Masarykova univerzita
Fakulta informatiky



Grafické prostredie pre správu výpočtových workflow analytickej platformy

Bakalárska práca

Erik Hricišák

Brno, jar 2021

Masarykova univerzita
Fakulta informatiky



Grafické prostredie pre správu výpočtových workflow analytickej platformy

Bakalárska práca

Erik Hricišák

Brno, jar 2021

Na tomto mieste sa v tlačenej práci nachádza oficiálne podpísané zadanie práce a vyhlásenie autora školského diela.

Vyhlásenie

Vyhlasujem, že táto bakalárska práca je mojím pôvodným autorským dielom, ktoré som vypracoval samostatne. Všetky zdroje, pramene a literatúru, ktoré som pri vypracovaní používal alebo z nich čerpal, v práci riadne citujem s uvedením úplného odkazu na príslušný zdroj.

Erik Hricišák

Vedúci práce: RNDr. Tomáš Rebok, Ph.D.

Poďakovanie

Rád by som sa poďakoval vedúcemu tejto bakalárskej práce RNDr. Tomášovi Rebokovi, Ph.D za ochotu, pomoc a usmernenie pri písaní práce. Rovnako vďaka patrí aj Bc. Denisovi Kasaničovi, ktorý mi vždy ochotne poradil a pomohol s pochopením požiadaviek, projektu a problémami, ktoré počas tvorby práce vznikli.

Zhrnutie

Cieľom tejto bakalárskej práce je návrh a implementácia grafického rozhrania pre definíciu a správu analytických workflows výskumného projektu ANALYZA.

V prvej časti práce je bližšie opísaný výskumný projekt, pre ktorý bolo grafické rozhranie pripravené a existujúce nástroje na správu workflows. V druhej časti práce sú zanalyzované požiadavky kladené na výsledné grafické prostredie, ako aj popísaný jeho návrh a implementácia spolu s prezentáciou výsledkov práce.

Kľúčové slová

workflow, DAG, GUI, JavaFX, Hibernate, Docker kontajner

Obsah

1 Úvod **1**

2 Projekt ANALYZA **3**
2.1 Dátový sklad 4
2.2 Výpočtový subsystém 4
2.3 Vizualizačný komponent 5

3 Nástroje na správu workflows **7**
3.1 Apache Airflow 8
3.2 Argo Workflows 9
3.3 Prefect 10
3.4 Iné nástroje 11

4 Analýza požiadaviek **13**
4.1 Funkčné požiadavky 14
4.2 Nefunkčné požiadavky 15

5 Návrh **17**
5.1 Sekcia 1: Obrazy 17
5.2 Sekcia 2: Moduly 18
5.3 Sekcia 3: Operácie 19
5.4 Grafický návrh 20

6 Implementácia **23**
6.1 JavaFX 23
6.2 Architektúra 23
6.2.1 MVC 23
6.2.2 Trojvrstvová architektúra 26
6.3 Obrazy modulov 27
6.4 Moduly 29
6.5 Operácie 31
6.5.1 Výber vhodného nástroja 31
6.6 Lokalizácia 33
6.7 Logovanie chýb 33
6.8 Nastavenia aplikácie cez konfiguračný súbor 34
6.9 Dizajn 34

7 Testovanie **35**

8 Záver **37**

A Elektronické prílohy **39**

Bibliografia **41**

Zoznam obrázkov

2.1 *Schéma zobrazujúca architektúru navrhnutého systému pre ukladanie a analýzu heterogénnych dát* [1] 3
2.2 *Vzťahová analýza tvoriaca základný náhľad na dáta uložené v dátovom sklade* [2] 6
3.1 *Workflow, sieť úloh, na ktorých je definované parciálne usporiadanie* 7
3.2 *Obrazovka grafického rozhrania Apache Airflow zobrazujúca prostredie grafového pohľadu na závislosti úloh vo workflow* [10] 8
3.3 *Obrazovka grafického rozhrania Argo Workflows zobrazujúca prostredie grafového pohľadu na závislosti úloh vo workflow* 9
3.4 *Obrazovka grafického rozhrania nástroja Prefect zobrazujúca prostredie grafového pohľadu na závislosti úloh vo workflow* [12] 10
4.1 *História projektov pre správu workflow vzhľadom k ich popularite na platforme Github vyvýjajúcej sa v čase, vyjadrenej v počte hviezd* 13
4.2 *Diagram prípadov použitia zobrazujúci funkčné požiadavky kladené na výsledný softvér* 16
5.1 *Návrh rozloženia obrazovky pre manipuláciu s operáciami* 20
6.1 *ERD diagram zobrazujúci dátový model aplikácie* 25
6.2 *Schéma zobrazujúca architektúru implementovanej aplikácie* 27
6.3 *Grafické rozhranie monitorovania aktuálne zostavujúcich sa obrazov kontajnerov* 29
6.4 *Grafické rozhranie pre prácu s vybraným modulom* 30
6.5 *Grafické rozhranie pre modelovanie operácií* 33

1 Úvod

Systémy na správu workflows naberajú v posledných rokoch na popularite. Čím komplexnejším a náročnejším sa stáva svet okolo nás, tým väčší dôraz je kladený na zjednodušovanie a automatizáciu repetitívnych aktivít. Projekt ANALYZA, ktorý vznikol ako výskumný projekt na Masarykovej univerzite pre Políciu Českej republiky si za cieľ kladie navrhnutie a prototypovú implementáciu systému pre správu a analýzu dát veľkého rozsahu. V rámci projektu vzniklo viacero analytických nástrojov, ktoré je možné skladať a vytvárať tak analytické workflow, ktoré je následne možné opakovane spúšťať na zvolených dátach, a tak zjednodušiť náročný proces analýzy veľkého množstva dát v doméne policajného vyšetrovania. V procese implementácie vznikla požiadavka na administratívny nástroj, ktorý by užívateľom umožnil jednoduché a jednotné prevedenie procesom vytvárania analytických workflows, od definícií jednotlivých čiastkových krokov, až po ich skladanie do komplexných štruktúr, a to užívateľsky príveti-vým spôsobom. Návrh a implementácia takéhoto nástoja je témou tejto bakalárskej práce.

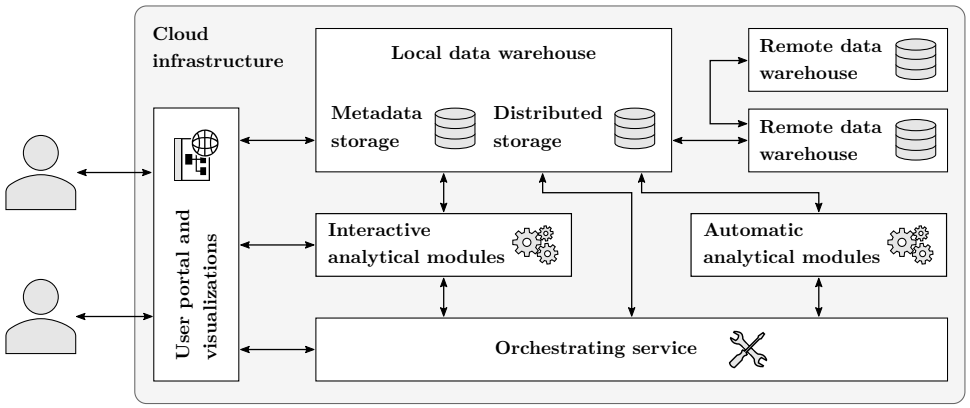
Na trhu je veľké množstvo komerčných možností, ktoré správu workflow riešia, projekt ANALYZA však vzhľadom na svoje úzko špecifikované požiadavky a doménu vyžadoval vlastné riešenie. V druhej kapitole práce preto popíšem projekt ANALYZA s bližším zameraním na jeho výpočtový komponent. V tretej kapitole, v rámci rešeršnej časti práce predstavím vybrané existujúce nástroje na správu workflows.

Druhá časť práce sa zaoberá návrhom požadovaného nástroja a výsledkami bakalárskej práce. V štvrtej kapitole zasadím bakalársku prácu do kontextu projektu ANALYZA a vymedzím funkčné a nefunkčné požiadavky kladené na vyvíjané grafické prostredie, v piatej kapitole následne navrhnem riešenie zadefinovaného problému. V šiestej kapitole čitateľa oboznámim s technológiami a prístupom, ktoré boli za účelom splnenia požiadaviek použité, ako aj s výsledkami práce. V siedmej kapitole sa zmienim o spôsoboch testovania aplikácie. V závere práce zhrniem výsledky a načrtnem ďalšie možné rozšírenia implementácie.

2 Projekt ANALYZA

Vyšetrovanie mnohých protiprávnych aktivít býva značne náročné. Policajní analytici musia často zhromaždiť veľké množstvo dát, ktoré je potrebné spolu porovnávať, analyzovať či hľadať v nich vzory a súvislosti. Náročnou úlohou sa takéto analýzy môžu stať najmä v kontexte rozsiahlych, komplexných prípadov zahrňujúcich prácu s množinou dát veľkého rozsahu. Či už ide o získané obrazy diskov, finančné transakcie alebo sieťové toky, vyšetrovatelia sú postavení pred neľahkú úlohu.

Projekt „Komplexní analýza a vizualizace heterogenních dat velkého rozsahu“ (ďalej len ANALYZA), realizovaný v spolupráci s Políciou Českej republiky, si za úlohu kladie vytvorenie systému umožňujúceho zhromažďovanie, ukladanie a analýzu heterogénnych dát veľkého objemu. Primárnym cieľom tohto výskumného projektu je preskúmanie možností a limitov takýchto medzidoménových analýz [1]. Po rozsiahlych konzultáciách s predpokladanými používateľmi systému z radov policajných analytikov bola navrhnutá a implementovaná architektúra projektu opierajúca sa o tri základné časti – komponent dátového skladu, výpočtového subsystému a vizualizačný komponent, zobrazená na Obrázku 2.1.



Obr. 2.1: Schéma zobrazujúca architektúru navrhnutého systému pre ukladanie a analýzu heterogénnych dát [1]

2.1 Dátový sklad

Dátový sklad je ústredným komponentom celého vyvíjaného systému, ktorého hlavnou úlohou je úschova dát. Skladá sa z troch častí:

- **samotné dátové úložisko** – vzhľadom k požiadavkám škálovateľnosti a robustnosti bolo zvolené distribuované riešenie pre uskladnenie dát, konkrétne Hadoop Distributed File System. Distribuovaný súborový systém je centrom pre uloženie jednak pôvodných dát, ako aj (medzi)výsledkov analýz.
- **relačná databáza** – relačná databáza využívajúca technológiu PostgreSQL. Umožňuje ukladanie štruktúrovaných metadát slúžiacich na identifikáciu, vyhľadávanie a filtrovanie dát uložených v distribuovanom súborovom systéme.
- **proxy komponent** – voliteľný komponent poskytujúci funkcionalitu pre vytváranie hierarchie dátových skladov. Tento komponent umožňuje vytváranie lokálnych dátových skladov, ich synchronizáciu s nadradeným (centrálnym) dátovým skladom a riešenie konfliktov vzniknutých propagáciou dát medzi rôznymi úrovňami dátového úložiska.

2.2 Výpočtový subsystém

Výpočtový subsystém poskytuje funkcionalitu pre analýzu dát uložených v dátovom sklade. Slúži na spravovanie (orchestráciu) analytických modulov a z nich zostavených analytických operácií. Pod pojmom analytický modul rozumieme konkrétnu izolovanú funkcionalitu realizovanú prostredníctvom Docker [1] kontajnera, ktorého obraz je uložený v repozitári projektu.

Samotné analytické moduly, ktoré boli v súvislosti s riešením projektu vytvorené, môžeme rozdeliť do dvoch skupín – automatizované a interaktívne. Automatizované moduly poskytujú funkciu transformácie daných dát a rôzne formy detekcie na základe predaných parametrov (napríklad indexácia dát, vytváranie deskriptorov obrazových dát, vyhľadávanie v obrazových dátach pomocou neurónových sietí

1. `https://www.docker.com/`

a pod.), zatiaľ čo interaktívne moduly sprístupňujú dáta analytikovi skrze užívateľské rozhranie poskytujúce vlastné pokročilé analytické schopnosti, ako napríklad nástroje Kibana a ElasticSearch, sprístupňujúce napríklad podrobný náhľad na sieťové toky [2].

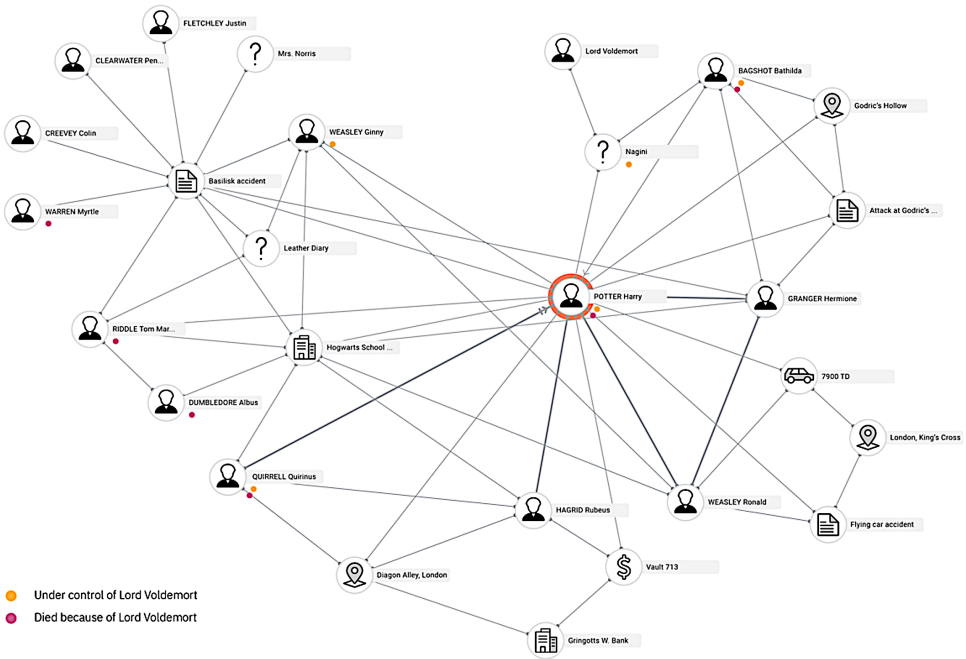
Takéto moduly je možné skladať, vytvárať z nich stromy závislostí a tvoriť tak komplexné analýzy – analytické workflows, v kontexte projektu nazvané operácie, ktorých definícia je témou tejto bakalárskej práce. Príkladom takýchto operácií môže byť analýza finančných transakcií sledovanej osoby a identifikácia osôb, s ktorými táto osoba obchodovala. Na základe ich výsledku môže byť spustený modul na identifikáciu osôb z obrazov diskov či modul pre analýzu telekomunikačného prevozu, ktorý môže potvrdiť, či dotyčné osoby spolu komunikovali [3].

Úlohou výpočtového subsystému je následne preddefinované workflows skrze svoj orchestračný komponent spravovať – spúšťať jednotlivé moduly, kontrolovať ich závislosti, sprostredkovať ich komunikáciu a komunikovať jednak s dátovým skladom, z ktorého preberá dáta a ukladá výsledky analýz, ako aj vizualizačným komponentom, poskytujúcim hlavné užívateľské grafické rozhranie.

2.3 Vizualizačný komponent

Centrálnou časťou vyvíjaného softvéru je z pohľadu užívateľa vizualizačný komponent, ktorý sprístupňuje možnosť zobrazenia dát uložených v dátovom sklade z rôznych pohľadov a predstavuje platformu pre vyvolanie analytických operácií spravovaných orchestračnou službou prostredníctvom výpočtového subsystému.

Základným nástrojom pre policajného analytika je takzvaná vzťahová analýza, pomocou ktorej môže získať komplexný pohľad na dáta reprezentujúce prípad vo forme všeobecného grafu, v ktorom vrcholy tvoria záujmové entity a hrany vzťahy medzi nimi. Vizualizačný komponent rovnako poskytuje množinu nástrojov na filtráciu a vyhľadávanie v záujmových dátach, možnosti agregácie, pridávania nových poznatkov a v neposlednom rade umožňuje spúšťať preddefinované analytické operácie na vybraných dátach skrze výpočtový subsystém, ktorých (medzi)výsledky sú ukladané do dátového skladu.

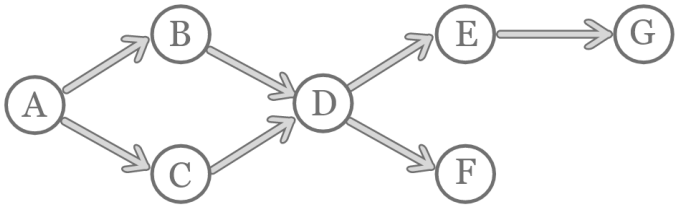


Obr. 2.2: Vzťahová analýza tvoriaca základný náhľad na dáta uložené v dátovom sklade [2]

3 Nástroje na správu workflows

Jednou z najdôležitejších výziev v rámci rôznych aktivít je optimalizácia procesov. Hlavnou prioritou je identifikácia opakujúcich sa činností, ich vyčlenenie na jednotlivé atomické zložky, následná automatizácia a možnosť opakovaného použitia. V tomto kontexte sa často skloňuje slovo workflow, ktoré môžeme zadefinovať ako proces skladajúci sa z niekoľkých krokov, na ktoré sú alokované rôzne zdroje, ktoré ich vykonávajú. Inak povedané, workflow je sieťou úloh, na ktorej môžeme definovať (parciálne) usporiadanie [4]. Koncept workflow je využiteľný v mnohých sferách, od definície úloh vo veľkých spoločnostiach prevádzkujúcich streamovacie služby hudobného obsahu [5], až po preddefinované workflow pomáhajúce v bioinformatike pri analýze genómu [6].

Takto postavená definícia je jednoducho prenositeľná aj do kontextu projektu ANALYZA. Zostavovaním samostatnej, diskrétnej funkcionality do skupiny na sebe závisiacich úloh je možné zadefinovať komplexné analytické operácie (workflows) spravované výpočtovým koponentom navrhnutého systému, ktoré je možné následne rutinne spúšťať na záujmových dátach v procese vyšetrovania trestných činov. Ako vhodná abstrakcia, ktorá je nositeľom požadovanej funkcionality sa ponúka kontajnerizácia jednotlivých čiastkových úloh pomocou Docker kontajnerov. Kontajner je štandardizovaná jednotka softvéru, ktorá obsahuje kód a všetky potrebné závislosti tak, že samotná aplikácia je schopná bežať rýchlo a spoľahlivo, nezáležiac na zvolenej platforme [7]. V nasledujúcich podkapitolách práce budú predstavené vybrané nástroje pre správu workflows, ktoré podporujú princíp kontajnerizácie jednotlivých krokov.

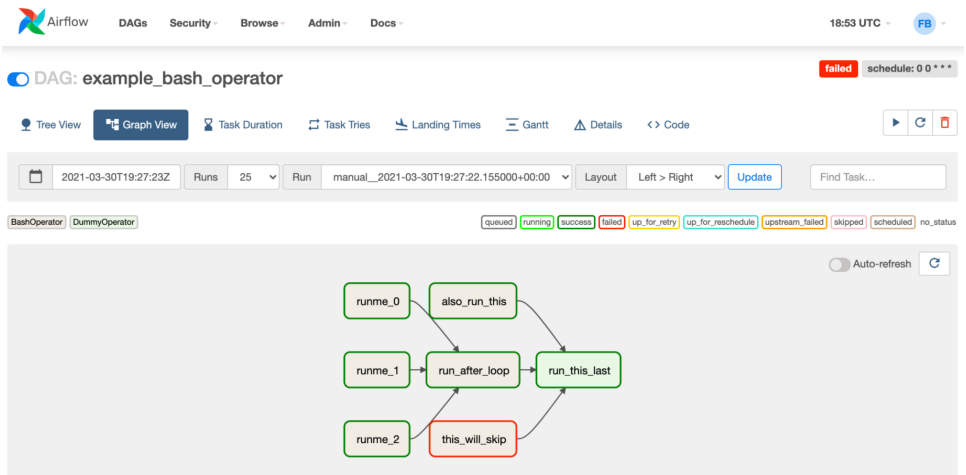


Obr. 3.1: Workflow, sieť úloh, na ktorých je definované parciálne usporiadanie

3.1 Apache Airflow

V súčasnosti jeden z najpopulárnejších nástrojov na správu workflows je Apache Airflow. Vyvinutý spoločnosťou Airbnb, sprístupnený pod Apache licenciu v roku 2015 [8], Airflow predstavuje nástroj, pomocou ktorého je možné definovať workflow ako acyklický orientovaný graf (DAG – z ang. Directed Acyclic Graph). DAG je definovaný ako skript napísaný v jazyku Python, ktorý reprezentuje jeho štruktúru – uzly grafu reprezentujú úlohy a hrany grafu reprezentujú závislosti medzi nimi [9]. Každá z úloh vo workflow je implementáciou rozhrania `Operator`, vďaka čomu je zabezpečená rozmanitosť vykonávaných úloh. Airflow poskytuje mnoho vstavaných operátorov poskytujúcich rôznu funkcionalitu, napríklad operátor na odosielanie mailov (`EmailOperator`), spustenie skriptu (`PythonOperator`), odoslanie HTTP požiadavky (`SimpleHttpOperator`) či spusteie Docker kontajneru (`DockerOperator`). Takto definované workflows je možné automaticky spúšťať v predom určených časoch.

Okrem štandardného rozhrania pre prácu s Airflow skrze príkazový riadok je užívateľovi k dispozícií aj webové užívateľské rozhranie umožňujúce monitorovanie behu, štatistík a vizualizáciu grafu reprezentujúceho workflow.

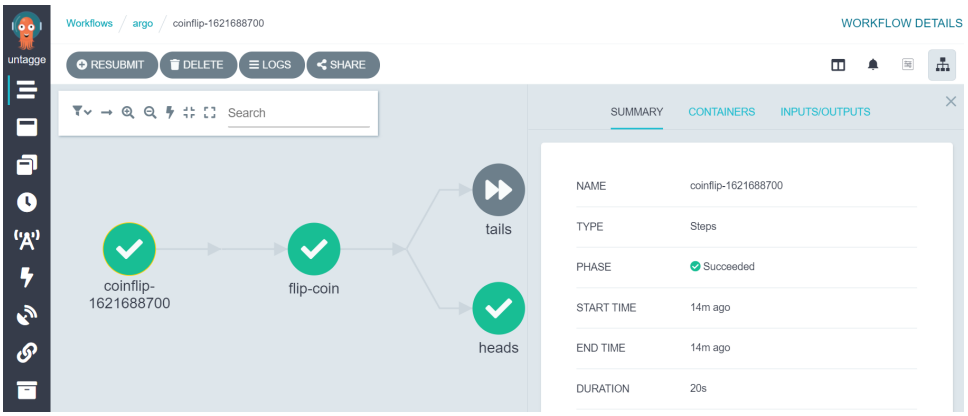


Obr. 3.2: Obrazovka grafického rozhrania Apache Airflow zobrazujúca prostredie grafového pohľadu na závislosti úloh vo workflow [10]

3.2 Argo Workflows

Argo Workflows je open-source nástroj pre správu workflow, ktorý môžeme považovať za priamu nadstavbu orchestračnej služby Kubernetes [2] [11]. Rovnako ako Apache Airflow, aj framework Argo definuje workflow ako orientovaný acyklický graf, každý krok vo workflow však priamo predstavuje funkcionalitu zabalenú ako kontajner, definovaný je ako súbor typu YAML [3]. Okrem natívneho nasadenia kontajnerov na výpočtové uzly pomocou nástroja Kubernetes medzi ďalšie výhody Argo Workflows patria aj parametrizácia úloh či vysoká miera paralelizmu. Argo Workflows umožňuje aj flexibilnejšiu komunikáciu a predávanie dát medzi jednotlivými krokmi workflow. Za nevýhodu oproti Airflow môžeme považovať úzke prepojenie s orchestračnou službou Kubernetes, a teda aj náročnejší proces zoznamovania sa s nástrojom pre ľudí s ním neoboznámených.

Rovnako ako Apache Airflow, aj Argo Workflows poskytuje podobnú funkcionalitu monitorovania workflows skrze grafické rozhranie. Používateľ si môže prezerať definovaný workflow, po kliknutí na ktorýkoľvek uzol je mu umožnený pohľad na bližšie informácie, ako napríklad kontajner stotožnený s daným uzlom, vstupné a výstupné parametre a pod., nemôže však graf priamo interaktívne editovať.



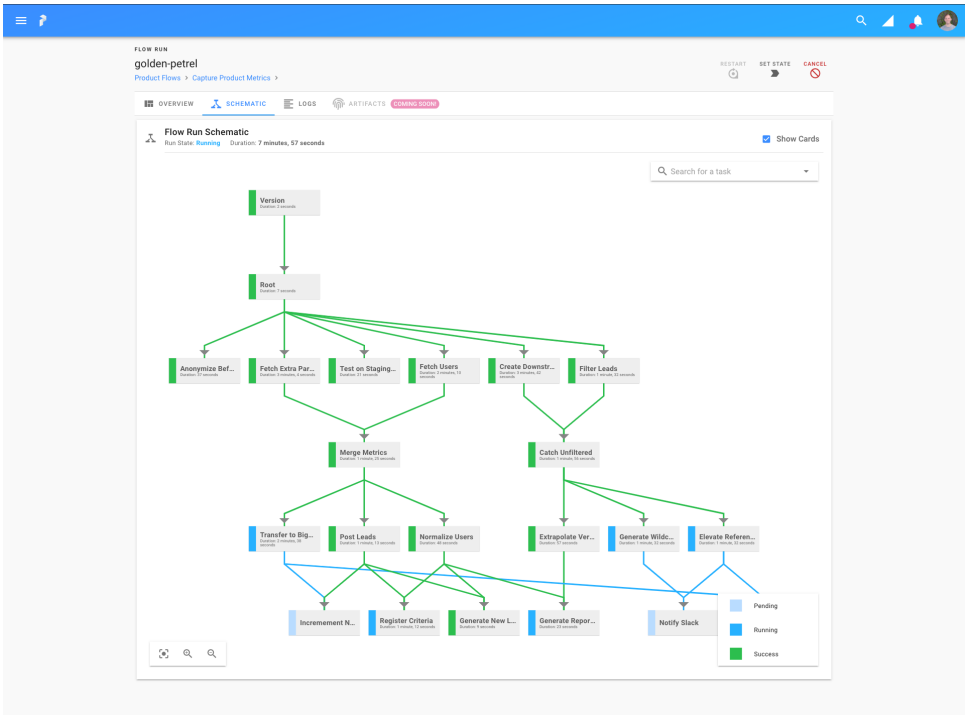
Obr. 3.3: Obrazovka grafického rozhrania Argo Workflows zobrazujúca prostredie grafového pohľadu na závislosti úloh vo workflow

2. https://kubernetes.io/
3. https://yaml.org/

3.3 Prefect

Prefect je jedným z najnovších nástrojov pre manažment workflows. Spôsob vytvárania jednotlivých úloh v rámci workflow je najjednoduchší spomedzi doteraz prestavených nástrojov - úlohou môže byť akákoľvek Python funkcia, ktorá je vhodne označená anotáciou, ktorú Prefect poskytuje. Následne môžu byť zostavované do komplexných štruktúr ako skripty využívajúce poskytnuté API (Aplikačné Programové Rozhranie, z anglického Application Programing Interface). Prefect ďalej poskytuje funkcionalitu pre priame predávanie parametrov medzi úlohami a ponúka možnosť dynamických workflows. Voľne dostupná je však len podmnožina funkcií tohto nástroja, pre využitie pokročilých nástrojov je potrebné zakúpenie licencie.

Prefect poskytuje robustné a moderné grafické rozhranie, v ktorom je možné okrem sledovania rôznych štatistík workflows tieto aj priamo spúšťať, sledovať ich beh a v prípade potreby opätovne štartovať.



Obr. 3.4: Obrazovka grafického rozhrania nástroja Prefect zobrazujúca prostredie grafového pohľadu na závislosti úloh vo workflow [12]

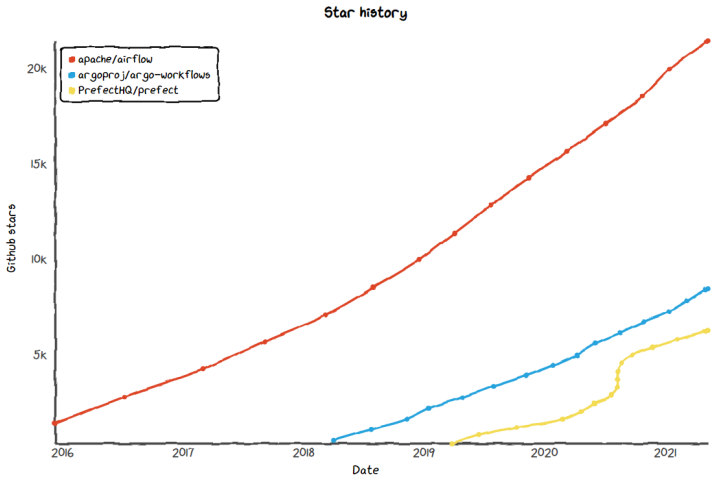
3.4 Iné nástroje

Frameworky na vytváranie pokročilých workflows sa v posledných rokoch stávajú populárnejšími a neustále vznikajú nové. Okrem vyššie spomenutých nástrojov je na trhu množstvo ďalších riešení, ktoré si sú navzájom v princípe podobné, líšia sa zväčša účelom, zložitosťou vytvárania workflow, spôsobom ich definície a rôznymi pokročilejšími funkciami. Za všetky spomeniem predovšetkým Luigi, knižnicu jazyka Python, udržiavanú dátovým tímom spoločnosti Spotify [5], ktorá sprostredkováva obmedzenejšiu funkcionalitu a grafické prostredie ako napríklad Airflow, či MLFlow, platformu špeciálne vytvorenú pre správu životného cyklu úloh a experimentov v oblasti strojového učenia [13].

4 Analýza požiadaviek

Ako súčasť výpočtového komponentu projektu ANALYZA bolo vyvinuté vlastné riešenie pre správu analytických workflows z čiastkových úloh. Jednotlivé moduly, predstavujúce kontajnery, sa skladajú do analytických operácii, predstavujúcich workflow. Operácie sú následne spúšťané z prostredia vizualizačného komponentu. Myšlienkovo najbližšie k výpočtovému komponentu projektu má Argo Workflows, ktorý je priamou nadstavbou orchestračnej platformy Kubernetes, ktorá je poskytovateľom orchestračných služieb aj v rámci projektu. Na nevyužití tohto nástroja alebo iných vyššie predstavených sa podpísali najmä dva faktory:

- **nedostupnosť** – zo všetkých v predchádzajúcej kapitole spomínaných nástrojov bol počas návrhu architektúry a počiatočných fáz implementácie vyvíjaného softvéru jediným dostupným Apache Airflow. Základným dôvodom, ktoré viedli k jeho nevyužitiu je v danom čase chýbajúca podpora pre úlohy vo forme Docker kontajnerov, ktorá bola pridaná v roku 2018. Vývoj nástrojov na správu workflows v čase je zobrazený v grafe na Obrázku 4.1.



Obr. 4.1: História projektov pre správu workflow vzhľadom k ich popularite na platforme Github vyvýjajúcej sa v čase, vyjadrenej v počte hviezd

- **požadovaná dodatočná funkcionalita** – keďže sa jedná o špecifický projekt v úzko špecifikovanej doméne, komerčne dostupné riešenia neprinášali všetku požadovanú funkcionalitu. Jednou z dodatočne implementovaných požiadaviek bolo riešenie správy bežiacich kontajnerov a ich prípadne znovu použitie v inej operácií, využívajúcej ten istý modul ako súčasť svojho workflow.

Z týchto dôvodov bola funkcionalita správy analytických workflows rozdelená do troch primárnych častí:

- **správa workflows** – výpočtový komponent projektu má za úlohu spúšťanie, správu a riadenie chýb kontajnerov, s pomocou orchestračnej služby Kubernetes. Interne si zaznamenáva stav aktuálne bežiacich kontajnerov a v prípade potreby umožňuje ich opätovné použitie.

- **spúšťanie workflows** – výber dát, na ktorých sa spustí analytický workflow a následné spustenie s prípadnými požadovanými parametrami je užívateľovi prístupné skrze vizualizačný komponent systému.

- **definícia workflows** – spôsob, akým sa definície modulov a z nich zložených workflow dostanú do prostredia výpočtového komponentu tak, že ten je následne plne schopný workflow spúšťať a riadiť.

Práve definícia workflows pre projekt ANALYZA je témou tejto bakalárskej práce. Vzhľadom k vytváranej platforme vzišla požiadavka na vytvorenie grafického rozhrania pre definovanie a správu analytických modulov a operácií. Na základe konzultácií s projektovým vedením a tvorcom výpočtového subsystému boli zadefinované nasledujúce požiadavky na vyvíjaný softvér.

4.1 Funkčné požiadavky

Primárnym cieľom grafického prostredia pre správu worflows bude poskytnúť jednoduché a jednotné prostredie pre definíciu analytických

modulov a operácií. Na rozdiel od nástrojov predstavených v predchádzajúcej kapitole, ktoré definovali workflow v jazyku Python alebo YAML súborom, tento nástroj nebude vyžadovať zručnosti akéhokoľvek programovacieho jazyka či framework-u. Grafické prostredie poskytne možnosť zadefinovať a do prostredia orchestrátora a výpočtového komponentu nahrať všetky potrebné definície modulov a operácií. Vyvinutý systém overí korektnosť dát pred ich nahraním do prostredia orchestrátora.

Súčasťou definície analytického modulu je aj obraz kontajneru, ktorý zapuzdruje jeho funkcionalitu. Obraz kontajneru je samostatný, spustiteľný balík obsahujúci všetko potrebné na jeho beh– kód, prostredie, nástroje, knižnice a nastavenia [7]. Súčasťou vyvíjaného nástroja teda bude aj prostredie, ktoré umožní užívateľovi nahrávať obrazy týchto kontajnerov do privátneho Docker repozitára projektu, prezerať si aktuálne dostupné obrazy a tieto spravovať.

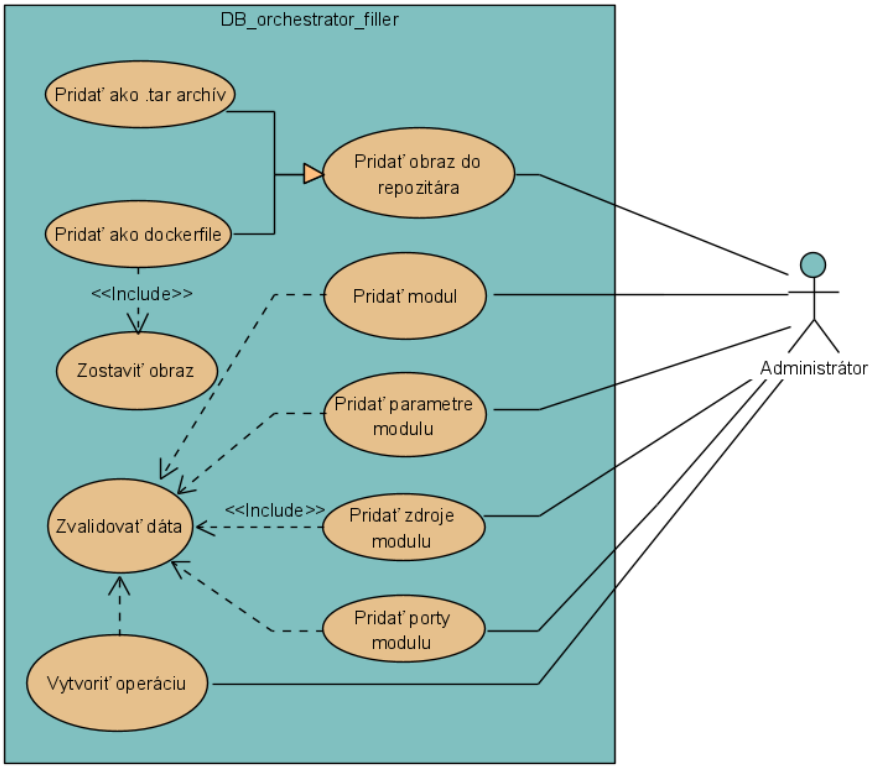
Jedným zo základných prostriedkov, ktoré v procese vývoja softvéru umožňujú zachytiť funkčné požiadavky je UML diagram prípadov použitia. Ten vo fáze analýzy cieľovej domény môže slúžiť ako prostriedok vhodne zhŕňujúci požiadavky na vyvíjaný systém, vyobrazuje aktérov a nimi očakávanú funkcionalitu. Očakávané funkcie grafického prostredia som zhrnul v diagrame prípadov použitia, vyobrazenom na Obrázku 4.2.

4.2 Nefunkčné požiadavky

Nefunkčné požiadavky pri vývoji softvéru definujú vlastnosti a obmedzenia kladené na vyvíjaný softvér. Pre daný nástroj boli vymedzené nasledujúce nefunkčné požiadavky, ktorých splnenie sa očakáva:

- **portabilita** – aplikácia by mala byť prenositeľná a spustiteľná na rôznych platformách, používateľ bude môcť aplikáciu používať nezávisiac od ním zvoleného operačného systému,
- **internacionalizácia** – aplikácia by mala poskytovať jazykový výstup v českom aj anglickom jazyku,
- **logovanie chýb** – v prípade výskytu chýb a neočakávaných udalostí budú tieto zaznamenané v separátnom súbore vo formáte vhodnom na odhalenie vzniknutej chyby,

- **užívateľská prívetivosť** - v neposlednom rade by aplikácia mala poskytnúť jednoduché a atraktívne užívateľské prostredie.



Obr. 4.2: Diagram prípadov použitia zobrazujúci funkčné požiadavky kladené na výsledný softvér

5 Návrh

Na základe požiadaviek kladených na vytvárané grafické prostredie opísaných v Kapitole 4 som sa rozhodol rozdeliť aplikáciu do troch hlavných sekcií, ktoré budú bližšie popísané v nasledujúcich podkapitolách.

5.1 Sekcia 1: Obrazy

Prvá časť aplikácie umožní užívateľovi prácu s obrazmi kontajnerov v Docker registri projektu. Register obrazov je ich úložisko zložené z repozitárov obrazov kontajnerov, každý obraz je v rámci repozitára jednoznačne identifikovateľný svojim tagom.

Docker Registry HTTP API v2 je aplikačné programové rozhranie umožňujúce komunikáciu s inštanciou registra, poskytuje služby na spravovanie obrazov Docker kontajnerov a umožňuje ich distribúciu [14]. Aplikácia užívateľovi prostredníctvom požiadaviek kladených na Docker Registry API umožní transparentne prehliadať obsah registra zadefinovaného v konfiguračnom súbore aplikácie, prípadne obrazy z registra zmazať.

Rovnako dôležitou súčasťou práce s obrazmi kontajnerov je aj vytváranie nových obrazov. Vyvíjané prostredie užívateľovi umožní obraz kontajneru do registru nahrať, pričom definícia obrazu môže byť poskytnutá v dvoch formách:

- **ako archív s príponou .tar** – jednou z možností reprezentácie obrazu Docker kontajneru je .tar súbor. Aplikácia užívateľovi umožní nahrať takýto súbor do registra pod daným názvom repozitára a tagom.

- **ako DockerFile** – DockerFile je textový dokument obsahujúci všetky príkazy, ktoré by užívateľ zadal manuálne pre zostavenie obrazu kontajneru, platforma Docker na základe inštrukcií z DockerFile-u obraz zostaví za neho [15]. Aplikácia teda pred nahraním obrazu do registra tento na základe inštrukcií popísaných v DockerFile zostaví.

5.2 Sekcia 2: Moduly

Základnou funkčnou jednotkou, s ktorou sa v kontexte projektu ANALYZA operuje, je analytický modul. Okrem obrazu Docker kontajneru, ktorý reprezentuje jeho funkcionalitu, je modul tvorený dodatočnými informáciami, ktoré sú potrebné pre jeho spustenie. Konkrétne ide o:

- **základné atribúty modulu**, ako sú napríklad jeho meno, sieťový port, na ktorom bude jeho služba poskytnutá, typ modulu (interaktívny alebo automatizovaný) a pod.,
- **zdroje modulu**, ktoré definujú potrebné informácie pre orchestračnú službu vzhľadom k nasadeniu modulu na výpočtové uzly,
- **parametre modulu**, ktoré umožňujú personalizovať konkrétny modul s vlastnými parametrami, ktoré budú využité pri jeho behu a zadefinovať parametre, ktoré modul pri spustení očakáva,
- **porty modulu**, ktoré definujú sieťové porty, ktoré služba predstavujúca daný modul pre svoj beh očakáva.

Modul, rovnako ako aj operácia, je vo výpočtovom komponente reprezentovaný ako záznam v relačnej databáze. Relačná databáza je nástrojom, ktorý sprostredkováva perzistentné ukladanie štruktúrovaných dát vo forme záznamov v databázových tabuľkách.

Vyvinutá aplikácia poskytne v svojej druhej časti možnosť prevádzať CRUD [4] operácie nad zvolenými dátami modulov. Ústredná časť poskytne pohľad na všetky dostupné moduly, spolu s možnosťou vyhľadávania a triedenia. Po dvojkliku na daný modul aplikácia užívateľovi sprístupní zadefinované parametre a porty s možnosťou ich správy. Pre lepšiu prehľadnosť si užívateľ v detaile modulu bude môcť prehliadnuť všetky operácie využívajúce daný modul.

Dôležitou súčasťou vytvárania modulu a jeho atribútov je zabezpečenie konzistencie zadávaných dát. Aplikácia teda pred samotným vytvorením otestuje validitu zadávaných dát. Aplikácia poskytne kontrolu pre nasledujúce scenáre:

4. CRUD je skratkou pre štyri základné operácie pre prácu s databázou: Create (Vytvoriť), Read (Zobraziť), Update (Aktualizovať) a Delete (Zmazať).

- **duplicita mien** – aplikácia overí, že v systéme sa nenachádza modul, port či parameter s rovnakým menom,

- **korektnosť poskytnutého portu** – číslo portu musí byť v rozsahu $0 - (2^{16} - 1)$,

- **jedinečnosť portu** – aplikácia musí zabezpečiť unikátnosť čísla portu pre daný modul.

Vo všetkých spomenutých prípadoch aplikácia informuje užívateľa o tom, že došlo k porušeniu integrity dát.

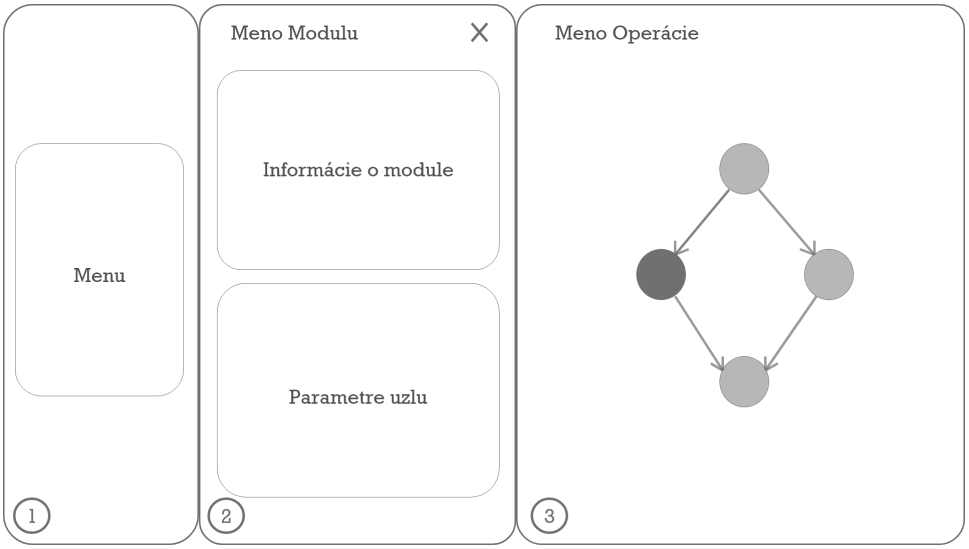
Pôvodný návrh aj implementácia počítali s vytvorením len dvoch sekcií, kde boli zlúčené sekcie obrazov a modulov. Obraz bol neoddeliteľnou súčasťou modulu a nahrával sa do registra priamo pri jeho definícii. Neskôr však po diskusiách došlo k rozhodnutiu oddeliť tieto dve časti aplikácie. Dôsledok tohto kroku je väčšia miera flexibility a modularity zavádzania modulov. Užívateľovi bude takto umožnené pri definícií modulu nevkladať do systému aj jeho obraz, ale vybrať si z obrazov už nachádzajúcich sa v repozitári. Tento modulárny prístup nielen prehľadne oddelí jednotlivé zodpovednosti a zjednoduší prácu s aplikáciou, ale zároveň umožní výber rovnakého obrazu pre množstvo definícii modulov.

5.3 Sekcia 3: Operácie

Definície operácie v systéme sú cieľovou funkcionalitou. Na rozdiel od nástrojov popísaných v rešeršnej časti práce v Kapitole 3 bola na základe špecifikácie požiadaviek vznesená požiadavka na vytvorenie nástroja, ktorý umožní vytvárať workflows zostavené z modulov, predstavujúcich jednotlivé čiastkové kroky, užívateľsky prívetivým spôsobom. Na základe grafických rozhraní nástrojov ako Argo Workflows či Apache Airflow bolo navrhnuté grafické prostredie na vytváranie a editáciu operácií, navrhnuté prostredie však poskytne užívateľovi interaktívny prístup k definícii workflow, a to priamou manipuláciou s grafom, ktorý workflow reprezentuje.

5.4 Grafický návrh

V priebehu návrhu bolo spracované prototypové riešenie rozmiestnenia grafických prvkov aplikácie v nástroji Adobe XD, a to pre sekciou práce s operáciami, ktorá je poslednou, ústrednou časťou pri definícii analytických workflows.



Obr. 5.1: Návrh rozloženia obrazovky pre manipuláciu s operáciami

Obrazovka pre zostavovanie workflows, ktorej návrh je načrtnutý na Obrázku 5.1, bude vertikálne rozdelená do troch častí. Prvá časť, spoločná pre všetky časti aplikácie, bude slúžiť na orientáciu medzi sekciou obrazov, modulov a operácií a nastaveniami aplikácie. Tretia, centrálna časť poskytne rozhranie pre interaktívnu prácu s orientovaným acyklickým grafom [5], ktorý predstavuje operáciu a definuje workflow. Jednotlivé uzly grafu budú predstavovať moduly. Užívateľovi bude po kliknutí na možnosť pridania nového uzlu umožnený výber z modulov, ktoré sú v databáze orchestrátora uložené a vybraný modul bude s uzlom grafu stotožnený. Užívateľ bude môcť pridávať hrany grafu, reprezentujúce závislosti. Hrana v grafe znamená, že

5. Pojem orientovaný graf(G) môžeme definovať ako usporiadanú dvojicu: $G = (V, E)$, kde V je konečná množina vrcholov grafu a E množina hrán, pričom hrany sú usporiadanou dvojicou vrcholov z množiny V.

modul, v ktorom hrana začína, musí byť spustený pred modulom, do ktorého orientovaná hrana vstupuje.

Po dvojitom kliknutí na ktorýkoľvek uzol grafu sa užívateľovi v časti dva zobrazia bližšie informácie o module reprezentujúcom daný uzol. Koncept uzlov umožňuje v rámci každej operácie pre moduly tvoriace workflow prepísať parametre, ktoré boli zadané v definícií konkrétneho modulu. Aplikácia teda v rámci druhej časti umožní takéto parametre uzlov zobrazovať, prepisovať a mazať.

6 Implementácia

V tejto kapitole budú bližšie priblížené využité nástroje, technológie a architektúra zvolená s ohľadom na dosiahnutie cieľov definovaných v predchádzajúcich kapitolách, ako aj implementačné výsledky práce.

6.1 JavaFX

Pre implementáciu grafického prostredia bola zvolená forma natívnej desktopovej aplikácie v jazyku Java. JavaFX je nástrojom, ktorý bol pre vytvorenie grafického rozhrania vybraný. JavaFX predstavuje open-source nástroj na vytváranie moderných desktopových klientov v jazyku Java, distribuovaný ako samostatný modul spravovaný pod OpenJFX [6] projektom. Framework je jednoduché do projektu importovať, a to ako závislosť zostavovacieho nástroja Maven [7], vďaka ktorému je aplikácia jednoducho zostaviteľná a spustiteľná.

6.2 Architektúra

Architektúru softvéru môžeme charakterizovať ako podobu softvéru, ktorá ho rozdeľuje do logických častí (komponentov), ich vzájomné usporiadanie a komunikáciu [16].

6.2.1 MVC

Medzi najrozšírenejšie architektonické návrhové vzory pri vývoji softvéru grafických rozhraní patrí vzor MVC (z ang. *Model-View-Controller*). Ten člení softvér na tri základné zložky – *Model*, *View* a *Controller*, s hlavným cieľom separácie zodpovednosti, zníženia komplexnosti a náročnosti na vývoj [17]. Mnou vybraný framework JavaFX je navrhnutý priamo s myšlienkou podpory vzoru MVC.

---

6. `https://openjfx.io/`
7. `https://maven.apache.org/`

View

View definuje základné grafické rozhranie, s ktorým užívateľ priamo interaguje. Primárnym prvkom pre vytváranie *View* v JavaFX je FXML dokument. Je to súbor typu XML, vďaka ktorému je možné deklaratívnym spôsobom definovať *Views* [18], ktoré sú nezávislé od implementácie programovej logiky. Okrem priamej možnosti vytvárania FXML súboru sa ponúka aj možnosť editovania a manipulácie s vytváraným grafickým prostredím pomocou nástroja SceneBuilder [8], ktorý umožňuje manipuláciu s vizuálnymi prvkami v reálnom čase.

Dôvodom jednoduchosti použitia deklaratívneho prístupu a nástroja SceneBuilder je štruktúra, akou JavaFX definuje *Views*. Centrálnym prvkom UI je okno aplikácie, reprezentované triedou `Stage`. Každá obrazovka má následne scénu (`Scene`), ktorá je nositeľom takzvaného scénického grafu. Graf scény je stromová dátová štruktúra, pričom všetky elementy, ktoré sú súčasťou scénického grafu nazývame uzly, tieto prvky dedia z abstraktnej triedy `Node` [19].

Uzly grafu scény je možné rozdeliť na dva hlavné typy. Prvým sú uzly, ktoré môžu mať potomkov v grafe scény. Najdôležitejšou skupinou v rámci tohto typu uzlov sú prvky predstavujúce kontajnery držiace grafické elementy grafu pohromade a umožňujúce rôzne druhy rozmiestnenia grafických prvkov na obrazovke. Tieto uzly sú podtriedami triedy `Pane`. Druhú skupinu tvoria listové uzly grafu – prvky, ktoré nemôžu mať potomkov, typicky ide o priame vizuálne prvky, ako napríklad textové pole, tabuľka či tlačidlo. Vďaka takto postavenej definícii grafického rozhrania je vytváranie užívateľských rozhraní jednoduché a intuitívne pomocou FXML súboru, ktorého štruktúra korešponduje s grafovou štruktúrou scénického grafu.

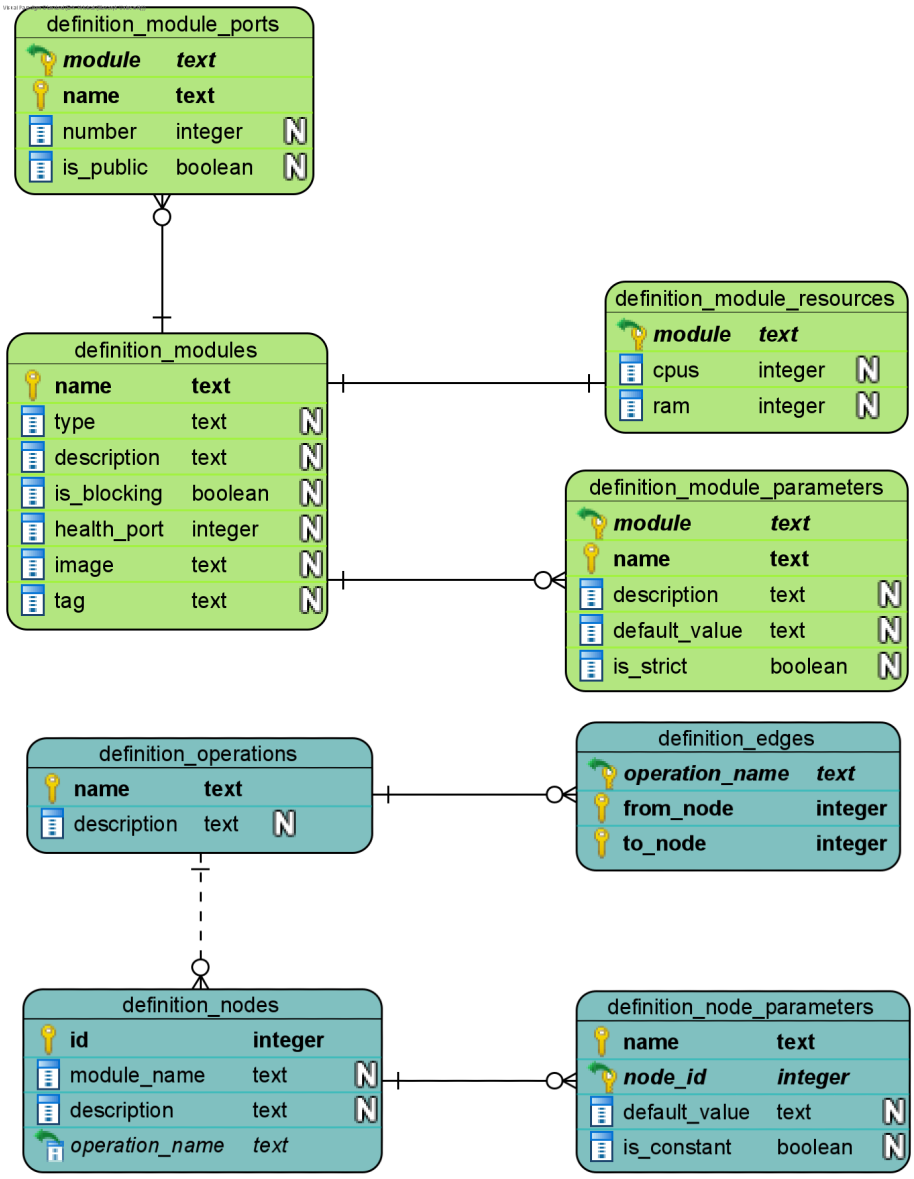
Vytvorené *Views* pre všetky obrazovky aplikácie je možné nájsť v zložke `resources/view` v priloženej implemenácii.

Model

Model predstavuje dáta, ktoré aplikácia spracováva a ktoré *View* zobrazuje. Dátový model použitý pri vytváraní aplikácie vychádza z modelu potrebného pre naplnenie orchestrátora dátami a kopíruje relačnú schému databázy, ktorú výpočtový komponent projektu používa. Dá-

8. `https://gluonhq.com/products/scene-builder/`

tový model vo forme ERD diagramu, zachycujúci modelové entity a vzťahy medzi nimi je vyobrazený na Obrázku 6.1.



Obr. 6.1: ERD diagram zobrazujúci dátový model aplikácie

Controller

Prepojením medzi *Model*-om a *View* je časť s názvom *Controller*. *Controller* umožňuje manipulovať s grafickými prvkami *View*, ovládať ich a poskytovať *Model* tak, aby *View* bolo nezávislé od implementácie *Controller*-a a zvoleného *Model*-u.

FXML súbor poskytuje samostatné atribúty definujúce, ktorý *Controller* k danému FXML dokumentu prislúcha. Rovnako každý uzol scénického grafu má voliteľný atribút `fx:id`, ktorý je jednoznačným identifikátorom uzlu a je ho tak možné referencovať v *Controller*-i prislúchajúcemu danému *View*. Balík `javafx.fxml` obsahuje anotáciu `@FXML`, ktorá signalizuje, že ňou označený prvok s rovnakým názvom, ako hodnota kľúča v `fx:id` uzlu *View*, má byť do prostredia *Controller*-a pri inicializácií *View* vložený. Triedy zastupujúce *Controller* taktiež umožňujú implementovať všetku potrebnú funkcionalitu.

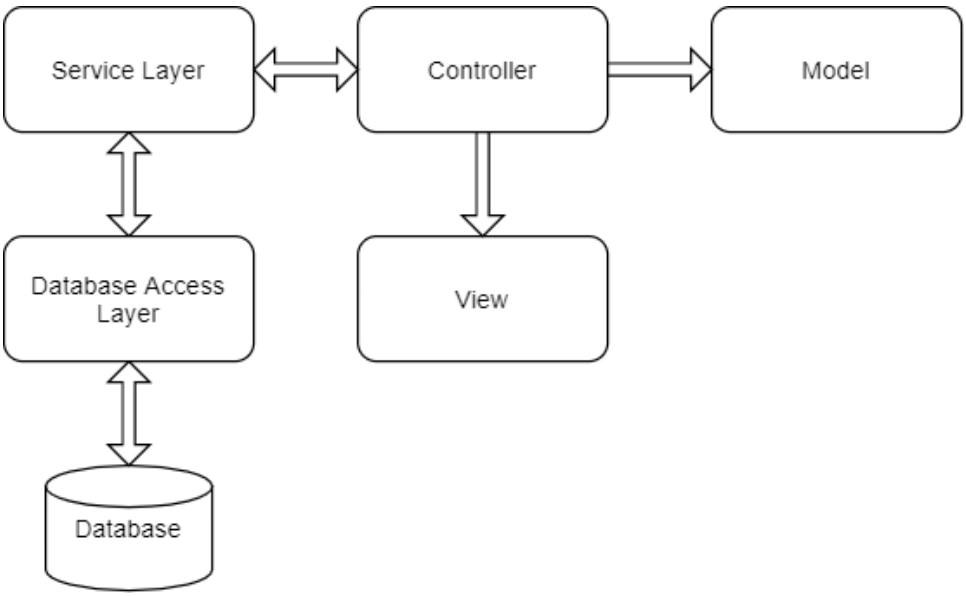
Triedy reprezentujúce *Controller* sú implementované v balíku `controller`. Spolu s *Controller*-mi pre náhľady obrazov kontajnerov, modulov a operácií sa v tomto balíku nachádzajú aj rôzne nástroje. Generické rozhranie `Loadable`, definujúce metódu `load()`, umožňuje triedam, ktoré ho implementujú, predávanie dát medzi jednotlivými *Controller*-mi pri načítavaní *Views*. Trieda `ViewLoader` poskytuje metódy pre načítanie *View* na základe názvu FXML súboru, rovnako ako metódu, ktorá zároveň načíta do prostredia *Controller*-a patriacemu k danému *View* dáta.

6.2.2 Trojvrstvová architektúra

Odhliadnúc od logického rozdelenia prezentačnej vrstvy podľa vzoru MVC je aplikácia horizontálne vyčlenená na ďalšie dve vrstvy. Okrem prezentačnej, ktorá bola bližšie predstavená v predchádzajúcich podkapitolách, sú ďalšími vrstvami servisná a vrstva prístupu k databáze. Každá z vrstiev v takto navrhnutom systéme závisí len od vrstvy umiestnenej v hierarchii pod ňou, spracováva výsledky jej volaní a poskytuje službu vrstve nadradenej. V programovacom jazyku Java sú tieto vrstvy typicky reprezentované balíkmi, komunikujú spolu skrze definované rozhrania [16].

Jednou z podstatných častí aplikácie je perzistentné uloženie dát o vytvorených moduloch a operáciách. Dátová vrstva, reprezentovaná

balíkom `databaseAccess`, slúži k interakcií aplikácie a externého úložiska. Servisná vrstva zapuzdrená v balíku `service` komunikuje s databázovou vrstvou skrze navrhnuté rozhranie, spracúva jej výstupy, implementuje prípadnú požadovanú logiku a posúva dáta prezentačnej vrstve, v kontexte tejto aplikácie *Controller*-u, ktorý následne updatuje *View*. Schéma navrhnutej architektúry je zachytená na Obrázku 6.2



Obr. 6.2: Schéma zobrazujúca architektúru implementovanej aplikácie

6.3 Obrazy modulov

Kategória obrazov modulov je vstupnou bránou pre prácu s aplikáciou. Na základe požiadaviek definovaných v Kapitole 4.1 bolo zjavne potrebné, aby sa pri komunikácií so vzdialeným Docker registrom, ako aj pri zostavovaní obrazov Docker kontajnerov komunikovalo asynchrónnym spôsobom. Je potrebné, aby hlavné vlákno aplikácie (`JavaFxApplicationThread`) pri týchto operáciách nezamrzlo a užívateľ mohol s aplikáciou interagovať aj počas zostavovania a nahrávania obrazov kontajnerov a aby mohol spúšťať viacero takýchto úkonov naraz. Za týmto účelom bola implementovaná séria asynchrónnych úloh, rozširujúcich generickú triedu `Task<T>` z balíka `javafx.concurrent`.

V každej implementácii triedy `Task<T>` je prepísaná metóda `call()`, ktorá enkapsuluje požadovanú funkcionalitu, vykonáva svoj kód vo vlákne na pozadí, a tým zabezpečuje fluidnosť prostredia aj pri dlho trvajúcich úlohách [20].

Keďže jediné vlákno umožňujúce korektnú manipuláciu so scénickým grafom je hlavné aplikačné vlákno `JavaFxApplicationThread`, poskytuje JavaFX sadu atribútov, na ktorých pozorovanie sa volajúci z hlavného aplikačného vlákna môže prihlásiť. Ide o údaje akými sú napríklad aktuálny stav vlákna, progres, či rôzne druhy správ, ktoré sa môžu počas vykonávania úlohy zadanej vláknu adekvátne meniť a reflektovať tak aktuálne dianie vo vlákne na pozadí. Akékoľvek zmeny v týchto atribútoch sa pri vhodnom použití prejavia zmenami v užívateľskom rozhraní.

Aplikácia pri práci so službou Docker využíva vo vláknach na pozadí primárne dva druhy prístupov. Docker Registry HTTP API v2 je aplikačné programové rozhranie, ktoré je využívané na manipuláciu s obrazmi kontajnerov uloženými v privátnom registri projektu – prehliadanie repozitárov, tagov a ich prípadné zmazanie. Aplikácia pripraví vhodné HTTP dotazy na aplikačné rozhranie, pričom používa dáta ako užívateľské meno, heslo, cestu k certifikátu na autentizáciu a adresu registra načítaním z konfiguračného súboru aplikácie. Aplikácia následne dáta upraví do podoby vhodnej na zobrazenie a vráti volajúcemu *Controller*-u.

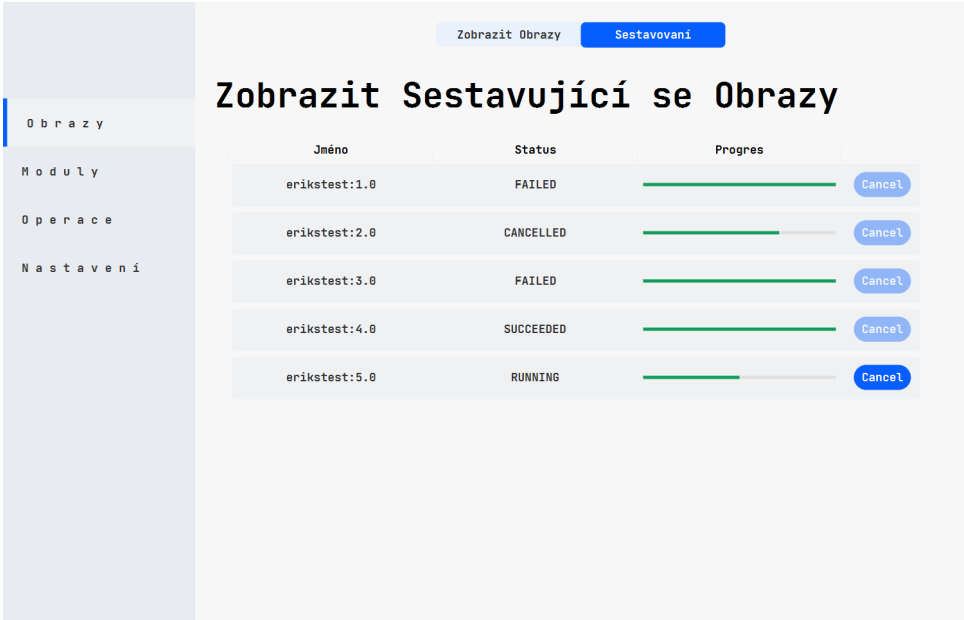
Druhú funkcionalitu zabezpečuje vhodné využitie knižnice docker-java. Tá reprezentuje wrapper nad Docker Engine API a sprostredkuje komunikáciu s Docker démonom za účelom zostavenia obrazu z DockerFile alebo načítanie obrazu vo forme .tar archívu a jeho následné nahratie do vzdialeného registra projektu.

Aplikácia užívateľovi umožní zvoliť, či má záujem o zostavenie obrazu z DockerFile alebo nahratie .tar archívu, v tom prípade očakáva od užívateľa korektné zadanie názvu repozitára a identifikujúceho tagu. Následne môže užívateľ sledovať stav vykonávanej úlohy v tomu prislúchajúcej tabuľke a v prípade potreby vlákno zastaviť. V prípade, že dôjde k akejkoľvek chybe, či už zadaním nekorektného archívu, problémom so službou Docker či pripojením k registru dôjde k vyhodeniu výnimky v danom vlákne a užívateľovi sa v tabuľke prebiehajúcich úloh zobrazí popis FAILED indikujúci zlyhanie. V prí-

pade úspešného nahrania do registra je užívateľ oboznámený stavovou správou SUCCEEDED.

Grafické rozhranie slúžiace na monitorovanie aktuálne zostavujúcich sa obrazov kontajnerov, ktoré je výsledkom implmenetácie, je zobrazené na Obrázku 6.3.

Triedy spracovávajúce všetky asynchrónne úlohy sú súčasťou balíka `tasks`.



Obr. 6.3: Grafické rozhranie monitorovania aktuálne zostavujúcich sa obrazov kontajnerov

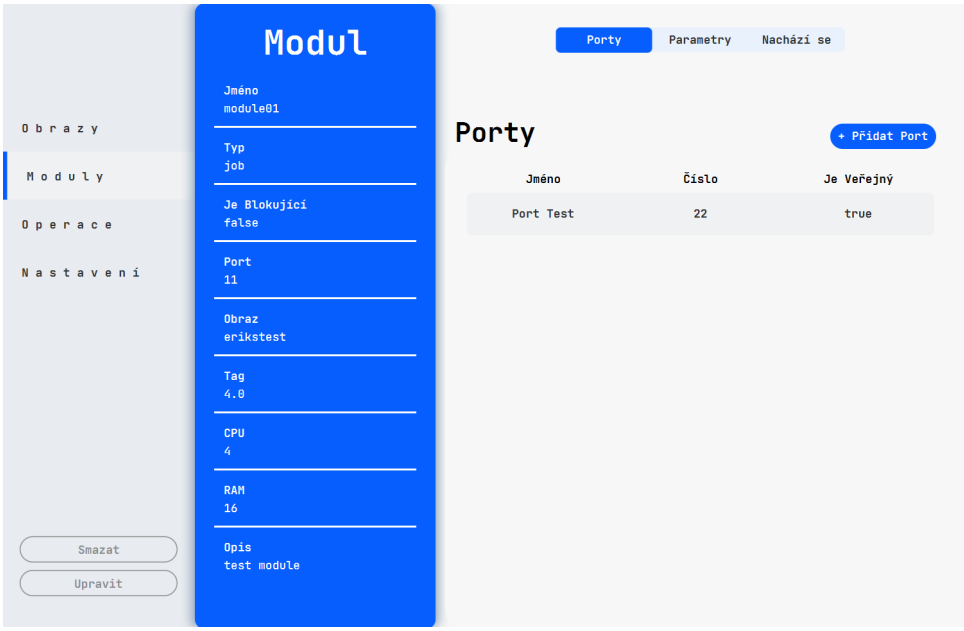
6.4 Moduly

Hlavnou úlohou tejto časti implementácie bolo umožniť CRUD operácie nad samotnými modulmi. Za cieľom komunikácie s databázou a vytvorenia modelu, ktorý popisuje ERD diagram na Obrázku 6.1, bolo zvolené riešenie ORM (z ang. Object-Relational Mapping, Objektovo-Relačné Mapovanie)

ORM prístup k riešeniu perzistencie umožňuje mapovanie databázových záznamov na objekty v objektovo orientovanom jazyku, akým je aj Java. Každej tabuľke, ktorá je definovaná v ERD modeli

na Obrázku 6.1, bola vytvorená samostatná entita v podobe triedy v balíku `entity`, ktorá ju reprezentuje. Takéto triedy vhodne označené anotáciami z Java Persistance API (JPA), ktoré je štandardizovaným rozhraním pre riešenie perzistencie v programovacom jazyku Java, je možné pomocou vhodného ORM frameworku premeniť na databázové záznamy. Najpopulárnejším ORM nástrojom pre jazyk Java je Hibernate. Ten je poskytovateľom implementácie pre JPA a v rámci vrstvy databázového pripojenia aplikácie poskytuje napojenie na databázu definovanú v konfiguračnom súbore aplikácie. ORM prístup spolu s rozdelením aplikácie do viacerých vrstiev poskytuje oddelenie rozdielnych funkcionalít a transparentnú manipuláciu s dátami, ktorých trvalé uloženie je potrebné.

Obrázok 6.4 zachycuje obrazovku umožňujúcu bližší pohľad na konkrétny zvolený modul. Panel v ľavej časti obrazovky poskytuje bližší pohľad na vybraný modul, pravá časť obrazovky zasa rozhranie pre prácu s portami, parametrami a operáciami, v ktorých sa daný modul vyskytuje ako súčasť workflow.



Obr. 6.4: Grafické rozhranie pre prácu s vybraným modulom

6.5 Operácie

Vzhľadom k požiadavkám a návrhu prezentovanému v Kapitole 5.4, prvým krokom implementácie bolo nájdenie vhodnej knižnice kompatibilnej s nástrojom JavaFX, ktorá poskytuje funkcionalitu zobrazenia a editácie všeobecného grafu. V nasledujúcich odsekoch práce sú predstavané možnosti, ktoré boli v priebehu implementácie zvažované.

yFiles yFiles for JavaFX je jedným z najpokročilejších nástrojov poskytujúcich prácu s grafmi. Ponúka veľké množstvo layoutov a vizuálov, umožňuje zobrazovať dynamické dáta aj automatické usporiadanie grafov. Okrem základnej funkcionality, ktorá umožňuje interaktívne vytváranie dát ďalej umožňuje pokročilé formy náhľadov, zoskupovanie uzlov do kategórií, ich animáciu a filtrovanie [21].

JUNG JUNG (Java Universal Network/Graph Framework) je open-source knižnica, ktorá má na starosti modelovanie, analýzu a vizualizáciu dát majúcich grafovú reprezentáciu [22]. Okrem očakávanej funkcionality poskytuje aj funkcie rôznych grafových algoritmov, ako je napríklad hľadanie najkrajšej cesty Dijskstrovým algoritmom či minimálnej kostry grafu, rovnako sprístupňuje rôzne druhy štatistických funkcií nad predmetným grafom.

JavaFXSmartGraph JavaFXSmartgraph je JavaFX knižnica pre vizualizáciu všeobecných grafov s podporou force-directed algoritmu na ich prehľadné usporiadanie v reálnom čase [23]. Ponúka okrem možnosti výberu stratégie usporiadania uzlov grafu aj možnosť ich grafickej úpravy prostredníctvom css súboru. Nástroj umožňuje vytvárať ako orientované, tak aj neorientované grafové reprezentácie, tieto dynamicky upravovať a jednoducho zachytávať udalosti ako napríklad dvojklik na uzol či hranu grafu.

6.5.1 Výber vhodného nástroja

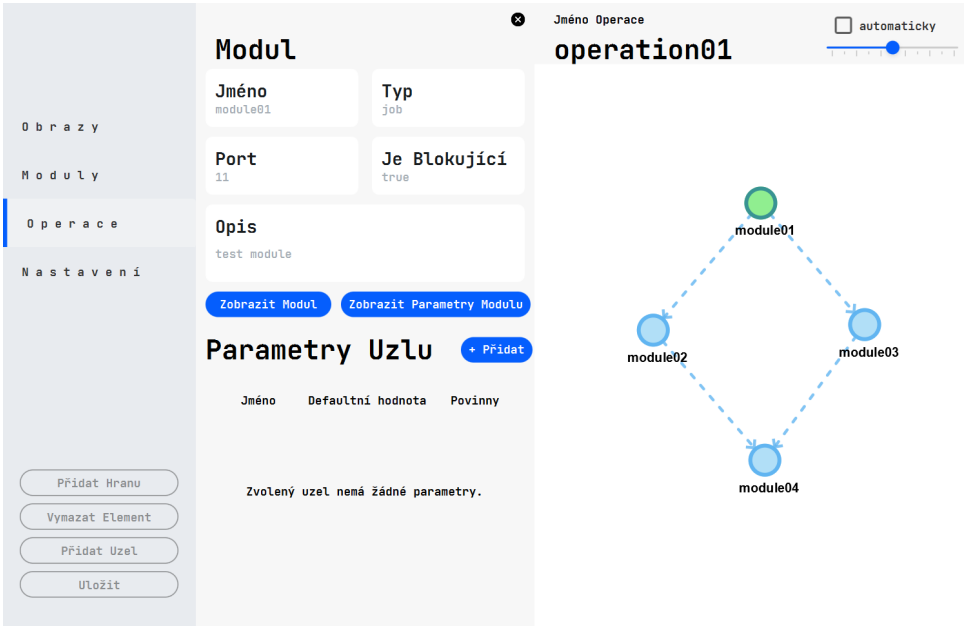
Vzhľadom k definovaným požiadavkám bola na implementáciu interaktívneho nástroja na tvorbu workflow vybraná knižnica JavaFXSmartGraph. Zatiaľ čo yFiles je najvyspelejším nástrojom, ide o proprietárny softvér, ktorého použitie je viazané na kúpu užívateľskej

licencie. Knižnica Jung je síce vytvorená s natívnou podporou frameworku Swing, JavaFX však umožňuje vzájomnú interoperabilitu týchto dvoch nástrojov prostredníctvom na to určeného uzlu scénického grafu `SwingNode`, do ktorého môže byť funkcionalita začlenená [24]. Výsledok po odskúšaní však bol príliš ťažkopádny, graficky nie optimálne prívetivý. Do úvahy prichádzala aj vlastná implementácia grafového problému, tá však bola po prototypovej implementácii zavrhnutá v prospech použitia knižnice JavaFXSmartGraph, u ktorej zavážila hlavne jednoduchosť a užívateľská prívetivosť.

Obrázok 6.5 vyobrazuje implementované grafické rozhranie podľa návrhu predstaveného v kapitole 5.4. Ľavý panel rozhrania sprístupňuje základnú funkcionalitu, t.j. možnosť pridania nového uzlu, po ktorom sa užívateľovi zobrazí ponuka modulov v systéme uložených, ktoré je možné stotožniť s uzlom grafu. Po dvojkliku na uzol sa zobrazia podrobnejšie informácie o module, rovnako ako aj parametre, ktoré daný uzol nesie. Parameter uzla môže prepísať ktorýkoľvek z definovaných parametrov modulu a tým je možné pre danú operáciu prispôsobiť očakávané parametre a ich východiskové hodnoty. Ak má užívateľ zvolený nejaký uzol, je možné prejsť do módu pridania hrany, v ktorom je užívateľovi umožnené dvojklikom na iný uzol vytvoriť medzi nimi závislosť. Pri každom pridaní hrany sa overí, či užívateľ môže takúto zmenu previesť a workflow zostane v konzistentnom stave. V prvom rade je overené, či medzi danými uzlami už hrana nie je, keďže multigraf v kontexte workflow nedáva zmysel. Aplikácia taktiež overí pomocou prehľadávania grafu do hĺbky, či pridaním hrany nevznikne v grafe workflow cyklus. Ak sa tak stane, užívateľ je na tieto javy upozornený formou vyskakovacieho okna.

Ak sa užívateľ rozhodne operáciu uložiť, *Controller* s využitím servisnej vrstvy zabezpečí transparentné uloženie dát do databázy. V prehľade všetkých operácií si užívateľ uložené operácie môže otvoriť a editovať už vytvorené workflow rovnakým spôsobom.

V hornej časti prostredia pre editáciu workflow sú umiestnené dva prvky umožňujúce užívateľovi lepšiu manipuláciu s grafom. Prvým je možnosť výberu medzi automatickým a statickým náhľadom grafu. Štandardne sú uzly grafu automaticky usporiadané, avšak užívateľ môže v prípade potreby túto možnosť deaktivovať. Druhým prvkom pre manipuláciu s grafom je možnosť zmeny veľkosti zobrazenia.



Obr. 6.5: Grafické rozhranie pre modelovanie operácií

6.6 Lokalizácia

Podpora lokalizácie do viacerých jazykov je zabezpečená prostredníctvom súborov typu `.properties`. Každá jazyková mutácia má vlastný `.properties` súbor identifikovateľný názvom prípony, ktorá predstavuje lokalizáciu konkrétneho súboru. V tomto súbore sú lokalizované dáta uložené vo forme kľúč=hodnota, pričom kľúč je priamo použitý na vhodnom mieste v FXML súbore. Aplikácia pri načítaní *View* vloží inštanciu triedy `ResourceBundle`, ktorá je nositeľom lokalizovaných dát, do *Controller*-a pre dané *View* na základe aktuálne zvoleného jazyka v nastaveniach aplikácie. Funkcionalita lokalizácie aplikácie do iných jazykov je následne ľahko rozšíriteľná pridaním vhodného `.properties` súboru s hodnotami kľúčov pre vybraný jazyk.

6.7 Logovanie chýb

Logovanie nečakaných udalostí je zabezpečené za pomoci štandardu SLF4J a jeho implementácie knižnicou Logback. V situáciách, keď

dôjde k neočakávaným chybám, ako napríklad pri načítaní novej obrazovky či chybám pri chýbajúcich atribútoch v konfiguračnom súbore aplikácie pri jej štarte, budú tieto chyby zaznamenané do logovacieho súboru spolu s časom, vláknom, v ktorom chyba nastala a príslušnou správou. Spôsob a miera logov sú definované v `logback.xml` súbore aplikácie.

6.8 Nastavenia aplikácie cez konfiguračný súbor

Aplikácia očakáva pre svoje správne fungovanie korektné údaje v jej konfiguračnom súbore `config.properties`. Ide o dáta súvisiace so správnym nastavením služby Docker a databázy, s ktorými aplikácia pracuje. To, že sú v konfiguračnom súbore všetky potrebné dáta je skontrolované ešte pred spustením aplikácie. Aplikácia sa dáta pokúsi načítať skrze vytvorené rozhranie `PropertiesLoader` a overí, či súbor obsahuje všetky kľúče definované ako konštanty vo výpočtovom type `RequiredProperties`.

6.9 Dizajn

Vďaka možnostiam, ktoré JavaFX poskytuje je možné dekorovať jednotlivé uzly scénického grafu pomocou kaskádových štýlov. Tento prístup umožňuje oddelenie layoutu a grafického prevedenia, ktoré je definované v samostatnom súbore `style.css`. Jednotlivé skupiny uzlov je možno identifikovať pomocou selektorov a vlastných tried, definovaných ako `styleClass` v atribúte uzlu scénického grafu FXML súboru.

JFoenix je knižnica, ktorá do JavaFX prináša prvky material dizajnu. V aplikácií je využitá pri textových poliach s plávajúcim popisom a inline validácií dát, ukazovateľoch progresu a iných grafických prvkoch s cieľom vizuálne prívetivého prostredia.

7 Testovanie

K procesu vývoja softvéru neoddeliteľne patrí aj jeho testovanie. Počas testovania softvéru ide najmä o odhalenie možných chýb vzniknutých v dôsledku ľudského faktora.

Jedným z takýchto nástrojov je jednotkové testovanie. Tento druh automatizovaných testov slúži na izolované testovanie jednotlivých komponentov (funkcii, tried, a pod.). S pomocou testovacieho nástroja JUnit5 a za využitia vstavanej databázy HSQLDB mnou implementované jednotkové testy overujú korektnosť fungovania vrstvy prístupu k databáze.

V ďalšom kroku bolo otestované priame napojenie sa na privátny Docker register projektu vhodným zadefinovaním prístupových údajov v konfiguračnom súbore aplikácie, ako aj funkcionalita manipulácie s obrazmi v ňom, ich zostavenie a následne nahratie. Rovnako úspešne prebehlo napojenie sa na databázu výpočtového komponentu projektu a manipuláciu s definíciami modulov a operácií v ňom uložených.

Aplikácia bola v záverečnej fáze úspešne spustená na operačných systémoch Microsoft Windows a linuxovej distribúcii Ubuntu, čím bola splnená nefunkčná požiadavka portability kladená na vyvíjaný softvér.

Vyvíjaný nástroj bol v priebehu implementácie pravidelne konzultovaný s viacerými členmi tímu ANALYZA a na základe priebežnej spätnej väzby bolo grafické rozhranie upravené po vizuálnej aj funkčnej stránke.

8 Záver

V priebehu tejto bakalárskej práce som sa bližšie pozrel na systémy pre správu workflows, zadefinoval stav výskumného projektu ANALYZA a na základe funkčných a nefunkčných požiadaviek som navrhol a do funkčnej formy implementoval grafické rozhranie pre správu obrazov, modulov a operácií pre popísaný projekt. Implementované grafické prostredie poskytuje možnosť v rámci jedného nástroja zadefinovať všetky potrebné čiastkové kroky s cieľom následného vytvorenia analytického workflow užívateľsky prívetivým spôsobom, a to manipuláciou s grafom, ktorý ho reprezentuje. Užívateľ bude pomocou vyvinutého nástroja schopný zaviesť do prostredia výpočtového komponentu projektu všetky dáta, ktoré sú pre jeho následné použitie očakávané, implementovaný nástroj tak dopĺňa celkovú funkčnosť systému výpočtového komponentu. Výsledkom je funkčná a otestovaná aplikácia, ktorá bola v roku 2021 doručená ako jeden z výsledkov projektu.

Vývoj softvéru je dynamicky proces. V priebehu bakalárskej práce viackrát došlo k zmene požiadaviek vzhľadom k iným komponentom systému a k vývoju softvéru bolo pristúpené agilným, inkrementálnym spôsobom, čo otvára možnosti ďalšieho vývoja a vylepšení implementácie do budúcnosti. Ďalším smerom vývoja by mohlo byť implementovanie autentizácie užívateľov či snaha o testovanie priamo funkcionality kontajnerov nesúcich v rámci projektu vyvinutú funkcionalitu, čo v tejto fáze projektu nebolo podporované ostatnými komponentmi. Ďalším možným smerom vývoja je podpora iných formátov vkladania modulov či operácií (napríklad nahratím a spracovaním súboru vo formáte JSON). Rovnako je stále možné popracovať na grafickej stránke a užívateľskom zážitku a spraviť tak aplikáciu ešte príjemnejšou a atraktívnejšou na použitie.

A Elektronické prílohy

- `db_orchestrator_filler.zip` - implementácia nástroja opísaného v tejto práci
  - zložka `src\main\java\brunomnsilva` obsahuje grafovú knižnicu JavaFXSmartGraph, dostupnú z [23]
  - ostatné súbory obsahujú implementáciu aplikácie popísanú v bakalárskej práci, vrátane `README.txt` súboru, ktorý poskytuje inštrukcie pre jej spustenie.

Bibliografia

1. REBOK, Tomáš et al. *Komplexní analýza a vizualizace heterogenních dat velkého rozsahu (projekt „ANALÝZA“), průběžná zpráva po čtvrtém roce řešení projektu*. Brno: Masarykova univerzita, 2021.
2. REBOK, Tomáš et al. *Komplexní analýza a vizualizace heterogenních dat velkého rozsahu (projekt „ANALÝZA“), průběžná zpráva po třetím roce řešení projektu*. Brno: Masarykova univerzita, 2020.
3. REBOK, Tomáš et al. *Komplexní analýza a vizualizace heterogenních dat velkého rozsahu (projekt „ANALÝZA“), průběžná zpráva po druhém roce řešení projektu*. Brno: Masarykova univerzita, 2019.
4. AALST, Wil van der; HEE, Kees van. *Workflow Management: Models, Methods, and Systems*. The MIT Press, 2004. ISBN 9780262720465.
5. *Luigi* [online]. Github [cit. 2021-04-28]. Dostupné z: `https://github.com/spotify/luigi`.
6. *Use of application containers and workflows for genomic data analysis* [online]. National Center for Biotechnology Information [cit. 2021-04-28]. Dostupné z: `https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5248400/`.
7. *What is a Container?* [Online]. Docker [cit. 2021-04-19]. Dostupné z: `https://www.docker.com/resources/what-container`.
8. *Airflow: a workflow management platform* [online]. Medium [cit. 2021-04-18]. Dostupné z: `https://medium.com/airbnb-engineering/airflow-a-workflow-management-platform-46318b977fd8`.
9. *Concepts* [online]. Apache Airflow [cit. 2021-04-18]. Dostupné z: `https://airflow.apache.org/docs/apache-airflow/stable/concepts.html`.
10. *UI / Screenshots* [online]. Apache Airflow [cit. 2021-05-21]. Dostupné z: `https://airflow.apache.org/docs/apache-airflow/stable/ui.html`.

11. *Argo Workflows - The workflow engine for Kubernetes* [online]. argo [cit. 2021-04-19]. Dostupné z: https://argoproj.github.io/argo-workflows/.

12. *Flow Run* [online]. Prefect [cit. 2021-05-21]. Dostupné z: https://docs.prefect.io/orchestration/ui/flow-run.html.

13. *MLFlow* [online]. mlflow [cit. 2021-04-28]. Dostupné z: https://mlflow.org/.

14. *HTTP API V2* [online]. Docker [cit. 2021-04-28]. Dostupné z: https://docs.docker.com/registry/spec/api/.

15. *Glossary* [online]. Docker [cit. 2021-04-28]. Dostupné z: https://docs.docker.com/glossary/.

16. MARTIN, Robert C. *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Pearson, 2017. ISBN 978-0-13-449416-6.

17. *Overview of ASP.NET Core MVC* [online]. Microsoft [cit. 2021-05-10]. Dostupné z: https://docs.microsoft.com/en-us/aspnet/core/mvc/overview?view=aspnetcore-5.0l.

18. VOS, Johan et al. *Pro JavaFX 9: A Definitive Guide to Building Desktop, Mobile, and Embedded Java Clients*. Apress, 2018. ISBN 978-1-4842-3042-8.

19. *Class Node* [online]. Oracle [cit. 2021-05-10]. Dostupné z: https://docs.oracle.com/javase/8/javafx/api/javafx/scene/Node.html.

20. *Class Task<V>* [online]. Oracle [cit. 2021-05-10]. Dostupné z: https://docs.oracle.com/javafx/2/api/javafx/concurrent/Task.html.

21. *yFiles product details* [online]. yWorks [cit. 2021-05-12]. Dostupné z: https://www.yworks.com/products/yfiles-for-javafx.

22. *The JUNG (Java Universal Network/Graph) Framework* [online]. JUNG [cit. 2021-05-12]. Dostupné z: http://www.datalab.uci.edu/papers/JUNG_tech_report.html.

23. *JavaFXSmartGraph* [online]. Github [cit. 2021-05-12]. Dostupné z: https://github.com/brunomnsilva/JavaFXSmartGraph.

24. *Embedding Swing Content in JavaFX Applications* [online]. Oracle [cit. 2021-05-11]. Dostupné z: `https://docs.oracle.com/javafx/8/embed_swing/jfxpub-embed_swing.htm`.