa zaujímame iba o výslednú štruktúru `GeneratedDatabase`.

Vkladanie do databázy je implementované v triede `DatabaseOutputDriver`. Vďaka využitiu štruktúr SQLAlchemy je jeho implementácia priamočiara. Zložitejšia časť je preklad internej štruktúry projektu na objekty, ktoré používa SQLAlchemy. Potrebujeme z našich tabuliek (`MetaTable`) vyrobiť inštancie `Table`. To zahŕňa konštrukciu stĺpcov a integritných obmedzení. Toto zabezpečuje trieda `StructureDeserializer`. Deje sa to priamo v generačnej procedúre. Dôvodom je, že tieto štruktúry by mohli využívať aj iné ovládače.

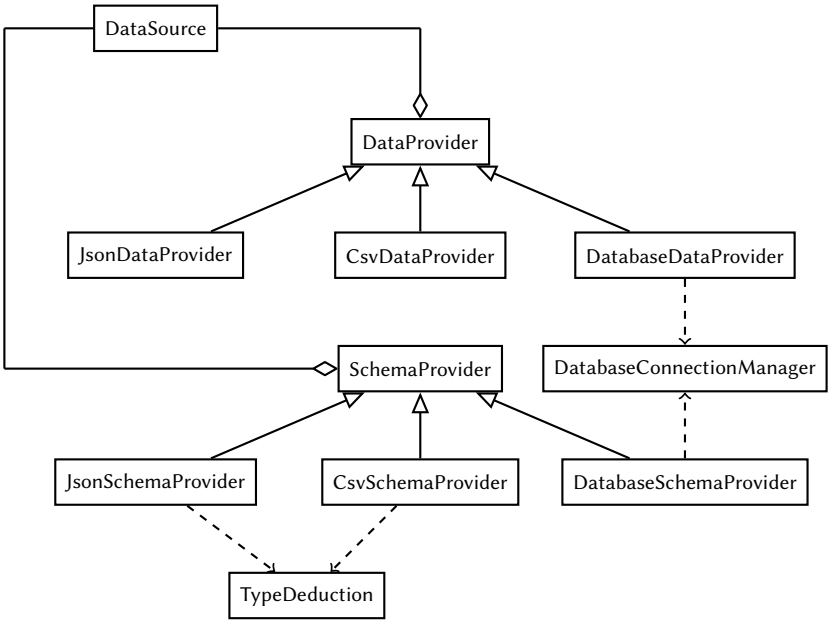
Databázový výstupný ovládač dostane v konštruktore prístupy k databáze vo forme objektu `DataSource` alebo URL. Pri zmene tabuľky vyhľadá primárny kľúč. Tento primárny kľúč sa po vkladaní dodatočne nahráva s použitím funkcie SQLAlchemy. Musíme reagovať aj na prípadné chyby. Ak nám výstupná databáza zahlási chybu týkajúcu sa integrity, interpretujeme to tak, že riadok nebol prijatý a procedúra môže pokračovať ďalej. Ostatné chyby sú fatálne.

Pre potreby vytvárania súborov v určitom formáte sme implementovali ďalšiu abstraktnú triedu `FileOutputDriver`. Rozširuje výstupný ovládač najmä o abstraktnú metódu `dump`, ktorá vráti obsah výstupného súboru. Návratová hodnota môže byť reťazec pre textové formáty alebo `bytes` napríklad pre zipovaný archív. Ovládač by teda nemal súbor ukladať na lokálny disk, iba vrátiť jeho obsah.

Všetky triedy dediace z `FileOutputDriver` sú automaticky zaregistrované ako výstupné ovládače pre potreby webového serveru a príkazového riadku. Stačí teda implementovať abstraktné metódy a ovládač sa zobrazí v grafickom prostredí a bude dostupný aj ako príkaz CLI. Pomáha tu trieda `FileOutputDriverFacade`. Sprístupňuje zoznam registrovaných ovládačov a naopak dokáže vytvoriť inštanciu ovládača podľa názvu. Názvy sú odvodené z názvu triedy.

5.3.3 Dátové zdroje

Po nahraní dátového zdroja je vytvorený v databáze záznam `DataSource`. Obsahuje buď cestu k lokálnej kópii súboru alebo prístupové údaje k databáze.



Obr. 5.9 Diagram vybraných tried modulu spracovávajúceho dátové zdroje.

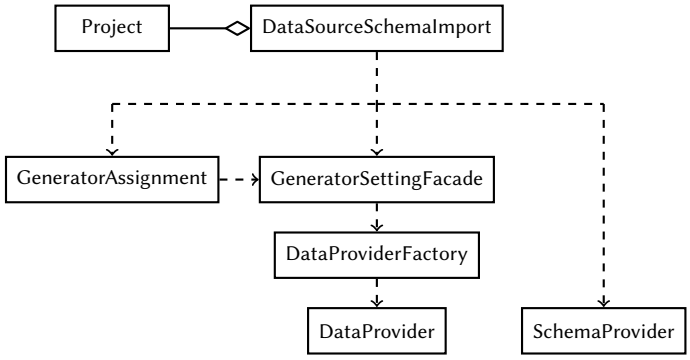
Z dátového zdroja budeme čítať štruktúru alebo tabulárne záznamy. Na diagrame 5.9 zobrazujeme triedy zaoberajúca sa týmito problémami. Sídlia v module `core.service.data_source`.

Čítanie schémy

Rozhranie pre čítanie schémy definuje trieda `SchemaProvider`. V konštruktore prijíma inštanciu `DataSource`. Má jedinú abstraktnú metódu `read_structure`, ktorej úloha je vrátiť zoznam tabuliek (`MetaTable`) zo zdroja. Na tieto tabuľky majú byť naviazané stĺpce a integritné obmedzenia.

Triedy `JsonSchemaProvider` a `CsvSchemaProvider` čítajú súbory z disku. Ani jeden z týchto formátov neobsahuje typové údaje o svojich stĺpcoch. Musíme teda čítať záznamy a typy odvodzovať z nich. Typy kontroluje a odvodzuje trieda `TypeDeduction`. Metódou `next_row` ju informujeme o prečítanom riadku z tabuľky a nakoniec dostaneme slovník typov pomocou metódy `get_types`. V prípade formátu JSON zisťujeme aj či stĺpec obsahuje hodnoty `null`. Na túto otázku odpovedá metóda `is_nullable` pre názov stĺpca. Zabezpečujeme aj kontrolu ostatných požiadaviek deklarovaných v sekcii 2.2.

Trieda `DatabaseDataProvider` dostane prístupy k databáze prostredníctvom inštancie `DataSource`. Ďalšou triedou `DatabaseConnectionManager` predanou do konštruktoru je riadené spojenie s databázou. Tento princíp nám umožňuje zdieľať jedno pripojenie medzi viacerými časťami aplikácie. Štruktúru



Obr. 5.10 Diagram tried slúžiacich na import schémy a čítanie zdrojových dát.

databázy čítame funkciami SQLAlchemy.

Databáza je v súčasnosti jediný zdroj, z ktorého čítame integritné obmedzenia. Tieto obmedzenia musia byť naviazané na tabuľky a tiež ich musíme zviazať s obmedzenými a referencovanými stĺpcami. Riešime to tak, že najprv skonštruujeme len tabuľky (`MetaTable`) a stĺpce (`MetaColumn`) bez obmedzení. Následne vyrobíme slovník týchto inštancií stĺpcov podľa názvu tabuľky a názvu stĺpca. Potom iterujeme po tabuľkách a zdrojových obmedzeniach. Vytvárame obmedzenia `MetaConstraint`, ktoré vďaka slovníku efektívne naviažeme na stĺpce.

Čítanie záznamov

Rozhranie pre čítanie záznamov definuje trieda `DataProvider`. V konštruktore dostane inštanciu `DataSource` a zoznam identifikátorov. Každý z týchto identifikátor sa skladá z názvu tabuľky a názvu stĺpca. Inštancia `DataProvider` zo zdrojových dát potom bude vyberať n-tice hodnôt na základe identifikátorov. Iterátor po n-ticiach vracia abstraktná metóda `vector_data`. V prípade, že dodáme práve jeden identifikátor, môžeme využiť abstraktnú metódu `scalar_data` vracajúcu iterátor hodnôt v jedinom stĺpci.

Okrem vyššie spomenutých metód implementuje `DataProvider` ďalšie metódy na spracovanie dát. V implementácií generátora vo výpise 31 sme predstavili metódy vracajúce minimum a maximum. Ako sme už spomínali, chceme nechať jednotlivým špecializáciám `DataProvider` možnosť preťaženia týchto operácií. Najmä v prípade databáz môže byť efektívnejšie nechať operáciu vykonať databázový systém — vyhneme sa prenosu kompletných dát a dostaneme priamo výsledok.

Integrácia so zvyškom systému

Popísali sme princípy nízkoúrovňových operácií vykonateľných s dátovými zdrojmi. Akcie zadávané na pokyn užívateľa sú:

- odhad hodnôt parametrov generátora,
- import schémy zo zdroja do projektu.

V časti 5.3.1 sme sa stretli s triedou `DataProvider` použitou na odhad parametrov. Musíme zabezpečiť, aby sme k danému dátovému zdroju skonštruovali vhodnú špecializáciu `DataProvider`. Ako to naznačuje obrázok 5.10, zabezpečuje to trieda `DataProviderFactory`. Táto trieda obdrží v konštruktore zoznam stĺpcov pridelených niektorému generátoru. Metóda `find_provider` nájde, skonštruuje a vráti vhodnú implementáciu. Zvažuje pritom hodnoty atribútov inštancie `DataSource` zviazanej so vstupnými stĺpcami.

Trieda `GeneratorSettingFacade` ponúka zjednodušené rozhranie pre odhad parametrov. Konštruujeme ju len s inštanciou `GeneratorSetting`. Konštrukciu generátora zabezpečí v metóde `maybe_estimate_params`. Pokiaľ sú jeho stĺpce zviazané s dátovým zdrojom, vykoná odhad parametrov.

Import schémy zabezpečuje trieda `DataSourceSchemaImport`. Operácia pozostáva zo štyroch fáz.

1. prečítanie štruktúry zdroja,
2. spojenie novej štruktúry so súčasnou štruktúrou projektu,
3. automatické priradenie generátorov importovaným tabuľkám,
4. odhad parametrov generátorov importovaných tabuliek.

Čítanie štruktúry zabezpečuje `SchemaProvider`. Zase musíme skonštruovať vhodnú špecializáciu. Nové tabuľky potom pridáme do projektu. V prípade kolízie názvov prepíše nová tabuľka starú. Využitím pomocnej triedy `GeneratorAssignment` priradeníme generátory importovaným tabuľkám. Túto triedu sme bližšie popisovali v časti 5.3.1. V poslednej fáze vykonáme odhad hodnôt parametrov pre každý generátor v importovanej tabuľke. K tomu znova využijeme triedu `GeneratorSettingFacade`.

5.3.4 Rozšírenia

V tomto texte načrtneme ako rozšíriť podporu dátových typov a databázových systémov.

V časti 2.2.3 sme definovali množinu podporovaných typov. Tieto typy sú deklarované v module `core.service.types`. Rozšírenie o nový dátový typ by si vyžadovalo pridanie práve do tohto modulu. Obmedzení sme však podporovanými typmi SQLAlchemy.[3] Musíme tiež vyriešiť, akým spôsobom s týmito typmi interagujú výstupné ovládače z časti 5.3.2. Zmeny v CLI a webovom serveri nie sú potrebné. Na strane webovej aplikácie stačí pridať nový typ medzi možnosti vo formulári pri tvorbe nového stĺpca.

Podporované databázové systémy sú SQLite a PostgreSQL. Pridanie ďalšieho podporovaného systému si vyžaduje nainštalovanie ovládača pre SQLAlchemy.[4] Toto postačuje na generovanie prostredníctvom CLI. Aby ovládač bol akceptovaný API, je potrebné zaregistrovať ho v triede `DataSourceConstants`. Potom bude možné pridať voľbu ovládača do možností vo formulári na strane webovej aplikácie.

5.4 Webový server

Webový framework Flask sme predstavili v časti 4.1.2. Prebrali sme aj knižnicu na serializáciu/deserializáciu objektov marshmallow. Ukázali sme aj definíciu schémy endpointu pomocou knižnice flasgger.

Množstvo kódu sme v rámci webového serveru obmedzovali na minimum kvôli maximálnej modularite. Väčšina endpointov teda pozostáva iba zo zbierania vstupov a volania funkcie v jadre. Časť jadra určená najmä na použitie z webového serveru je modul `core.facade`. Obsahuje triedu pre každý základný objekt v internej databáze. Tieto triedy ponúkajú operácie relevantné k daným objektom.

Životný cyklus požiadavky na server

Pri spracovaní požiadavky potrebujeme alokovať niekoľko zdrojov a tiež zabezpečiť ich uvoľnenie. Ide najmä o tieto zdroje:

- pripojenie k internej databáze,
- inštancia `User` pre prihláseného užívateľa,
- pripojenia k externým databázam.

Pri využívaní funkcií ORM potrebujeme objekt typu `Session`. Ako sme to vysvetľovali v časti 4.1.3, tento objekt si drží otvorenú transakciu. Po vybavení požiadavky sa musíme uistiť, že transakcia je ukončená.

[3] `https://docs.sqlalchemy.org/en/14/core/type_basics.html`
[4] `https://docs.sqlalchemy.org/en/14/dialects/index.html`

Využijeme tu funkcie frameworku Flask. Sprístupňuje nám objekt pod názvom g, do ktorého môžeme ukladať objekty ako do slovníka. Budeme mať teda jednu funkciu vracajúcu objekt `Session` z tohto slovníka. Ak neexistuje, tak je vytvorený a uložený. Ďalej zadefinujeme funkciu, ktorá sa zavolá po vybavení požiadavky a zabezpečí uzatvorenie transakcie. Tieto funkcie sú súčasťou modulu `web.service.database`.

Inštanciu prihláseného užívateľa tiež môžeme ukladať do objektu g. Tento objekt si nevyžaduje špeciálnu akciu na konci požiadavky, Flask sa postará automatické uvoľnenie.

Nevýhoda objektu g je jeho naviazanosť na Flask. Preto sme sa rozhodli implementovať vlastný dependency injector. Celý injector vieme tiež uložiť v objekte g. Relevantné metódy sa nachádzajú v module `web.service.injector`.

Už skôr spomenutou triedou `DatabaseConnectionManager` sú riadené pripojenia k externým databázam. Stačí teda každému objektu využívajúcemu externé databázy podsunú jednu spoločnú inštanciu. Po vybavení požiadavky ukončíme spojenia s databázami volaním metódy `clean_up`.

Po vybavení požiadavky zavoláme na injectore metódu `clean_up`. Táto metóda zavolá `clean_up` na všetkých spravovaných objektoch — vrátane vyššie spomenutej `DatabaseConnectionManager`.

Vlastný dependency injector

Po vzore injectoru z Angularu popísanom v časti 4.1.4 implementujeme vlastný. Budeme tiež využívať typové anotácie konštruktoru. Umiestnili sme ho do modulu `core.service.injector`.

Interne si ukladá v slovníku pre každý typ najviac jednu inštanciu. Verejné rozhranie sa skladá z troch jednoduchých metód:

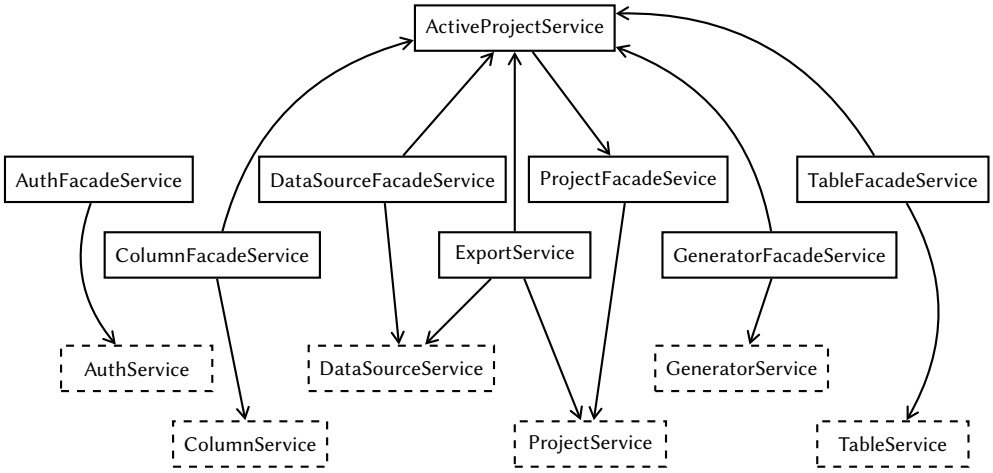
- `provide` — zadáme typ a inštanciu, ktorú si injector uloží do slovníka,
- `get` — vrátenie inštancie pre zadaný typ, buď zo slovníka alebo rekurzívne skonštruovanej,
- `clean_up` — uvoľnenie slovníka a prípadné zavolanie `clean_up` na inštancie implementujúce rozhranie `HasCleanUp`.

Dependency injection vieme využiť aj pri základných typoch ako reťazec. Ukazujeme to vo výpise 32. Zo základného typu vytvoríme nový typ pomocou štandardnej funkcie `NewType`. Služba zaujímajúca sa o túto hodnotu pridá do konštruktoru parameter `secret_key: SecretKey`.

Praktické využitie nastáva pri službe `TokenService` zodpovednej za vytváranie a overovanie tokenov JWT. Potrebujeme k nej dostať reťazec využívaný na

Výpis kódu 32 Depedency injection základných typov.

```
from core.service.injector import Injector
from typing import NewType

SecretKey = NewType('SecretKey', str)
injector = Injector()
injector.provide(SecretKey, 'top secret')
print(injector.get(SecretKey))  # 'top secret'
```



Obr. 5.11 Prehľad závislostí medzi vybranými službami webovej aplikácie. Služba, na ktorú šípka ukazuje, je v argumentoch konštruktora služby, od ktorej šípka vedie. Vrcholy s čiarkovaným ohraničením predstavujú služby z vygenerovaného kódu.

kódovanie a dekódovanie tokenu. Tento reťazec je súčasťou konfigurácie aplikácie. Inštanciu `Injector` naplníme na začiatku vybavenia požiadavky konfiguračnými hodnotami. `TokenService` si potom necháme skonštruovať injectorom.

5.5 Webová aplikácia

Na diagrame 5.11 zobrazujeme prehľad najdôležitejších služieb na strane webovej aplikácie. Čiarkované služby na spodnej časti sú kompletne vygenerované zo špecifikácie API. Každá z nich dostane názov podľa nami zvoleného *tagu*. Tag prideľujeme každému endpointu. Pridelené tagy približne zodpovedajú tomu, s akým databázovými objektami dané endpointy pracujú.

Vo vygenerovanej službe dostaneme metódy pre každý endpoint vzťahujúci sa k tagu. Tieto metódy majú pripravené typované parametre. V tele vyvolajú HTTP požiadavku a vrátia `Observable` s typovaným výsledkom. Pre každý end-

point môžeme v skutočnosti využiť dve metódy — jedna vráti iba dáta, druhá obsahuje aj HTTP hlavičky odpovede, čo je užitočné pri sťahovaní súboru.

Aby sme naplno využili výhody single-page aplikácie, budeme centrálne spravovať stav. To bude hlavná úloha ostatných služieb viditeľných na obrázku. Keď sa užívateľ prihlási, dostaneme o ňom potrebné informácie raz. Pri zmene navigácie už nepotrebujeme znova čakať na výsledok z API – využívame stále to, čo máme uložené. Podobne spravujeme stav projektu. V ďalších odsekoch tento mechanizmus upresníme.

Prihlásený užívateľ

V súvislosti s užívateľom ponúkame nasledujúce endpointy: registrácia, prihlásenie, prečítanie stavu užívateľa a zmena prihlasovacích údajov. Každý z nich vracia rovnaký objekt typu `UserView`. Pre tieto endpointy máme vygenerované metódy v triede `AuthService`. Služba `AuthFacadeService` k tomu pridáva správu stavu užívateľa. Jej hlavná vlastnosť je `user$: BehaviorSubject<UserView>`. Keď sa ďalší komponent alebo služba bude zaujímať o prihláseného užívateľa, tak bude využívať práve túto vlastnosť.

V časti 4.1.4 sme predstavili `Subject`. `BehaviorSubject` rozširuje jeho funkcionalitu o „súčasnú hodnotu“.[5] Túto súčasnú hodnotu (správu) pošle novým odoberateľom okamžite. To je presne princíp, ktorý sa hodí na situáciu prihláseného užívateľa. Niekedy sa zaujímame len o užívateľa prihláseného práve v tomto momente. Inokedy môže byť žiadúce reagovať na zmenu prihláseného užívateľa (teda prihlásenie, registráciu alebo odhlásenie).

Služba `AuthFacadeService` implementuje metódy `register` a `login` volajúce príslušné endpointy prostredníctvom `AuthService`. Po úspešnom prihlásení notifikujeme pozorovateľov poslaním nového stavu cez vyššie zmienenú vlastnosť `user$`. Neprihláseného užívateľa sme sa rozhodli reprezentovať hodnotou `null`. Pridali sme aj metódu `logout`, ktorá jednoducho emituje stav `null`.

Autentifikáciu riešime pomocou *tokenu JWT*.[6] Tento token vydávame pri prihlásení alebo registrácii na strane API. Prihlásený klient token pripája v HTTP hlavičke `Authorization` pri každej požiadavke. O pridávanie hlavičky do požiadaviek sa stará `AuthInterceptorService`.

Po prihlásení uložíme token do *local storage*.[7] `AuthInterceptorService` jednoducho pridáva autorizačnú hlavičku vždy, keď v local storage nájde token. Pri nábehu aplikácie vyskúšame z API prečítať údaje o prihlásenom užívateľovi. Ak sa nám to podarí, tak dostaneme údaje užívateľa. Ak sa nám to nepodarí, znamená to, že nemáme platný token a teda nie sme prihlásení. V metóde `logout`

[5] `https://www.learnrxjs.io/learn-rxjs/subjects/behaviorsubject`

[6] `https://tools.ietf.org/html/rfc7519`

[7] `https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage`

jednoducho zmažeme uložený token.

Týmto dosiahneme trvanie prihlásenia aj po opustení stránky a bez použitia cookies. Komplexnejšie riešenie by mohlo zneplatňovať tokeny pri odhlásení a rozlišovať *access token* a *refresh token*.[8]

Aktívny projekt

Služba `ProjectFacadeService` spravuje zoznam užívateľových projektov prostredníctvom vlastnosti `projects$: BehaviorSubject<ProjectView[]>`. Obsahuje funkciu sprístupňujúcu vytvorenie nového projektu. Pritom využije volanie API, výsledný projekt pridá do zoznamu a notifikuje odoberateľov zmien. Tento zoznam je špecifický pre prihláseného užívateľa. Musíme teda odoberať zmeny prihláseného užívateľa a pri prihlásení zoznam obnoviť z API. Po odhlásení ho zmažeme.

Väčšinou pracujeme iba s jedným projektom, nie celým zoznamom súčasne. Cesty týkajúce sa projektu obsahujú `/project/:id`, kde `:id` je identifikátor projektu. Využívame viacvrstvové smerovanie. Komponent `ProjectComponent` je súčasťou každej stránky týkajúcej sa projektu. Ako príklad zoberme cestu `/project/:id/preview` zobrazujúcu `AppComponent`, `ProjectComponent` a v ňom sa zobrazí `PreviewComponent`.

Vytvorili sme službu `ActiveProjectService`. Jej úloha je spravovať práve jeden aktívny projekt. Zmeny v projekte sú dostupné z verejnej vlastnosti `project$: BehaviorSubject<ProjectView>`. Aktívny projekt existuje práve vtedy, keď aktivovaná cesta obsahuje `/project/:id`.

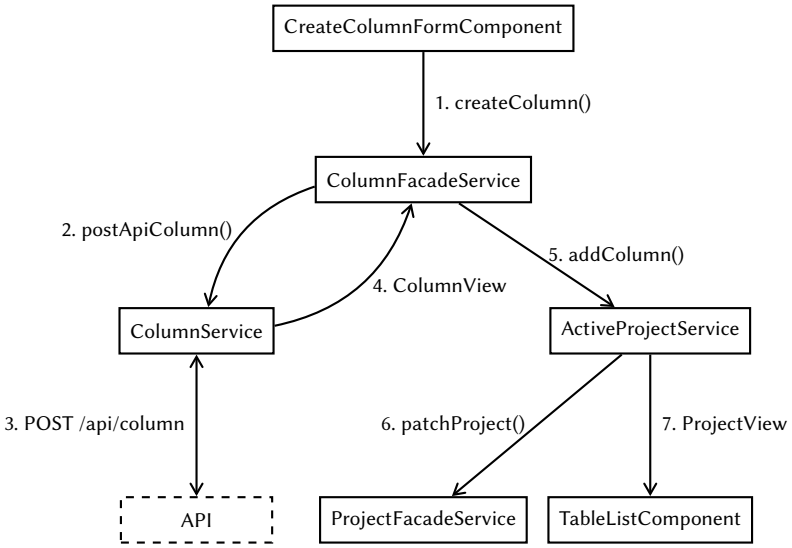
Úloha `ActiveProjectService` je centrálne spravovať stav aktívneho projektu v rámci webovej aplikácie. Poskytuje metódy meniace stav — pridanie a odoberanie stĺpcov, tabuliek či generátorov. Zmeny nezapisuje do API. Posúva ich však do zoznamu, ktorý udržuje `ProjectFacadeService`.

Ďalšie služby ako `ColumnFacadeService` k tejto funkcionalite pridávajú API. Konkrétne táto služba ponúka metódu vytvárajúcu stĺpec a metódu odoberajúcu stĺpec. Každá z nich spraví najprv požiadavku na API a po úspechu zmení stav v `ActiveProjectService`.

Dátový tok po vytvorení stĺpca ilustruje diagram 5.12. Užívateľ vytvorí stĺpec z formulára `CreateColumnFormComponent`. Výsledný stĺpec a chyby sa v skutočnosti propagujú až k pôvodnému formuláru, ktorý má šancu reagovať. Ako prijímateľ zmeny stavu projektu je uvedený komponent zobrazujúci tabuľky v projekte `TableListComponent`. Prijímateľov zmien môže byť v skutočnosti viacero.

Tento princíp sa opakuje aj v ostatných službách závisiacich na aktívnom

[8] `https://tools.ietf.org/html/rfc6749#section-1.4`



Obr. 5.12 Prehľad toku dát pri vytvorení nového stĺpca.

projekte. Vytvorenie tabuľky zabezpečuje `TableFacadeService` a výsledná entita je typu `TableView`. Mazanie stĺpcov alebo tabuliek môže spôsobiť zmazanie integritného obmedzenia v inej tabuľke — vtedy z API vraciame kompletnú schému projektu. Služba `ExportService` sa stará o spustenie generovania dát alebo export konfigurácie. Stav aktívneho projektu iba číta.

Kapitola 6

Užívateľská dokumentácia

V tejto kapitole poskytneme návod pre koncového užívateľa. Na začiatku predstavíme grafické prostredie. Potom ukážeme, ako je možné spustiť generovanie pomocou CLI. Na konci ukážeme niekoľko scenárov načrtajúcich ako využiť aplikáciu na dosiahnutie konkrétnych cieľov.

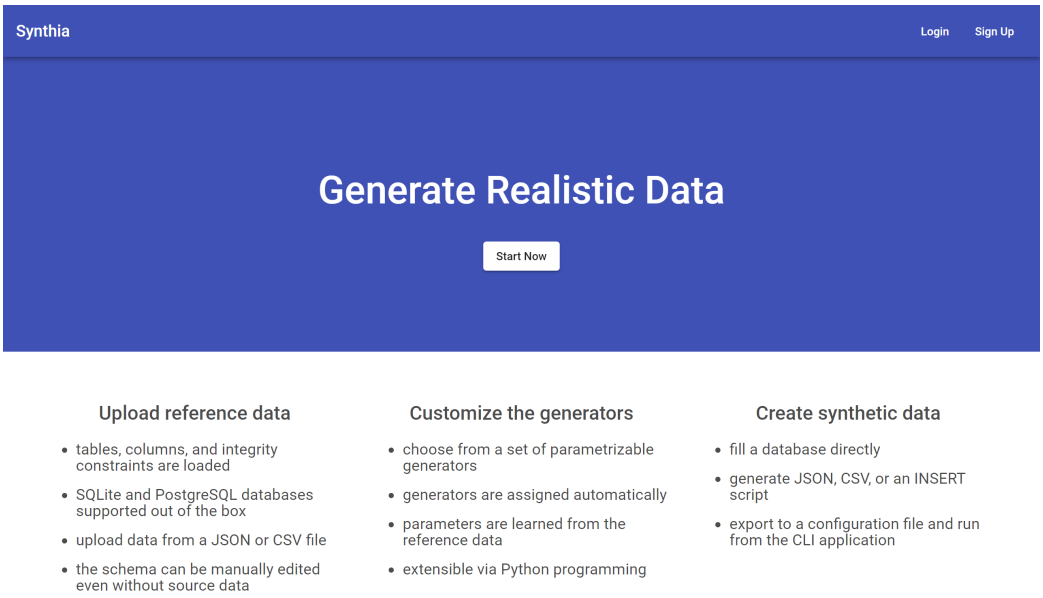
6.1 Grafické prostredie

Budeme potrebovať spustenú webovú aplikáciu. Odporúčame využiť jednu z alternatív popísaných v častiach A.1 a A.2.

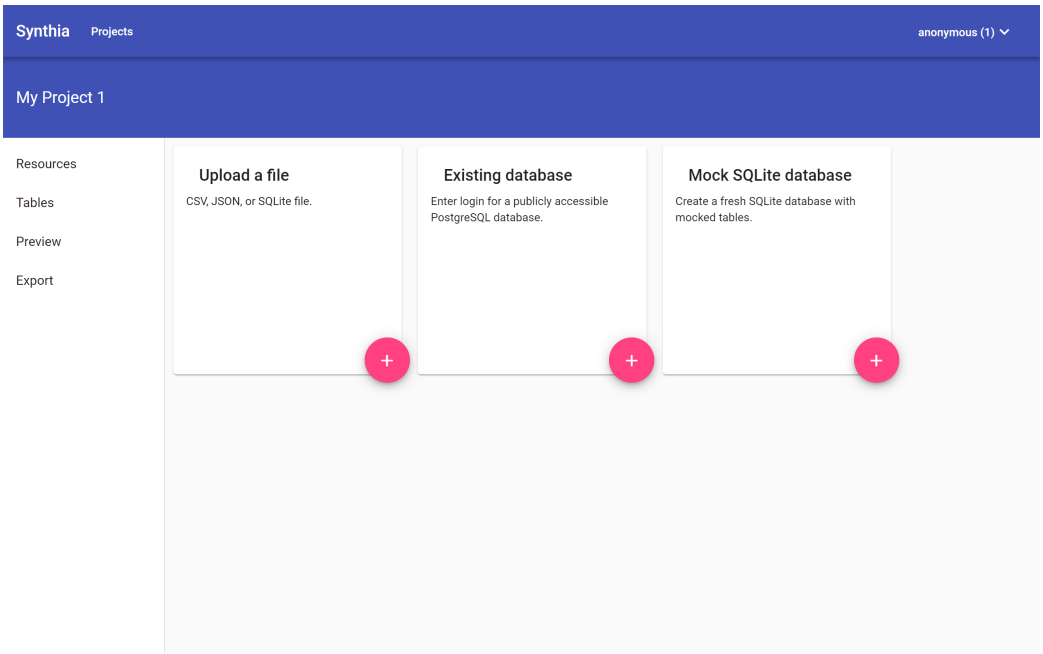
Z úvodnej stránky zobrazenej na obrázku 6.1 by sme sa mohli prihlásiť alebo registrovať tlačidlami vpravo hore. Vyplňovanie formulárov preskočíme potlačením veľkého tlačidla „Start Now“ v strede obrazovky. Týmto vytvoríme anonymný účet a zároveň jeden projekt. Do anonymného účtu sa nedá prihlásiť z iného prehliadača, slúži iba na rýchle odskúšanie systému.

Po vytvorení anonymného účtu a projektu sa ocitáme v pracovnom prostredí nášho projektu, ako to je vidieť na obrázku 6.2. Veľký nápis „My Project 1“ vľavo hore je názov projektu, s ktorým pracujeme. V ľavom paneli vidíme položky menu vzťahujúce sa k projektu. Stránka vykreslená napravo od menu ukazuje prehľad dátových zdrojov. Každý dátový zdroj je reprezentovaný štvorcovou položkou. Tri položky, ktoré vidíme, ešte nie sú skutočné dátové zdroje, iba nám ich umožňujú pridať. Stlačením veľkého ružového tlačidla „+“ vzťahujúceho sa k položke „Mock SQLite database“ pridáme dátový zdroj vo forme databázového súboru. Táto databáza obsahuje niekoľko predpripravených tabuliek.

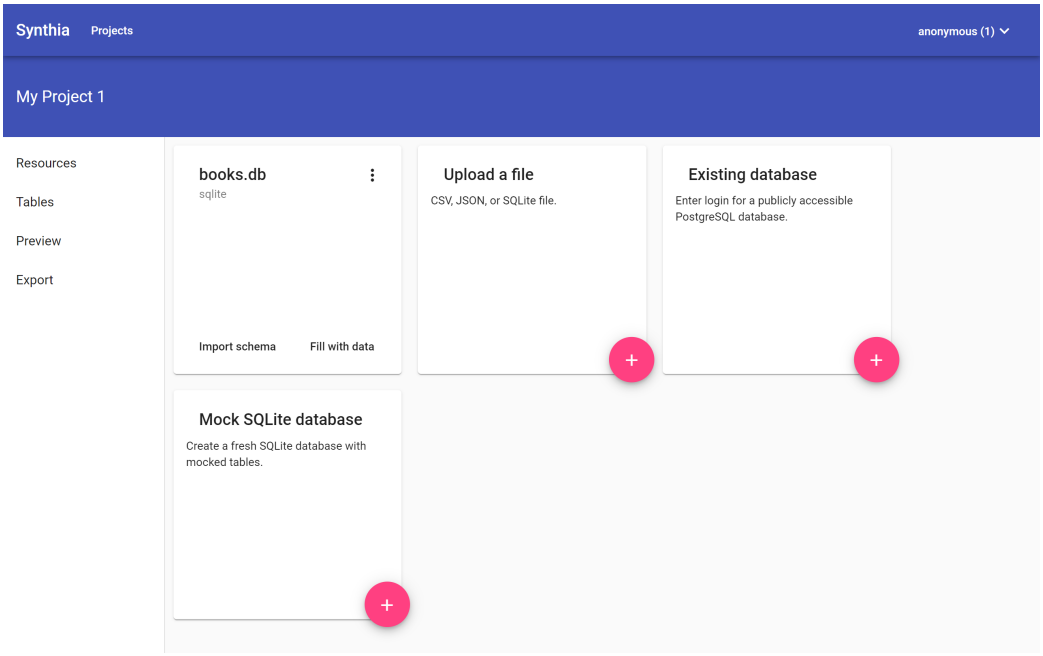
Pridaná databáza sa zobrazí na stránke dátových zdrojov, ako to vidíme na obrázku 6.3. Podobne by sme mohli ružovými tlačidlami „+“ nahrať súbory s tabulárnymi dátami (vo formáte určenom v časti 2.2.2) alebo prístupové údaje k databáze. Systém nám umožňuje z každého dátové zdroja nahrať schému tlačidlom



Obr. 6.1 Úvodná stránka aplikácie.



Obr. 6.2 Stránka s dátovými zdrojmi.



Obr. 6.3 Stránka s dátovými zdrojmi po pridaní databázy.

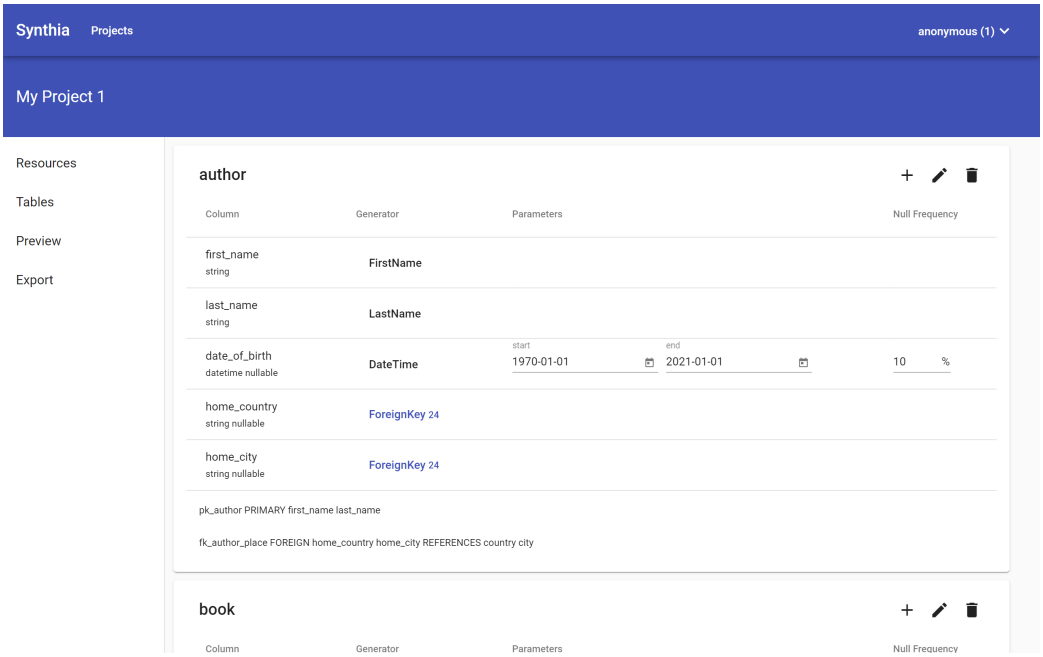
„Import schema“. Ďalšie možnosti dátových zdrojov sa ukrývajú v kontextovom menu označeným tromi zvislými bodkami. Máme aj možnosť stiahnuť nahraný súbor vrátane databázového súboru. To by mohlo byť užitočné, keby sme doň vložili vygenerované dáta.

Je povolené importovať schémy z viacerých zdrojov rôznych typov. Tabuľky sú v projekte unikátne identifikované svojím názvom. Ak nahráme schémy z dvoch zdrojov, ktorých názvy tabuliek sú v konflikte, uvidíme iba tabuľku zo zdroja, ktorý sa nahral ako posledný. Ak by bolo žiadúce generovať dáta pre rôzne tabuľky s rovnakým menom, je to príležitosť rozdeliť prácu do viacerých projektov.

Nahrávať schémy zo zdroja môžeme aj viackrát. To je užitočné, ak sa v databáze zmenila schéma. Mohla pribudnúť alebo ubudnúť nová tabuľka, stĺpec či integritné obmedzenie. Pri každom nahratí však nové tabuľky prepíšu staré, vrátane všetkých nastavení vzťahujúcich sa k tabuľke.

Po kliknutí na položku „Tables“ v ľavom menu sa zobrazí schéma projektu. Ukazujeme to na obrázku 6.4. Vidíme zoznam tabuliek a pri každej tabuľke je zoznam jej stĺpcov a integritných obmedzení. Ku každej tabuľke sa vzťahujú tri ikonky. Ikonou „+“ pridáme k tabuľke nový stĺpec, ikonou ceruzky môžeme zmeniť názov tabuľky a ďalšou ikonou by sme tabuľku zmazali.

Sústreďme sa teraz na zoznam stĺpcov. Pri každom stĺpci vidíme jeho názov a pod ním dátový typ spolu s indikáciou, či povoľuje hodnoty `null`. Ak sa myšou



Obr. 6.4 Schéma projektu.

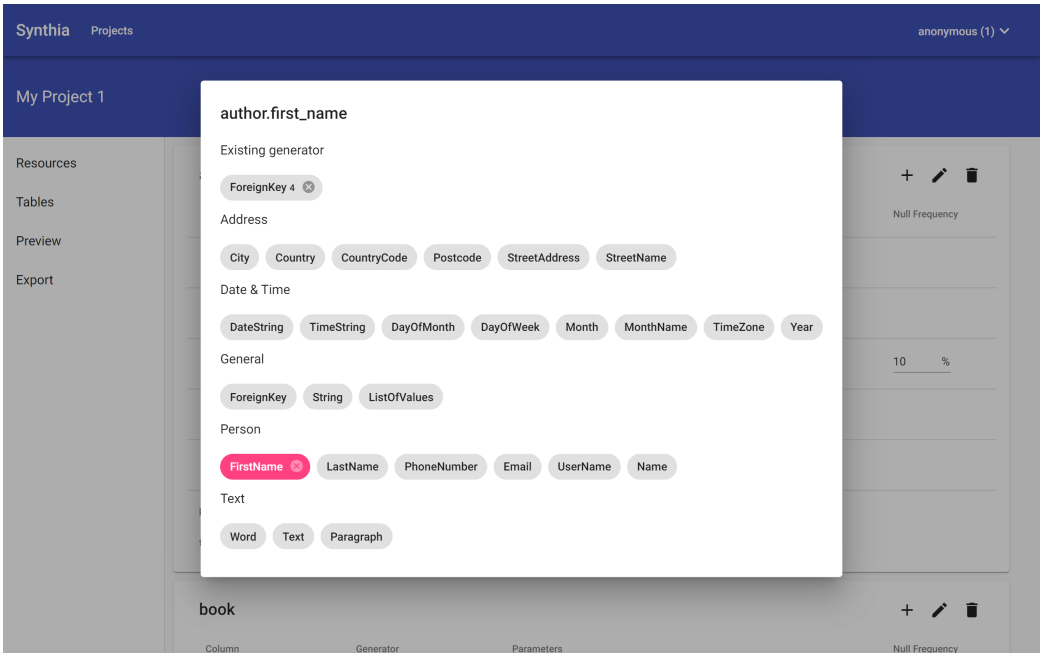
postavíme na názov stĺpca, zobrazí sa ikona „x“, ktorou zmažeme stĺpec z tabuľky. Vedľa nej vidíme ikonu ceruzky, ktorá nám dovoľuje editovať názov a ďalšie charakteristiky stĺpca.

Vpravo od názvu stĺpca je názov generátora. Kliknutím na tento názov sa otvorí dialóg, kde môžeme vybrať iný generátor. Tento dialóg vidíme na obrázku 6.5. Ponuka generátorov sa líši pre jednotlivé stĺpce podľa dátového typu. Keďže väzba generátorov a stĺpcov je many-to-many, vidíme tu dva typy možností. Hneď na začiatku je zoznam existujúcich multistĺpcových generátorov. Kliknutím na jednu z týchto možností priradíme stĺpec k existujúcemu generátoru. Rozlíšiť, ku ktorým stĺpcom už daný generátor patrí, dokážeme jedine podľa malého čísla vedľa názvu generátora (ID). Generátory z ďalších kategórií už vytvárajú novú inštanciu generátora pre daný stĺpec.

Na obrázku 6.5 vidíme mnoho generátorov pre reťazce a niekoľko všeobecných generátorov vhodných pre viacero dátových typov. Niektoré z nich sú špecifické pre jednu doménu. Príkladom je generátor emailov (`Email`) alebo generátor názvov ulíc (`StreetName`). Pri pohybe myši nad výberom generátora sa pri niektorých z nich zobrazí krátke zhrnutie jeho funkcie.

V tomto texte stručne opíšeme iba základnú sadu generátorov. Pozostáva z nasledujúcej ponuky:

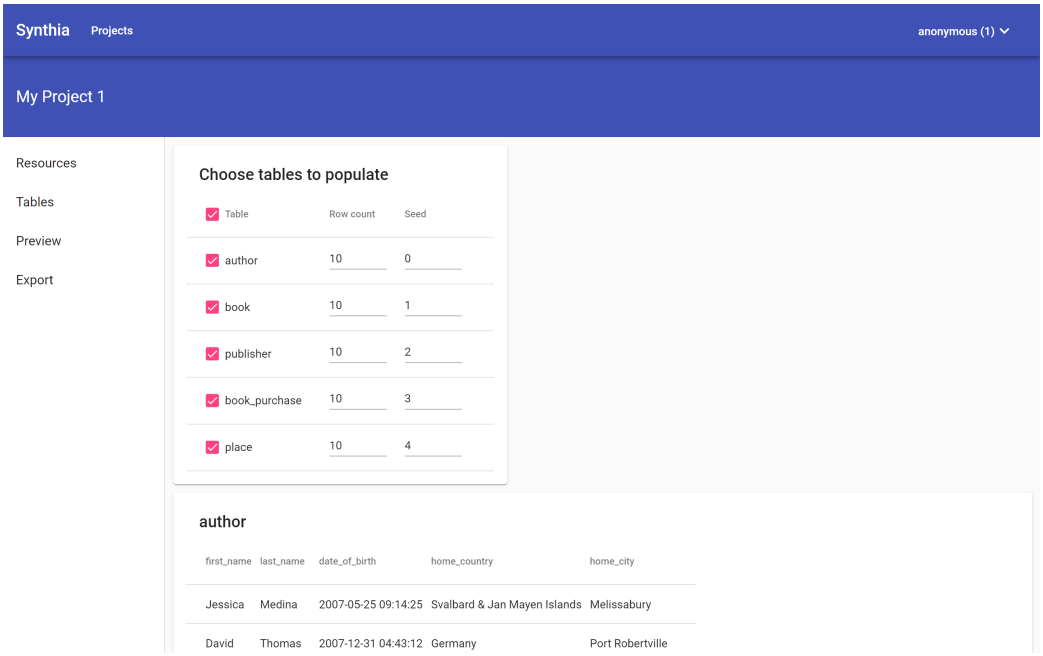
- `String` — generátor najprv zvolí dĺžku reťazca rovnomerne zo zadaného



Obr. 6.5 Výber generátora.

rozsahu a následne z písmen anglickej abecedy vytvorí náhodný reťazec danej dĺžky,

- `Integer` — výstupom je celé číslo rovnomerne vybrané zo zadaného rozsahu,
- `Float` — uniformný generátor čísel zo zadaného intervalu,
- `Gaussian` — generátor čísel z normálneho rozdelenia podľa zadanej strednej hodnoty a smerodajnej odchýlky,
- `DateTime` — rovnomerne vyberá čas a dátum zo zvoleného intervalu,
- `Bernoulli` — vyberá pravdivostné hodnoty so zadanou pravdepodobnosťou pravdy,
- `ForeignKey` — generátor cudzích kľúčov vyberajúci hodnoty z referencovanej tabuľky,
- `PrimaryKey` — generátor primárnych (surrogate) kľúčov,
- `ListOfValues` — generátor uniformne vyberajúci z čiarkou oddelených hodnôt (prípadne iným oddeľovačom).



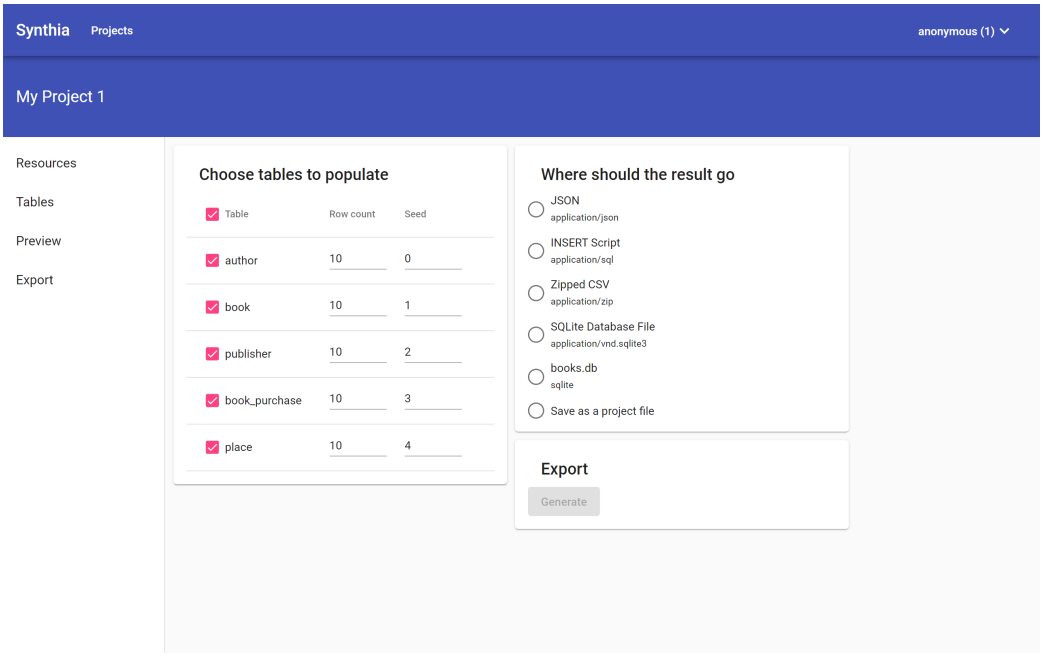
Obr. 6.6 Náhľad generovaných dát.

Vráťme sa ešte k zoznamu stĺpcov v tabuľke. Priradené multistĺpcové generátory sa odlišujú od jednostĺpcových modrou farbou a výpisom ich identifikátora. Napravo od generátora sa zobrazujú polia, kam zadávame ich parametre. Pri každom priradení stĺpca ku generátoru sú parametre odhadnuté zo zdrojových dát (ak nejaké existujú). Parametre generátora môžeme tiež manuálne prispôsobovať. V takom prípade sa automaticky po chvíli ukladajú do databázy. Pri viacerých stĺpcoch vidíme aj parameter „Null Frequency“. Ten určuje podiel `null` hodnôt v generovaných dátach.

Na stránke „Preview“ z obrázka 6.6 vidíme náhľad generovaných dát. V hornej časti stránky vidíme komponent umožňujúci meniť voľby tabuliek, počty záznamov v tabuľkách a seed. Seed je číslo, ktorým sa inicializuje stav náhodných generátorov. Generovanie s rovnakými seedmi vždy vygeneruje rovnaké záznamy. Seed môže zostať aj nevyplnený, v takom prípade dostaneme pri každom behu potenciálne iné hodnoty. Po zmene niektorého z nastavení v tomto komponente sa načíta náhľad generovaných záznamov podľa aktuálnych nastavení.

Z každej tabuľky by malo byť vo východiskovom nastavení vygenerovaných 10 záznamov. Môže sa stať, že v niektorej tabuľke ich bude menej. To môže byť spôsobené zlou konfiguráciou generátorov — vygenerované riadky nespĺňaju integritné obmedzenia a preto sú zamietnuté.

Kliknime na poslednú položku v menu — „Export“. Vidíme stránku ako na



Obr. 6.7 Export.

obrázku 6.7. Na ľavej strane vyberieme tabuľky do výstupnej sady. Ide o rovnaký komponent s rovnakou interpretáciou hodnôt, ako sme videli v náhľade. Pri každej tabuľke zvolíme počet želaných riadkov a prípadne nastavíme seed.

V pravom paneli zvolíme výstupný formát. Možnosti výstupu pochádzajú z troch zdrojov. Ako prvé sa objavujú výstupné súborové ovládače definované v jadre (JSON, INSERT Script, Zipped CSV, SQLite). Všetky štyri fungujú princípom generovania dát do súboru, ktorý si stiahneme z webové prostredia. Ako ďalšie možnosti sa ponúkajú zdrojové databázy. V našom prípade je tam iba jediná — „books.db“. Po zvolení tejto možnosti sa spúšťa priame napĺňanie databázy. Posledná možnosť je uložiť nastavenia do projektového súboru. V tomto prípade sa nič negeneruje. Spustí sa sťahovanie so schémou projektu obsahujúcou tabuľky, nastavenia generátorov a nastavenia z ľavého panelu.

6.2 Príkazový riadok

V tejto sekcii budeme potrebovať prostredie pre spustenie CLI aplikácie. Na inštaláciu odporúčame využiť postup z prílohy A.3.1.

Na stránke „Export“ zobrazenej na obrázku 6.7 zvoľme možnosť „Save as a project file“ a stlačme „Generate“. Vygenerovali sme a stiahli konfiguráciu projektu a exportných nastavení. V nasledujúcom texte budeme predpokladať, že

Výpis kódu 33 Použitie príkazového riadku. Využívame rovnakú syntax, ako sme zvolili v analýze (vo výpise 8).

```
1 ./gen.py -h
2 ./gen.py json -h
3 ./gen.py script ~/Downloads/My_Project_1_proj.json
4 ./gen.py zip ~/Downloads/My_Project_1_proj.json out.zip
5 ./gen.py insert ~/Downloads/My_Project_1_proj.json sqlite://
6 ./gen.py sqlite ~/Downloads/My_Project_1_proj.json output.db
```

cesta k stiahnutému súboru je `~/Downloads/My_Project_1_proj.json`. Cestu k tomuto súboru predáme ako parameter nášmu CLI programu.

Spustiteľný script `gen.py` sa v našom softvérom riešení nachádza v adresári `backend`. Nasledujúcim príkazom spustíme generovanie dát vo formáte JSON. Vypisované budú na štandardný výstup.

```
./gen.py json ~/Downloads/My_Project_1_proj.json
```

Prvý argument (`json`) definuje formát výstupu a druhý cestu ku konfiguračnému súboru. Ak chceme vypisovať do súboru, môžeme pridať cestu k výstupnému súboru ako tretí argument. Tento súbor bude prepísaný alebo vytvorený.

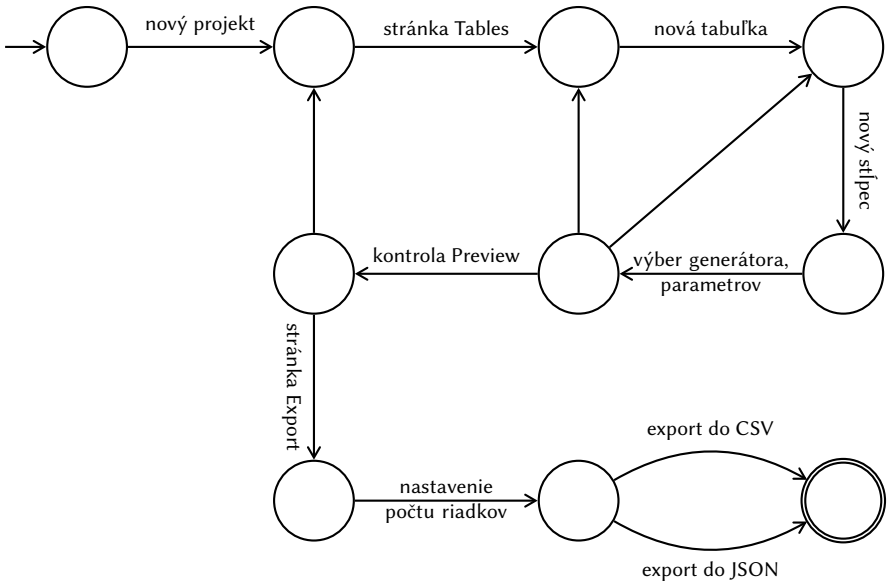
```
./gen.py json ~/Downloads/My_Project_1_proj.json out.json
```

Ďalšie možné spôsoby použitia CLI uvádzame vo výpise 33. Prepínačom `-h` alebo `--help` zobrazíme možné akcie (na riadku 1) alebo syntax danej akcie (na riadku 2).

Príkaz na treťom riadku vygeneruje sadu príkazov `INSERT` a vypíše ich na štandardný výstup. Akcia `zip` na štvrtom riadku generuje zazipované CSV súbory. V tomto prípade musíme uviesť aj výstupný súbor, pretože výpis binárnych dát na štandardný výstup by bol nezmyselný.

Predposledný príkaz vkladá záznamy priamo do databázy. Prístup k reálnej databáze predáme pomocou URL[1] v treťom pozičnom argumente. Príkaz na piatom riadku nebude celkom fungovať. Snaží sa zapísať dáta do novej SQLite databázy v pamäti, ktorá nemá vytvorené tabuľky. Reálna URL nesúca prístupové údaje by mala tvar `postgresql://user:password@localhost/database`. Ak chceme vkladať dáta do databázy SQLite, môžeme využiť príkaz na poslednom riadku. Ten sa postará o vytvorenie SQLite databázy so schémou projektu v súbore `output.db`.

[1] `https://docs.sqlalchemy.org/en/14/core/engines.html`



Obr. 6.8 Stavový automat popisujúci scenár manuálneho zadávania schémy.

6.3 Scenáre použitia

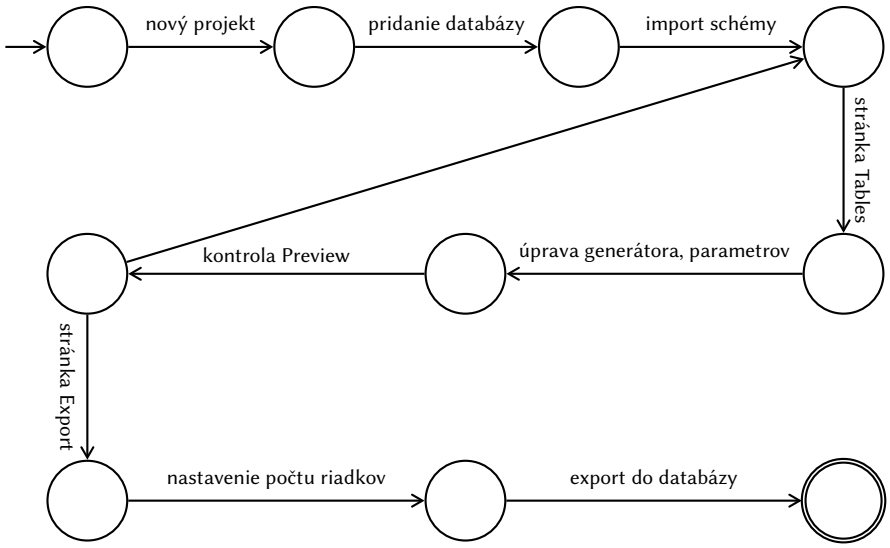
V tejto sekcii popíšeme tri možné scenáre, ako koncový užívateľ bude využívať aplikáciu. Líšia sa medzi sebou tým, či užívateľ má dostupnú schému dát, a cieľom výstupu. Nejde však o jediné možné spôsoby využitia.

6.3.1 Manuálna tvorba schémy

Ak nie sú dostupné referenčné dáta ani schéma databázy, užívateľ môže vytvoriť schému v grafickom prostredí. Scenár je zobrazený na obrázku 6.8. Tento scenár je podobný spôsobu funkcie webových riešení z časti 2.1.1.

Začneme vytvorením nového projektu. V ňom sa presunieme sa na stránku „Tables“. Veľkým ružovým tlačidlom „+“ vytvoríme tabuľku. Do tabuľky pridáme stĺpec a nastavíme mu generátor, prípadne parametre. Môžeme vytvárať ľubovoľne veľa stĺpcov a tabuliek. Medzitým je možné kontrolovať podobu generovaných záznamov na stránke „Preview“.

Nakoniec prejdeme na stránku „Export“. Zvolíme počty želaných záznamov pre každú tabuľku. Vyberieme typ výstupného súboru — CSV, JSON alebo SQLite. Stlačíme tlačidlo „Generate“. Ak všetko prebehne v poriadku, spustí sa sťahovanie.



Obr. 6.9 Stavový automat popisujúci scenár manuálneho zadávania schémy.

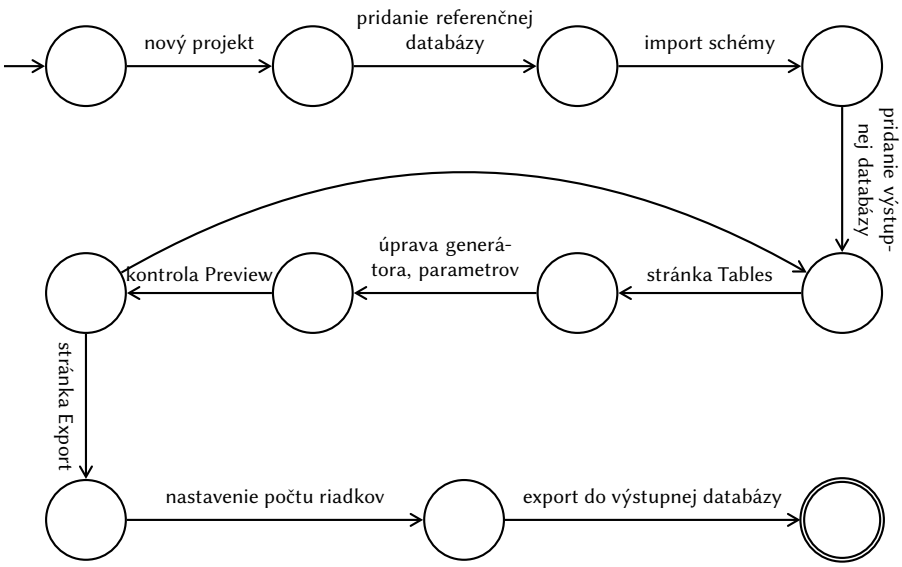
6.3.2 Zväčšenie objemu dát v databáze

Tento scenár popisuje situáciu, keď užívateľ má dostupnú databázu a chce v nej zväčšiť množstvo záznamov. Zahŕňa to aj situáciu, ak je táto databáza prázdna. Možný priebeh je popísaný obrázkom 6.9. Ide o podobný spôsob práce, aký je typický pre natívne riešenia popísané v časti 2.1.1.

Na začiatku vytvoríme nový projekt. Na úvodnej stránke projektu pridáme prístupy k databáze. Táto databáza musí byť dostupná zo serveru, na ktorom je spustená aplikácia. To znamená, že nie je možné využiť online demo, ak naša databáza nie je verejne prístupná. Po pridaní databázy klikneme na tlačidlo „Import schema“.

Presuňme sa na stránku „Tables“. Vidíme zoznam importovaných tabuliek a stĺpcov. Generátory boli automaticky priradené. Ak sa v zdrojovej databáze nachádzali záznamy, parametre generátorov sú z nich odhadnuté. Generátory alebo parametre môžu byť teraz dodatočne menené. Podobu generovaných záznamov skontrolujeme na stránke „Preview“.

Po nastavení generátorov prejdeme na stránku „Export“. Zvolíme počty želaných záznamov pre každú tabuľku. Vo výbere výstupného cieľa vyberieme našu databázu. Stlačíme tlačidlo „Generate“. Po dokončení generovania sa zobrazí správa informujúca o výsledku operácie.



Obr. 6.10 Stavový automat popisujúci scenár generovania dát podľa vzoru.

6.3.3 Generovanie dát podľa vzoru

V tejto časti popíšeme scenár, keď máme dostupnú naplnenú (referenčnú) databázu. Cieľom je naplniť inú (výstupnú) databázu s rovnakou schémou podobnými dátami. Postup ilustrujeme diagramom 6.10 Ide o prirodzenú situáciu pri vývoji aplikácii, keď je dostupná produkčná databáza, a chceme vytvoriť testovaciu databázu. Existujúce riešenia však túto situáciu robia komplikovanú, pretože v každom projekte pracujú iba s jedinou databázou.

Rovnako ako v predchádzajúcich scenároch začneme vytvorením nového projektu. Pridáme prístupy najprv k referenčnej databáze. Klikneme na tlačidlo „Import schema“. Potom pridáme prístup k výstupnej databáze, ale neimportujeme.

Podobne ako v časti 6.3.2 sa teraz môžeme presunúť na stránku „Tables“. Prispôsobujeme generátory a ich parametre. Kontrolujeme náhľad záznamov na stránke „Preview“.

Na stránke „Export“ volíme počty želaných záznamov. Ako cieľ výstupu volíme v poradí druhú pridanú (výstupnú) databázu. Prácu dokončíme stlačením tlačidla „Generate“.

Alternatívne by sme miesto priameho vkladania mohli vygenerovať konfiguračný súbor. Generovanie s vkladaním do výstupnej databázy by sa spúšťalo prostredníctvom CLI, ako sme to popisovali v časti 6.2. Výstupnú databázu by nebolo potrebné pridávať ako dátový zdroj. Výhoda tejto alternatívy je, že môžeme proces generovania integrovať do automatizovaného testovania.

Kapitola 7

Možnosti vylepšenia

V tejto kapitole poukážeme na oblasti systému, ktoré by mohli byť zlepšené. Predstavíme konkrétne vylepšenia a rozšírenia, vďaka ktorým by sa umocnili výhody našej aplikácie voči existujúcim.

Nasadenie do produkcie

Aby bol systém nasaditeľný do produkcie ako webová služba, zostáva vyriešiť ešte niekoľko nedostatkov. Užívateľské prostredie by mohlo byť sprehľadnené. Nezaoberali sme sa profilovaním a vylaďovaním procedúr generovania či importu. Nemáme nastavené limity na veľkosť nahrávaných súborov, veľkosť pripojených databáz a všeobecne množstvo požiadaviek, ktorý je náš webový server ochotný spracovať. Je teda priamočiare zahltiť server — nemusel by zvládnuť importovať schému a odhadnúť parametre generátorov z veľkej databázy, najmä ak by dostal viac takých požiadaviek súčasne.

Editácia integritných obmedzení

Bolo by užitočné pridať do grafického prostredia možnosť manuálne pridávať integritné obmedzenia. Súčasťou toho by malo byť aj odobratie alebo úprava už existujúcich integritných obmedzení. Nezáleží pritom, či obmedzené a referencované stĺpce boli vytvorené z dátového zdroja alebo manuálne.

Takmer celá vnútorná architektúra je pre túto vlastnosť pripravená. Stačí k nej dorobiť endpointy a grafické komponenty. Malo by teda ísť o priamočiare rozšírenie.

Ďalšie štatistické rozdelenia

Vynechali sme implementáciu Poissonovho pravdepodobnostného rozdelenia a exponencionálneho rozdelenia. Po vzore implementovaného Gaussovho rozde-

lenia by tieto implementácie mali zahŕňať odhad parametrov zo vstupných dát a generovanie. Pri tomto rozšírení by sme sa mali stretnúť iba s problémami, ktoré priamo súvisia s danými rozdeleniami. Súčasný architektonický návrh by nemal vytvárať žiadne technické prekážky.

Generátory rešpektujúce viacrozmerné rozdelenia

Pomocou zavedených nástrojov je možné implementovať generátory rešpektujúce viacrozmerné rozdelenia. Podobne ako sme implementovali jednorozmerné Gaussovo rozdelenie by sme mohli implementovať viacrozmerné Gaussovo rozdelenie. Využili by sa pritom existujúce rozhrania pre multistĺpcové generátory, ktoré už používa generátor (kompozitných) cudzích kľúčov. Ďalej by sme to mohli zovšeobecniť na zmes viacrozmerných Gaussových rozdelení. Tú by sme mohli odhadovať zo vstupných dát pomocou EM algoritmu.

Načítavanie schém

Mohli by sme pridať možnosť načítania schém zo súborov popisujúcich schémy CSV súborov. Jeden takýto formát sme spomenuli v časti 2.2.2. Rýchle a jednoduché riešenie by bolo považovať tieto schémy za zdroj dát, ktorý by nám síce dával schému ale žiadne záznamy. Rovnako by sme implementovali ďalšie schémy pre iné formáty.

Viac nastavení exportu

Na stránke „Export“ definovanej v časti 3.1.5 by sme mohli ponúkať viac možností prispôsobenia. Pri výstupe do databázy by bolo možné ponúknuť možnosť vyprázdnenia tabuliek pred začiatkom vkladania. Ďalšia možnosť by mohla byť vytvorenie tabuliek ak neexistujú. Tieto možnosti by sme mohli ponúkať aj pri generovaní `INSERT` scriptu. Tiež by potom bolo vhodné nechať užívateľa zvoliť, pre ktorý SQL dialekt má byť script generovaný.

Integrácia ďalších formátov a databázových systémov

Pre niektorých užívateľov by mohla byť užitočná možnosť nahrať dáta vo formáte XML. Toto rozšírenie by malo byť priamočiare po vzore implementácií formátov CSV a JSON. Tiež sa ponúka príležitosť pridania XML ako výstupného formátu.

Podpora ďalších databázových systémov by mala byť rovnako jednoducho pridateľná. V závislosti od konkrétneho systému by si to vyžadovalo nainštalovanie ovládačov a ich povolenie v kóde. Približný postup sme popísali v časti 5.3.4.

Spolupráca viacerých užívateľov

Miesto toho, aby projekt patril práve jednému užívateľovi, by sme mohli túto väzbu zovšeobecniť na many-to-many. Teda užívatelia by mohli spoločne spolupracovať na jednom projekte. Toto rozšírenie by si vyžadovalo výraznejšie zmeny na frontende. Rozbili by sa naše predpoklady, že zmeny v stave môžu prichádzať len od jediného užívateľa. Manažment stavu na frontende by si vyžadoval aj spoluprácu s backendom, keď by zmeny na projekte robil iný užívateľ ako prihlásený.

Profilovanie

Ďalšou príležitosťou je implementovať profilovanie dát. Mohli by sme napríklad vypočítavať histogramy či počty unikátnych hodnôt. Profilovať je možné nielen zdrojové dáta, ale aj generované. Užívateľ by mohol mať možnosť porovnať vybrané metriky vypočítané z referenčných dát a porovnať ich s tým, čo je generované. Tieto informácie by mohli byť využité aj ďalšími typmi generátorov.

Kapitola 8

Záver

Na záver vyhodnotíme našu prácu v kontexte cieľov stanovených v časti 1.1. Softvérové riešenie porovnáme s analyzovanými riešeniami podľa kritérií stanovených v časti 2.1.

Podarilo sa nám implementovať webovú aplikáciu s REST API a CLI. Webové rozhranie je dostupné z každého zariadenia s moderným webovým prehliadačom. CLI si vyžaduje inštaláciu Pythonu, ktorý je však dostupný na viacerých platformách. Detaily inštalácie sú popísané v prílohe A. Podporujeme databázové systémy PostgreSQL a SQLite. Implementáciu ostatných sme vynechali, ale mala by byť priamočiara. Požiadavky na multiplatformovosť boli splnené s týmto kompromisom.

Zdrojové kódy sme zdokumentovali a zverejnili sme ich pod permisívnou licenciou. Vývojové prostredie je jednoducho spustiteľné pomocou Docker Compose. V časti 5.3.1 sme predstavili deklaratívne rozhranie, pomocou ktorého je možné programovať vlastné typy generátorov. Podobne jednoducho je možné pridávať podporu ďalších dátových zdrojov alebo výstupných formátov. Požiadavky na otvorenosť a rozšíriteľnosť považujeme za splnené.

Podporujeme import dát z databázy alebo zo súborov CSV a JSON. Pripravili sme základnú sadu parametrizovaných generátorov pre rôzne dátové typy. Jednotlivé tabuľky vo výslednej schéme môžu pochádzať z rôznych zdrojov, vrátane manuálnych zmien prostredníctvom GUI. Ak sa schéma v pôvodnom zdroji zmení, novo načítané tabuľky prepíšu staré. Podporujeme viacero možností výstupu, vrátane `INSERT` scriptu, výstupu do databázy a výstupu vo formátoch CSV a JSON. Dávame aj možnosť export konfigurácie a následného generovania prostredníctvom CLI. Splnili sme teda cieľ týkajúci sa všeobecnosti aplikácie.

Ak má užívateľ dostupnú databázu s hotovou schémou či súbor s referenčnými dátami, môže ich nahrať do systému. Sme schopní z nich prečítať tabuľky, stĺpce, ich dátové typy a integritné obmedzenia. Stĺpcom prideľujeme vhodné generátory na základe rozšíriteľného algoritmu, ktorý sme popísali v časti 4.2.5.

Názov	generatedata	Mockaroo	Red Gate	Datanamic	**Synthia**
Licencia	GPL 3	komerčná	komerčná	komerčná	**MIT**
Open source	áno	nie	nie	nie	**áno**
Platforma	web	web	Windows	Windows	**web**
Podpora DBMS	`INSERT`	`INSERT`	SQL Server	mnohé	**mnohé**
Iné výstupy	mnohé	mnohé	nie	`INSERT`	**áno**
API	REST	REST	CLI	CLI	**REST, CLI**
Schéma z DB	nie	nie	áno	áno	**áno**
Programovanie	nie	Ruby	mnohé	SQL	**nie**
Cudzie kľúče	nie	nie	áno	áno	**áno**

Tabuľka 8.1 Vlastnosti generátorov v skratke. Tabuľka 2.1 rozšírená o naše riešenie (zvýraznené).

Parametre generátorov sú konfigurované podľa referenčných dát. Systém zabezpečuje integritu vygenerovaných dát až na obmedzenia `CHECK`. Týmto považujeme cieľ automatizácie generovania dát za splnený.

Celkovo sa nám podarilo splniť hlavné ciele práce. Výhody voči existujúcim riešeniam spočívajú najmä v tom, že sme celú funkcionalitu sprístupnili ako webovú službu. Navyše sme ponúkli zovšeobecňujúci pohľad na generovanie dát. Nelimitujeme sa iba na databázy, ako to robia natívne riešenia a v porovnaní s webovými riešeniami sme pridali spracovanie zdrojových dát. Programovanie nových typov generátorov síce nemáme integrované do grafického prostredia, ponúkame však všestranné rozhranie pre pridávanie generátorov deklaratívnym spôsobom. Vytvorili sme rozšíriteľnú platformu, ktorá prirodzene pracuje aj s multistĺpcovými generátormi. Vďaka tomu sa nám podarilo vyriešiť problém cirkulárnych závislostí, s ktorým si ostatné riešenia nevedeli poradiť. Porovnanie nášho riešenia s existujúcimi sme zhrnuli v tabuľke 8.1.

Zoznam použitej literatúry

[1] Carlos Coronel a Steven Morris. *Database Systems: Design, Implementation, and Management*. 13th. Cengage Learning, 2018. ISBN: 978-1-337-62790-0.

[2] Hector Garcia-Molina, Jeffrey D. Ullman a Jennifer Widom. *Database Systems: The Complete Book*. 2nd. Pearson, 2008. ISBN: 978-0131873254.

[3] Y. Shafranovich. *Common Format and MIME Type for Comma-Separated Values (CSV) Files*. 2005. URL: https://tools.ietf.org/html/rfc4180.

[4] Jeni Tennison, ed. *CSV on the Web: A Primer*. 2016. URL: https://www.w3.org/TR/tabular-data-primer/.

[5] *The JSON data interchange syntax*. 2nd. Ecma International. 2017. URL: https://www.ecma-international.org/publications-and-standards/standards/ecma-404/.

[6] Michael Bayer. "SQLAlchemy". In: *The Architecture of Open Source Applications Volume II: Structure, Scale, and a Few More Fearless Hacks*. Ed. Amy Brown a Greg Wilson. aosabook.org, 2012. URL: http://aosabook.org/en/sqlalchemy.html.

[7] Martin Mareš a Tomáš Valla. *Průvodce labyrintem algoritmů*. CZ.NIC, 2017. ISBN: 978-80-88168-22-5.

Príloha A

Spustenie

Aplikácia pozostáva z viacerých častí — webový server, databáza a frontend. Každá z nich vyžaduje vlastné závislosti a časti musia navzájom komunikovať. Z toho dôvodu sa spustenie komplikuje. Podľa individuálneho spôsobu použitia ponúkame niekoľko možností. Zdrojové kódy sú dostupné vo verejnom repozitári na adrese `https://github.com/pecimuth/synthia`.

A.1 Online demo

Na rýchle odskúšanie je ideálne online demo. Dostupné je z webovej adresy `https://synthia-generator.web.app/`. Nasadili sme sem aplikáciu s kompletnou funkcionalitou. Užívateľské dáta sú však pravidelne premazávané.

A.2 Vývojové prostredie webovej aplikácie

Na účely vývoja je ideálne využiť spustenie pomocou systému Docker Compose.[1] Z koreňového adresára obsahujúceho súbor `docker-compose.yml` spustíme služby príkazom:

```
docker-compose up
```

Prebehne stiahnutie a zostavenie potrebných prostredí, môže to trvať dlhšiu dobu. Tento príkaz zaberie jeden terminál, takže ďalšie príkazy je potrebné spúšťať z iného terminálu.

Ako internú databázu využívame PostgreSQL. Pri prvom spustení alebo pri zmene schémy databázy je potrebné spraviť migráciu. Pripravili sme príkaz, ktorý zmaže celú existujúcu schému a vytvorí ju nanovo (čím sa aj stratia všetky

[1] `https://docs.docker.com/compose/`

dáta). Nasledujúci príkaz musí byť spustený z koreňového adresára po nábehu databázy spustenej príkazom vyššie.

```
docker-compose exec backend flask recreate-database
```

Kód vygenerovaný zo schémy API sa nenachádza v repozitároch. Rovnako je potrebné vygenerovať ho znova pri zmene schémy API. Dosiahneme to nasledujúcim príkazom:

```
docker-compose exec frontend npm run api
```

Z adresy `http://localhost:4200/` je dostupné užívateľské webové prostredie. Na adrese `http://localhost:5000/apidocs/` sa nachádza interaktívna API dokumentácia.

A.3 Natívne prostredia

V tejto sekcii popíšeme ako nainštalovať a spustiť izolované prostredia jednotlivých častí.

A.3.1 Backend

Natívne prostredie je vhodné na spúšťanie generátora z príkazového riadku alebo testov.

Presuňme sa do podadresára `backend`. Potrebujeme Python verzie aspoň 3.8. Vytvoríme virtuálne prostredie, aktivujeme ho a nainštalujeme doň závislosti.

```
cd backend
python3 -m venv env
source ./env/bin/activate
python -m pip install -r requirements.txt
```

Teraz môžeme používať CLI aplikáciu.

```
./gen.py -h
```

Pre spustenie testov najprv nainštalujeme knižnicu pytest a ďalším príkazom ich spustíme.

```
python -m pip install pytest
python -m pytest
```

A.3.2 Frontend

K vývoju sme používali node[2] verzie 12. Závislosti nainštalujeme pomocou npm z podadresára `frontend`.

```
cd frontend
npm install
```

Testy spustíme príkazom:

```
npm run test
```

[2] `https://nodejs.org/en/`