FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání bakalářské práce

Název:	Anketa ČVUT - reporting
Student:	Lucián Kučera
Vedoucí:	Ing. Michal Valenta, Ph.D.
Studijní program:	Informatika
Obor / specializace:	Webové inženýrství 2021
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2024/2025

Pokyny pro vypracování

Anketa ČVUT pro reporting používá již velmi zastaralou technologii a ani vizuál webových stránek s reporty nevyhovuje. Cílem této práce je analýza, návrh a implementace nového reportingu. Reimplementace se týká jak uživatelského rozhraní, tak také revize, změny a přidání funkcí. Součástí práce je též diskuse a volba implementační platformy.

1. Popište postup tvorby stávajících reportů dostupných na adrese https://anketa.cvut.cz/reports/
2. Shromážděte požadavky na další typy reportů.
3. Navrhněte novou vizuální podobu webu a reportů.
4. Proveďte diskusi a volbu implementační platformy.
5. Návrh implementujte, nasaďte, řádně otestujte a zdokumentujte.

Elektronicky schválil/a Ing. Jaroslav Kuchař, Ph.D. dne 4. prosince 2023 v Praze.

Bakalářská práce

ANKETA ČVUT - REPORTING

Lucián Kučera

Fakulta informačních technologií
Katedra softwarového inženýrství
Vedoucí: Ing. Michal Valenta, Ph.D.
16. května 2024

Obsah

Seznam obrázků

Seznam tabulek

Seznam výpisů kódu

Chtěl bych hlavně poděkovat Ing. Michalu Valentovi, Ph.D., za vedení této bakalářské práce. Děkuji rodičům a bratrovi, kteří mě při této práci podpořili.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací. Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 citovaného zákona.

V Praze dne 16. května 2024

Abstrakt

Tato bakalářská práce řeší tvorbu nového systému zobrazujícího reporty z aplikace Anketa ČVUT, jelikož aktuální řešení bylo vyhodnocené jako nepostačující. V teoretické části práce se zaměřuji na analýzu aktuálně používaného řešení, popis zvolených technologii a také se zaměřuji na vyhledávání v textu na straně klienta a v Oracle databázi.

Výsledkem práce je systém, který zobrazuje potřebná data uživateli ze systému Anketa ČVUT, podporující anglický a český jazyk a je nasazený na školní infrastruktuře. Klientská část byla napsaná v knihovně React v jazyku Typescript a serverová část byla napsaná v frameworku Spring Boot v jazyku Kotlin.

Klíčová slova anketa ČVUT, reporting, webová aplikace, React, Spring Boot

Abstract

This bachelor's thesis solves the creation of the new system displaying reports from Survey CTU system, because the current solution was deemed insufficient. Theoretical part of the thesis contains analysis of the current solution, description of the chosen technologies and describes the problems of searching text in Oracle database and on the client side.

Result of this thesis is a system, that displays data to the user from Survey CTU application, supporting English laguage and Czech language, this system is currently deployed on university infrastructure. The client side was made using React library using programming language Typescript and the server side was made using Spring Boot using Kotlin as programming language.

Keywords survey ČVUT, reporting, web application, React, Spring Boot

Seznam zkratek

ČVUT	České vysoké učení technické
IoC	Inversion of Control
DI	Dependency Injection
HoC	Higer order Component
SQL	Structured Query Language
HQL	Hibernate Query Language
API	Aplication interface
REST	Representational State Transfer
ORM	Object relational mapper
JPA	Java persistance API
XML	Xtensible markup language
AST	Abstract syntax tree
CSS	Cascading style sheets
KSP	Kotlin symbol processing
API	Application interface
JAR	Java archive
HTML	Hyper text markup language
CI/CD	Continuos Integration/Continuos Delivery

Úvod

Reporting se stal běžnou součástí všech organizací, v dnešní době je potřeba analyzovat data, za účelem vylepšení služeb, nebo je třeba informovat o tom, co se aktuálně děje. Taková potřeba vznikla i na ČVUT, pro systém Anketa ČVUT, který má vlastní reporty, ale bohužel jejich forma je aktuálně nevyhovující. Reporty jsou dostupné všem, ale hlavně mají informovat zaměstnance a studenty ČVUT.

Téma jsem si zvolil, protože mě zajímá práce daty, tvorba webových aplikací, byla to skvělá příležitost si vyzkoušet nové technologie a možnost se seznámit s problémem tvorby reportovacího systému. Problém je netriviální a existuje celá řada řešení dané problematiky, ale většinou se jedná o obecné řešení, které nemusí vždy vyhovovat. Také to byla možnost se podílet na projektu pro školu s reálným využitím.

Cílem této práce je navrhnout, implementovat, nasadit a otestovat nový lepší systém pro zobrazování reportů pro Anketu ČVUT.

V teoretické části se budu věnovat aktuálnímu řešení a dalšímu řešení, jež nebylo nasazeno, Také se budu věnovat databázi Ankety ČVUT obzvlášť části určené pro reporty, následně zmíním požadavky na nový systém. dále se budu věnovat problematice vyhledávání v textu a popisu zvolených technologii, které budou využité při tvorbě nového systému.

V praktické části budu popisovat proces návrhu uživatelského rozhraní, jednotlivá rozhodnutí, proč jsem učinil určitá rozhodnutí a důvody proč tato rozhodnutí vedou k lepší uživatelské zkušenosti a lepšímu uživatelskému rozhraní. Potom popíši postup při vývoji aplikace a zdokumentuji způsob nasazení klientské a serverové části v rámci CI/CD pipieline dev-ops nástroje Gitlab na školní infrastrukturu. Následně se budu věnovat zajímavým problémům, které bylo potřeba vyřešit při implementaci mého řešení, a nakonec budu popisovat způsob testovaní a zhodnotím výsledky.

Kapitola 1
Analýza

1.1 Dostupné reportingové nástroje

V reportingu[1] jde o prezentování shromážděných dat, tak aby byly srozumitelné cílové skupině, kde data se nejčastěji reprezentují pomocí tabulek a grafů. Pro tento účel existuje velké množství již existujících nástrojů. V následující kapitole popíšu dva již existující nástroje určené pro reporting.

V oblasti reportingu existuje velké množství řešení, takže není možné pro mě analyzovat všechny možnosti, tak jsem se rozhodl analyzovat dvě populární řešení Apache Superset a Kibana, obě řešení je snadné si rozběhnout na vlastním zařízení pomocí konfiguračního souboru docker-compose.yml.

1.1.1 Apache Superset

Apache Superset[2] patří mezi velice populární řešení dané problematiky. Důvod proč jsem se rozhodl prozkoumat toto řešení je, že se tím aktuálně zabývá jedna diplomová práce, konkrétně Reportovací systém 2.0 pro datový sklad ČVUT.

Vývojová konfigurace systému se skládá z šesti kontejnerů: databáze PostgreSQL, Redis distribuované úložiště klíč/hodnota, samotný systém, inicializační kontejner, celery a celery beat systém pro spuštění periodických činností. Já se budu věnovat jenom samotnému Supersetu.

Celkově systém má skoro dva milióny řádků, když jsem to kontroloval. Klient je napsaný v Reactu a Backend v frameworku Flask-AppBuilder.

Nástroje podporuje velké množství zdrojů dat od většiny SQL databází po distribuované prohledávací engine jako je Elasticsearch 2.3.4 / Apache Solr. Podmínkou pro podporu je existence SQLAlchemy dialektu pro daný systém a driveru, SQLAlchemy je SQL nástroj a ORM napsané v jazyce Python, dialekty povaluji Supersetu se dotazovat nad daty, které nejsou uložené v SQL databázích pomocí jazyka SQL.

Datové zdroje systému musí být v denormalizované formě jedné tabulky, nelze propojovat tabulky pomocí SQL joinů v rámci Supersetu. Lze použít například pohled, nebo materializovaný pohled, pokud potřebujeme data z více tabulek.

Systém podporuje správu vizualizací, správu dashboardů a správu datových zdrojů a prohledávání datových zdrojů pomocí SQL editoru, také má velkou možnost konfigurace přístupových práv.

1.1.2 Kibana

Důvod analýzy Kibany[3] je, že jsem se s nástrojem již setkal ve škole v předmětu BI-BIG (Big data), kde jsem v něm vypracoval semestrální práci. Kibana je vizualizační nástroj podobně jako Superset, ale na rozdíl od něj se specializuje jenom na Elasticsearch 2.3.4, také množství vizualizací je značně omezenější, ale to se mírně kompenzuje využití gramatiky Vega, určené pro tvorbu grafů, ale z mých zkušeností to bylo velice náročné na použití. Výhodou je snadnější tvorba základních vizualizací pomoci nástroje Lens, který je jednou z nabízených možností z mého pohledu, je snadnější na použiti než rozhraní nabízené Supersetem.

Kibana je hlavně zaměřená na zobrazování dat v určitém časovém rozsahu, na rozdíl od Supersetu. Také obsahuje velké množství způsobů dotazování, například KibanaQueryLanguage, dotazování se pomocí Rest Api a nového jazyku ES—Ql.

V našem případě potřebujeme zpracovávat data z databáze Oracle a můžeme jednotlivé požadavky přeposílat z databáze Ankety do Elasticsearch pomocí nějakého programu.

1.1.3 Shrnutí aktuálních řešení

Aktuální řešení, která se zabývají problémem datové vizualizace a reportování, existuje velké množství, dokonce dalo by se říct nepřeberné množství od webových aplikací po klasické desktopové aplikace, jako Power BI a další podobné nástroje. Již zmíněné nástroje mají za sebou roky vývoje a pravděpodobně milióny hodin od daleko zkušenější vývojářů, než jsem já, takže nemá smysl vyvíjet podobné nástroje, jelikož já se jim nemam šanci v rámci této práce vyrovnat. Proto v této práci se pokusím zaměřit na věci, které v nástrojích chybí.

Jedna z věcí, která mi chyběla při vypracování semestrální práce v rámci předmětu BI-BIG je lepší podpora formátování dat získaných z databázi. Pokud bych něco seskupil podle id fakulty, tak by to zobrazovalo id, místo užitečnějších informací pochopitelnějších uživateli, také data z databáze nebylo možné překládat. Některé vyspělejší komerční nástroje podporují různé formy překladu, ale osobně jsem si je nevyzkoušel, takže se nemohu na toto téma vyjádřit.

Nástroj Kibana, má omezenější tvorbu vizualizací, například nelze používat kategorická data na x-ové ose čárových grafů.

Superset má daleko větší možnosti a podporuje vše, co bych potřeboval, až na možnost překládání.

Na závěr shrnutí existujících řešení bych chtěl říci, že moje práce se se zabývá tvorbou nového systému, a proto nevyužiji existující řešení.

1.2 Aktuální stav

1.2.1 Anketa reporty v1

Aktuální verze je dostupná zde [4]. Aktuálně se reporty vytvářejí pomocí skriptu v databázi, generujícího HTML stránky. Stránky se předají webovému serveru a ten je zpřístupní pro uživatele. Skript je spuštěný periodicky v určeném časovém intervalu.

Aktuálně je generovaná jedna HTML stránka pro každý report. Reporty se generují pro fakulty jednotlivě a jenom pro aktuální semestr. Každý sloupec má vlastní textový vstup, kde uživatel může zadat dotaz. K filtrování dat z reportu každá stránka obsahuje funkci napsanou v jazyce Javascript, funkce se volá při změně dotazu a kontroluje, zda řádky v daném sloupci mají hodnotu, která je obsažená v textovém poli. Filtrování funguje na principu, že řádky musí splňovat všechny dotazy najednou.

Nevýhodou aktuálního řešení je omezení na filtrování na straně klienta, filtrování je velice jednoduché a má omezenou vyjadřovací schopnost, také podporuje jenom textové vyhledávání, nelze vyhledat například rozsahem. Další problém je rozhraní, které není uživatelsky nejpřívětivější,

například ze stránky reportu se nelze vrátit na stránku s výběrem při využití navigačních prvků aplikace. Samotná aplikace také obsahuje malé množství funkcionalit, například aplikace nepodporuje více jazyků a další funkcionality, o kterých se zmíním v kapitole s funkčními a nefunkčními požadavky 1.4.

Aktuální verze obsahuje pět reportů. První report zobrazuje informace o předmětech v anketě, druhý report obsahuje informace o vyučujících a předmětech, třetí report informuje o vyplnění předmětů, čtvrtý report zprostředkovává informace o anketních otázkách a poslední report obsahuje stejné informace jako report o vyučujících, pouze je prohozené pořadí sloupců. Také na konci každé stránky se vyskytuje seznam předmětů, které nebyly zahrnuté do ankety.

1.2.2 Anketa reporty v2

Anketa Reports druhá verze má snahu vylepšit aktuální stav aplikace. Klientská část [5] je napsaná v frameworku Nuxt.js a serverová část [6] je napsaná v frameworku Spring Boot. Serverová část aplikace je velice jednoduchá, pouze filtruje pohledy podle klíče ankety a semestru. Klientská část je také velice jednoduchá, ale na straně klienta se vyskytují chyby, které má také aktuální verze, například filtrování se provádí na straně klienta. Zhoršila se vyjadřovací schopnost uživatele, jelikož prohledávací pole je jen jedno a prohledává jen některé vybrané sloupce. Dále samotná architektura řešení na straně klienta je chybná, protože se používají natvrdo napsané url a to znemožňuje aplikaci vyvíjet na lokálním prostředí a nasadit ji do jiného prostředí při využití stejného kódu jenom se změnou konfigurace. K nasazení této verze nedošlo, jelikož nebyla dokončena.

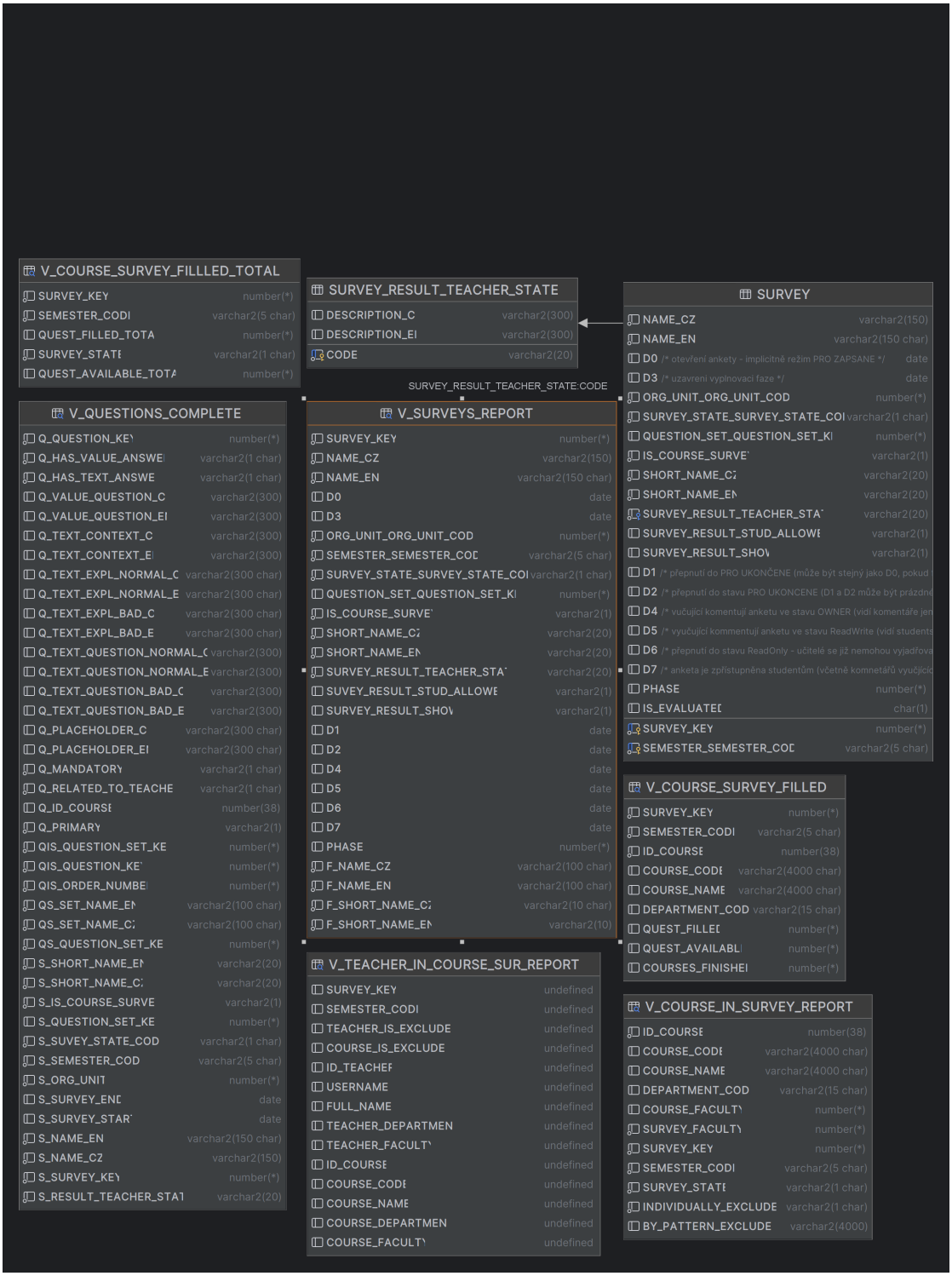
1.3 Analýza databáze ankety

Nejdůležitější částí této práce jsou data, která jsou uložená v databázi. Aplikace získává data z SQL databáze Oracle [7] používané systémem Anketa ČVUT. Osobně jsem se s Oracle databází před tím nesetkal, tak jsem se musel doučit, jak funguje.

Databáze ankety je velice komplexní, aktuální řešení využívá dvě schémata: ANK_ANKETA a NOVA_ANKETA. Celkem mezi těmi schématy je přes šest-set materializovaných pohledů, přes tisíc tabulek, pár desítek pohledů, několik desítek funkcí a dále je databáze propojená s Usermap databází a databází KOSu, z samotného výčtu objektů v databázi týkajících se ankety je zřejmé, že se jedná o velice komplexní projekt. Samotný projekt Anketa je ve vývoji jíž několik let, tak je pro mě nemožné popsat vše, co se jej týká a zanalyzovat celou databázi, proto provedu jen analýzu části databáze týkající se mojí aplikace.

Na obrázku 1.1 je vidět databázové schéma systému pro reporting v2. Aplikace používá šest pohledů a 1 tabulku. Pohledy jsou využívané jednotlivými reporty.

1. **V_COURSE_SURVEY_FILLED_TOTAL:** má informace o vyplnění anket, má informace o počtu dostupných lístků a počet vyplněných lístků.
2. **V_COURSE_SURVEY_FILLED:** obsahuje informace o vyplnění ankety, ale data se týkají předmětů na rozdíl od V_COURSE_SURVEY_FILLED_TOTAL.
3. **V_QUESTIONS_COMPLETE:** účelem pohledu je zobrazit informace o otázkách v anketě.
4. **V_SURVEYS_REPORT:** informuje o anketách a obsahuje čas událostí ankety.
5. **V_TEACHER_IN_COURSE_SUR_REPORT:** propojuje předměty v anketě s vyučujícími, kteří daný předmět vyučujíc.
6. **V_COURSE_IN_SURVEY_REPORT:** zprostředkovává informace o předmětech v anketách.



Obrázek 1.1 Diagram verze v2. Získán z Datagrip

7. **SURVEY:** tabulka obsahující veškeré informace o anketě, verze V2 ji využívá k získání semestru.

1.4 Požadavky

Požadavky jsem shromáždil v rámci analýzy předchozích systémů a v rámci konzultací s vedoucím práce, který systém používá a je správcem ekosystému Anketa ČVUT.

1.4.1 Funkční požadavky

1.4.1.1 F1: Zobrazení reportů

Hlavním funkčním požadavkem je možnost pro uživatele zobrazit si reporty s daty ankety.

1.4.1.2 F2: Předání filtrů pomocí URL

Aplikace má zprostředkovat parametry vyhledávání v URL, aby si uživatelé mezi sebou, pokud nastane potřeba, mohli sdílet filtry.

1.4.1.3 F3: Filtrování

Uživatel bude moci filtrovat jednotlivé reporty, podle specifikovaných polí, vhodným způsobem podle typu dat.

1.4.1.4 F4: Stránkování dat

Uživatel si může zvolit, jestli chce data stránkovat, nebo jestli data nechce stránkovat za cenu, že se data budou načítat delší dobu. Uživatel si může zvolit stránku, kterou si zobrazí.

1.4.1.5 F5: Export

Aktuální i druhá verze nepodporují export dat do běžných datových formátů. Ze vzhledu, že aplikace slouží hlavně k prohlížení dat, tak by bylo dobré mít možnost data exportovat do běžných datových formátů jako je CSV a JSON.

1.4.2 Nefunkční požadavky

1.4.2.1 N1: využití databáze

Aktuálně aplikace provádí filtrování na straně klienta, což je práce, která by měla být ponechaná databázi vzhledem k tomu, že jeden z následujících požadavků je stránkování, tak je potřeba, aby se data filtrovala v databázi. Také databáze je přizpůsobená k vyhledávání a většinou to dokáže vykonat rychleji, než skript napsaný v jazyce Javascript.

1.4.2.2 N2: Persistence filtrů

Filtry jednotlivých reportů by se měly zachovat i když uživatel zavře prohlížeč, k uložení lze například využít jedno z úložišť v prohlížeči.

1.4.2.3 N3: Jednoduchost

Aplikace nemá být komplikovaná na použití a má mít intuitivní rozhraní pro uživatele.

1.4.2.4 N4: Nasazení

Aplikaci je potřeba nasadit na školní infrastrukturu, aby nahradila aktuální verzi a mohla být reálně využita.

1.4.2.5 N5: Zlepšení vzhledu aktuálního systému

První i druhá verze reportingu má již zastaralý a nepěkný vzhled, proto je potřeba jej aktualizovat.

1.4.2.6 N6: Výběr jazyka

První verze, která je aktuálně nasazená, nepodporuje jiný jazyk než češtinu. Vzhledem k tomu, že škola má i program vyučovaný v angličtině, je potřeba do nové verze přidat anglický jazyk.

1.4.2.7 N7: Režim

Aktuálně aplikace podporuje jen světlý režim a nepodporuje tmavý režim, v dnešní době je tmavý režim pro webové aplikace standardem, který by se měl dodržet.

1.4.2.8 N8: Responsivita

V dnešní době většina uživatel prohlíží web z mobilního zařízení, proto by měla aplikace být navržena, aby podporovala různé velikosti obrazovek.

1.4.2.9 N9: Rozšiřitelnost

Aplikace by měla být dobře zdokumentovaná a řádně otestovaná, aby bylo možné ji rozšířit v budoucnosti.

1.4.2.10 N10: Kotlin

Aktuálně probíhá/proběhl refactroing backendu Ankety ČVUT do jazyka Kotlin z využitím Spring Boot frameworku. Vedoucí projektu Anketa rozhodl, že Kotlin bude jednotným jazykem pro projekty napsané pro Anketu.

1.4.2.11 N11: React

Stejně jako v případě backendu, tak rozhodl vedoucí projektu Anketa, že klientské části budou psané při využití knihovny React, aby bylo vše napsané při využití jednotných technologii.

1.5 Případy užití

1.5.1 Aktér - uživatel

Aplikace má jediného aktéra a tím je nepřihlášený uživatel. Aplikace slouží veřejnosti pro zobrazení dat z Ankety ČVUT.

1.5.2 Případy užití

1.5.2.1 UC1: Zobrazení reportu

První případ užití je zobrazení stránky reportu.

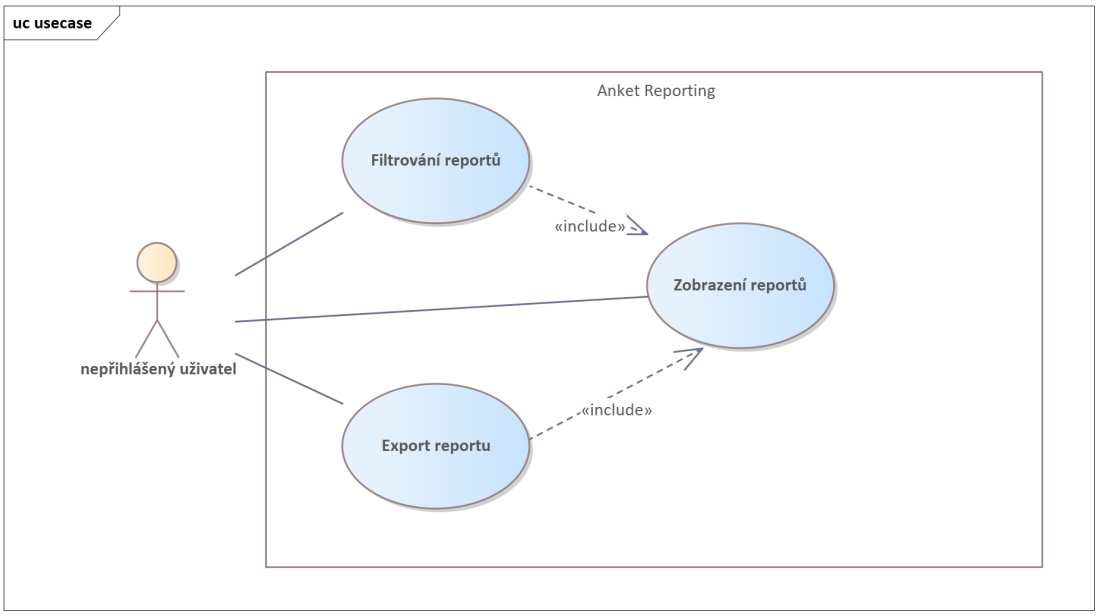
1. Uživatel si vybere v navigaci, jaký report si zobrazí.
2. Následně uživatel klikne na odkaz příslušející danému reportu.

1.5.2.2 UC2: Filtrování reportu

1. Nejdříve si uživatel zobrazí stránku reportu 1.5.2.1.
2. Uživatel vyplní filtrovací pole v rámci formuláře, aplikace bude reagovat na změnu filtru a automaticky načte data.

1.5.2.3 UC3: Export

1. Nejdříve si uživatel zobrazí stránku reportu 1.5.2.1.
2. Uživatel si otevře formulář určený pro export.
3. Následně uživatel formulář vyplní.
4. Nakonec uživatel stiskne tlačítko pro stažení dat.



Obrázek 1.2 Diagram případů užití

Kapitola 2
Návrh

2.1 Architektura

Aplikace využívá architekturu [8]klient–server[9]. Klient posílá požadavky na server a server vrací klientovi odpovědi. Já používám tříúrovňovou architekturu, kde klient je prezentační vrstva, aplikační server je aplikační vrstva a databáze je datová vrstva. Samotný aplikační server používá třívrstvou architekturu, kde kontrolery (třídy přijímající požadavky a vracející odpovědi) jsou prezentační vrstvou, servisní třídy tvoří aplikační vrstvu a repository třídy tvoří datovou vrstvu.

2.1.1 SPA

SPA [8] česky „aplikace s jednou stránkou“ je moderní přístup pro psaní uživatelského rozhraní. SPA na rozdíl od tradičních MPA aplikací, načte jenom jednu HTML stránku, CSS styly a Javascript, který generuje HTML za běhu. Při tvorbě aplikace je třeba rozhodnout, kterou variantu zvolit. Je třeba uvažovat výhody a nevýhody přístupů, ale i to nemusí být rozhodující, tak je třeba zvážit jaké technologie vývojáři preferují.

Mezi výhody SPA patří skvělé technologie vytvořené komunitou pro jejich vývoj, například React. Další výhodou je, že se obsah načítá jen jednou, poté se již získávají jen data, to umožňuje přívětivou uživatelskou zkušenost, jelikož uživatel má pocit, že aplikace reaguje na jeho vstup jako obvyklá nativní aplikace.

Mezi nevýhody SPA patří pomalejší první načtení dat, ale to většinou není až tak citelné. Další nevýhodou je špatná SEO, ale v mém případě SEO nepotřebují, neboť vytvářím aplikaci pro školu, která nemá komerční účely.

2.1.2 REST API

REST [8] je standard pro psaní rozhraní nabízející zdroje. Server v rámci mojí práce slouží pouze jako REST API, nabízející data ve formátu JSON. Takový přístup jsme v rámci předmětu TWA nazvali „dumb server“.

Hlavní důvodem vzniku REST API je standardizace rozhraní webových aplikací, url se jmenují podle zprostředkovávaného zdroje, například url zpřístupňující data o uživatelích by měla tvar

/users a url pro konkretního uživatel by měla tvar

/user/id. Také standart definuje, jaká HTTP metoda se má použít a jaký HTTP status se má vrátit.

2.2 Způsob vyhledávání

Důležité je rozhodnout, jak se budou data filtrovat, jakým způsobem. Například jestli se všechny sloupce nesloučí do jednoho a budou považované za text, který potom bude prohledaný klíčovými slovy, nebo bude záležet na typu sloupce a tak podobně.

Aplikace má být hlavně jednoduchá na použití a má umožnit uživateli snadno vyhledat data, která potřebuje. Problém je takový, že tento úkol není jednoduchý, aby šlo snadno a dobře vyhledávat přes velké množství sloupců, obzvlášť když máme větší množství dat a chceme, aby pro naše potřeby byla aplikace relativně rychlá.

V této sekci popíši všechny možné způsoby, vyhledávání, které jsem zkusil a popíši proč jsem se rozhodl je nepoužít, nebo naopak použít. Také bych rád zmínil pár užitečných termínů, využívaných při vyhledávání ať už v databázích, tak v datech obecně.

2.2.1 Levenshteinova editační vzdálenost

Levenshteinova editační vzdálenost je vzdáleností dvou textových řetězců. Vzdálenost uvažuje tři operace, které lze provést nad řetězcem: změnit znak na libovolný jiný, odstranit znak a přidat libovolný znak. Algoritmus lze řešit dynamickým programováním s asymptotickou složitostí $\mathcal{O}(nm)$. Algoritmus se využívá při vyhledávání v textech, umožňuje hodnotit, jak jsou si texty podobné, ale samotná vzdálenost není bezchybná, jelikož delší texty jsou v nevýhodě, i když by například delší text obsahoval celý dotaz a jiný kratší text by neobsahoval nic z dotazu. Hlavně se Levenshteinova vzdálenost využívá při fuzzy/přibližném vyhledávání, kdy se povoluje, že dotaz nemá přesnou shodu s vyhledávanými texty. Levensthteinovu vzdálenost například implementuje nástroj Oracle [10], nebo Elasticsearch [11] při fuzzy vyhledávání.

2.2.2 Index v databázi

Index je termín obecně využívaný v oblasti dat s významem struktury urychlující vyhledávání v datech. Indexy jsou různých druhů, nejčastěji se využívají B-Stromy. Díky indexu lze rychle řadit data, protože často samotné indexy obsahují data již seřazená, také lze vykonávat rychlá rozsahová hledání, prefixové textové hledání a spousta dalších operací je využívá k zrychlení.

Z předchozího odstavce se může zdát, že indexy jsou jenom výhodné a bez nevýhod, tomu tak však není, indexy je potřeba udržovat při změně dat, to může výrazně zpomalit úpravu dat a přidávání nových dat, také zabírají netriviální množství paměti na disku [12].

2.2.3 Invertovaný index

Obecně se v SQL databázích datové struktury určené k rychlému vyhledávání nazývají indexy, invertovaný index slouží také k vyhledávání.

Invertovaný index [13] je základní datovou strukturou určenou pro vyhledávání v textu. Databáze Oracle jej využívá pro textové indexování. Invertovaný index funguje na principu, kde každé slovo odkazuje na dokumenty, ve kterých se nachází. Potom se vyhledávají takové dokumenty, které splňují výraz použitý pro hledání. Invertovaný index využívá vlastnosti, že jednotlivá slova jsou obsažena jenom v malé podmnožině dokumentů, pokud by slova byla obsažená ve většině dokumentů, tak vyhledání je nucené procházet velké množství dat, proto se používají seznamy zakázaných slov vyskytujících se často v textech, aby takový problém nenastal. Samozřejmé můžeme chtít, abychom hledali i podle takových slov a seznam zakázaných slov nezavedeme, i bez toho opatření je invertovaný index rychlejší, než kdybychom vyhledávali bez něj, protože oddělí slova od sebe a tím pádem lze využívat na ně databázové indexy jako je b-strom index, bez použití invertovaného indexu by nešlo vyhledávat efektivně podle slova uprostřed věty.

2.2.4 Exaktní a přibližné hledání

V této kapitole bych se rád zmínil o mých vlastních pozorováních ohledně exaktního a přibližného hledání, co považuji za jejich pro a proti. Důležitým rozhodnutím je, jestli aplikace vyžaduje exaktní vyhledávání, nebo přibližné, nebo dokonce vyžaduje kombinaci způsobů.

Přibližné vyhledávání má výhodu, že uživatel nepotřebuje znát data a muže se pokusit najít něco, co by ho zajímalo, i když neví, jestli něco takového se v databázi nachází. Zásadní nevýhoda je, že se nám mohou vrátit nežádaná data. Také má přibližné vyhledávání častokrát složitější implementaci, než jenom porovnání rovnosti. Pro přibližné vyhledávání lze využít již zmíněny invertovaný index 2.2.3, nebo Levenshteinovu vzdálenost 2.2.1.

Exaktní vyhledávání je jednoduché na implementaci. Porovnají se jedna strana s druhou, jestli jsou stejné. Další výhodou je, že vždy dostaneme to, co chceme a nic navíc. Nevýhodné je, pokud chceme vyhledat například podle emailu nebo jména, tak je potřeba celé napsat, což není nejpřívětivější z pohledu uživatele.

Kombinací přístupů zmíněných v předchozích odstavcích nazývám vyhledávání s napovídáním. Vyhledávání s napovídáním je v dnešní době velice populární, právě všechny textové editory napovídají programátorům názvy proměnných, nebo syntaxi jazyka. Ve svojí aplikaci bych rád využil tento přístup, protože se mi zdá nejvhodnější pro aplikaci, kterou chci vytvořit a zdá se mi velice uživatelsky přívětivé.

2.3 Technologie určené k vyhledávání

Důležitou součásti aplikace je vyhledávání v datech. V této kapitole popíši technologie využívané za tímto účelem.

2.3.1 Vyhledávání v Oracle

Jelikož databáze, kterou projekt bude určitě využívat je Oracle, tak jsem se zaměřil při průzkumu způsobu vyhledávání hlavně na ní. Jelikož je Oracle SQL databáze, tak některé z následujících metod lze aplikovat obecně, ale také jsem se zaměřil na metody, které jsou specifické pro ni.

2.3.1.1 Specifické metody Oracle databáze

Vyhledávání v textu je implementované v databázi Oracle v balíčku Oracle Text [14]. Oracle text nabízí velké množství způsobů vyhledávání v textu, podporuje různé formáty dat, umožňuje prohledávat v rámci více sloupců, a dokonce i v rámci více tabulek, zmíním se o těchto funkcionalitách v detailu dále. Dále ve verzi 23c Oracle byla přidána nová knihovna, která zlepšuje podporu prohledávání v rámci celé databáze. Tuto knihovnu bohužel nelze využít, jelikož Anketa ČVUT používá verzi 19c, ale i tak se o ní zmíním, kdyby se v budoucnu provedla migrace na novější verzi databáze.

Oractle text nabízí několik druhů indexů. `CONTEXT` a `SEARCH` index je určený pro indexovaní textových dokumentů, je nutné je udržovat operací na synchronizaci indexu. Hlavní index, který jsem se rozhodl použít je `CTXCAT`, jenž funguje dobře na kratší texty a je transakční, to znamená, že se upravuje při operacích `delete`, `update` a `insert` na zdrojovou tabulku. Poslední je `CTXRULE` určený pro klasifikaci textu.

V následujících odstavcích zmíním důležité termíny týkající se balíčku Oracle text. Nejdříve se zmíním o termínu datastore, jenž definuje zdroj dat určeny k prohledání.

Základní datastore je sloupec, v tomto případě se index získává data přímo z něj. Výhoda je, že index je jednoduchý na údržbu, jelikož v tomto případě se o něj dokáže Oracle sám postarat.

Následující Datastore zprostředkující vyhledávání přes více sloupců
je `MULTI_COLUMN_DATASTORE`, umožňující získání dat z více sloupců v rámci jedné tabulky. Výhoda je, že nejsme omezení na jeden sloupec. Můžeme klidně prohledat celou tabulku, ale pokud se

data upraví na jiném sloupci, než je indexovaný, tak je potřeba definovat proceduru, která nám index synchronizuje, buď v rámci triggeru(události), nebo v rámci periodické události.

Poslední Datastore co zmíním, je `USER_DATASTORE`. Index využívá proceduru jako zdroj dat, která má pevně definované rozhraní, vrátí CLOB (Character large object) pro každý řádek a v něm se vyhledává. Oproti `MULTI_COLUMN_DATASTORE` jej lze využít na prohledávání více tabulek, ale je složitější na synchronizaci, složitost synchronizace závisí na definici indexu, z kolika tabulek získává data.

Teď bych vysvětlil další důležitý termín, a to je lexer, jehož úkol je rozdělit text na jednotlivé termíny. Oracle nabízí několik lexerů, zásadním důvodem pro jejích nabídku je podpora více jazyků, například čínština má jinou implementaci než angličtina.

Dále lze v Oracle text definovat ignorovaná slova, jak bylo zmíněno v kapitole 2.2.3, jak má být index uložen a další různá nastavení pro dokumenty různých formátů, jako jsou HTML dokumenty nebo docx dokumenty.

Rád bych zmínil balíček `DBMS_SEARCH`[15], který usnadňuje prohledávání textu v rámci celé databáze, protože za nás vytvoří triggery, jenž synchronizují datovou strukturu udržující si data. Jeho hlavní výhodou je snadnost použití, balíček za nás všechno složité vykoná. Možnou výhodou nebo nevýhodou, záleží na úhlu pohledu, je, že data je třeba prohledávat ve speciální tabulce, v níž jsou uložená data v JSON sloupci. Pokud chceme získat data přímo z tabulky, kterou jsme určili pro prohledávání, tak je potřeba napojit výsledky vyhledávání identifikátorem.

Vyhledávaní se provede pomocí operátoru `Contains` [16], operátor má jako argument indexovaný sloupec a textově napsaný dotaz. Dotaz se skládá z hledaných frází, oddělitelných standardními logickými operátory jako je `AND`, `OR` a `NOT`. Na samotné fráze lze aplikovat funkce a celý výraz je možné libovolně uzávorkovat.

2.3.2 Operátor Like

Narodil od komplikovaného přístupu nabízeném balíčkem Oracle Text, tak `Like` [17] operátor je velice jednoduchý na údržbu a využití, ale jeho schopnosti jsou značně omezenější. Operátor `Like` nám umožňuje vyhledávat přesné výrazy, prefixové vyhledávání, postfixové vyhledávání a zda text obsahuje daný řetězec. Pokud se využijí logické spojky jazyka SQL, tak lze dosáhnout podobných výsledků jako u předchozího řešení, alespoň co se týče textového vyhledávání. Zásadním problémem toho řešení je výkon, protože ne vždy se bude využívat index, pokud nevíme, co děláme, a index nám zaručuje v SQL databázích rychlé vyhledávání, bez indexu pro větší množství dat ztrácíme rychlost. Prefixové vyhledávání a postfixové lze nastavit, aby využívalo index, ale obsahové vyhledávání nelze, a to může být kamenem úrazu naší aplikace, ale jak již bylo zmíněno, řešení je značně jednodušší na implementaci.

2.3.3 Fuse.js

Další technologii, kterou bych rád zmínil je Fuse.js [18], protože vyhledávání není jenom problém na stráně serveru, ale i na straně klienta, tak je potřeba využít další technologie. Fuse.js mi umožní v rámci aplikace zlepšit uživatelskou zkušenost z vyhledáváním. Fuse.js využívá aproximační vyhledávání, takové vyhledávání se nazývá fuzzy, které nám najde nejlepší možnosti. Důvod pro vyhledávání na straně klienta je, že občas máme určitá data na straně klienta, které je možné prohledat, například máme velké množství stránek a potřebujeme je prohledat, nebo máme vstupy, které nabízejí nápovědy. V mojí aplikace Fuse.js využívám například při výběru sloupců, při výběru fakulty nebo úvodní výběr samotného reportu.

2.3.4 Elasticsearch

Elasticsearch [19] je distribuovaný vyhledávací engine, jak je napsáno na jejich stránkách, dalo by se jej považovat za NoSQL databázi. S nástrojem jsem si již setkal v rámci předmětu BI-BIG (big data), kde na jeho využití byla semestrální práce. Elasticsearch je napsaný v jazyce Java a na pozadí využívá Apache Lucene, což je vyhledávací engine. Nástroj samotný je přizpůsobený k rychlému vyhledávání dat a spoustě dalších účelů. Výhodou je rychlé vyhledávání a automatické indexování dat, které musíte provádět manuálně v rámci SQL databází. Nevýhoda je, že data je potřeba denormalizovat, jelikož podpora pro joiny není příliš velká. Také je potřeba použít další nástroj Logstash. Logstash slouží k sběru dat, následnému zpracování a konečnému odeslání zpracovaných dat do cíle. Zdroj může být například soubor. Filtr například zpracuje CSV data. Nejobvyklejší cíl je Elasticsearch.

K nástroji existuje velké množství alternativ zabývajících se problematikou vyhledávání, stejně jako Elasticsearch, například: Opensearch fork Elasticsearche, nebo v dnešní době často vidím nástroj Algolia, jenž využívá například dokumentace Spring Frameworku pro indexování dokumentace, ale poslední zmíněný nástroj je komerční a já v rámci vlastních projektů využívám pouze open source nástroje.

S vedoucím jsem se dohodl na tom, že Elasticsearch se nevyužije, protože dat není tak velké množství, abychom nástroj upotřebili a samotný Oracle nám nabízí vyhledávání v textu, jehož rozsah stačí pro naše potřeby, ale pokud by bylo zapotřebí, tak nevylučuji, že by bylo možné v budoucnu Elasticsearch využit.

2.4 Technologie na straně klienta

2.4.1 Typescript

Typescript [20] je nadstavba jazyka Javascript, zpříjemňující vývoj a zvyšující udržitelnost kódu tím, že do jazyka přidává podporu pro typování. Díky typům může editor lépe napovídat a lepe kontrolovat chyby, tyto dvě výhody samotné z mých zkušeností mi značně zlepšují produktivitu.

Také bez Typescriptu je velice náročné spolupracovat v týmu na projektu bez toho, aby typy byly dokumentované někde stranou. V dnešní době se čistý Javascript používá obvykle na starších projektech.

2.4.2 React

React [21] je knihovna vytvořená Metou (dříve Facebook), která revolucionizovala tvorbu webových aplikací, aktuálně je nejpoužívanější ze všech na trhu. Základem tvorby rozhraní je členění na komponenty, ze kterých se skládají složitější komponenty a z nich ještě složitější, až konečně samotná stránka.

V Reactu lze komponentu definovat buď jako třídu, nebo jako funkci, v dnešní době se doporučuje využívat funkcionální přístup, který je aktuálně rozvíjený. Pouze v starých projektech se může člověk setkat s třídami.

Komponenty Reactu využívají rozšíření jazyka nazývané JSX v případě Javascriptu a TSX v případě Typescriptu, přidávající možnost psát do souboru podobný syntax jako HTML, což se později promění na javascript/typescript.

Výpis kódu 2.1 Ukázka komponentu v Reactu

```
function InputPlain({ className, ...rest }: InputProps) {
    return (
        <input className={twMerge(
            "leading-5 pl-2 py-2 bg-font-100 outline-1",
            "outline-font-800 outline focus:outline-2 ",
            "dark:bg-font-800 dark:text-font-100 ",
            "dark:outline-font-700 dark:[color-scheme:dark]",
            className
        )} {...rest} />
    )
}
export default InputPlain;
```

2.4.2.1 Reconciliation

Nejdříve je potřeba vědět, že html dokument je v podstatě strom, značka `html` určuje kořen dokumentu. Tomuto rozhraní, kdy se přistupuje k HTML dokumentu jako stromu se říká DOM (dokument object model). React neupravuje přímo DOM [22], ale udržuje si v paměti Virtual-DOM, na kterém provádí nejprve změny. Procesu, kdy se synchronizuje Virtuální DOM s reálným se nazývá Reconciliation [23].

Díky virtualizaci DOMU může React být použit i mimo web, například existuje knihovna React Native, podporující vývoj na desktop, android i iOS.

2.4.2.2 Životní cyklus

Životní cyklus [24] komponentu se skládá ze tří událostí:

1. **mount:** když se komponent objeví na obrazovce.
2. **update:** když se změní stav komponentu, nebo jsou komponentu předaná jiná data.
3. **unmount:** když je komponent odstraněn z obrazovky.

2.4.2.3 Higher-Order Component pattern

Higher-order components [25], zkráceně HOC je návrhový vzor využívaný při tvorbě aplikace v Reactu. Jeho primárním využitím je obohatit jiný komponent o často využívanou logiku v aplikaci. Například máme stránky a všechny mají sidebar a navigaci nahoře stránky, tak můžeme vytvořit HOC a poté zabalit stránky do něj. V dnešní době se hlavně používá na obalení komponent další logikou vyžadující JSX, pro logiku samotnou se používají custom hooks 2.4.2.4.

Výpis kódu 2.2 Ukázka HOC komponentu v Reactu

```
function withReport<T extends JSX.IntrinsicAttributes>(
    Component: React.FC<T>
) {

    return function ComponentWithReport(props: T) {
        return (
            <ReportContextProvider>
                <Component {...props} />
            </ReportContextProvider>
        )
    }
}
```

2.4.2.4 Hook pattern

Hook je vzor používaný v Reactu, který umožňuje funkcím přístup ke stavu aplikace. Slouží zejména k vytvoření znovupoužitelné logiky, kterou mohou využívat další hooky a komponenty. Hook musí být prefixován „use“ a poté musí následovat velké písmeno.

Výpis kódu 2.3 Ukázka hooku v Reactu

```
import { RefObject, useEffect } from "react";

function useOutOfBounds(
{ target, current, close }:
{
    target: EventTarget | null,
    current: RefObject<HTMLElement>, close: () => void
}) {
    useEffect(() => {
        if (target && current.current) {
            if (!current.current.contains(target as Node)) {
                close()
            }
        }
    }, [target, close, current]);
}

export default useOutOfBounds
```

React [21] sám o sobě již má několik hooků implementovaných. Zde vám představím ty nejdůležitější.

Nejzákladnější hook je `useState`, sloužící k přechovávání stavu aplikace. Vrací dvě hodnoty: ukazatel na aktuální stav a funkci určenou k modifikaci stavu. Důležité je vědět, že modifikace stavu je asynchronní, což znamená, že se stav nemění rovnou pří zavolání funkce ke změně stavu.

Dále je potřeba nemodifikovat stav přímo, protože React kontroluje referenci při změně, takže by docházelo ke změně stavu bez překreslení.

Dalším důležitým hookem je `useEffect`. `useEffect` nám umožňuje reagovat na události životního cyklu aplikace 2.4.2.2, má dva argumenty funkci (callback) a pole závislostí. Funkce je volaná při události `mount` a poté vždy při detekci změny závislostí, návratová hodnota je buď nic, nebo další funkce, volaná při události `unmount`.

Následně chci zmínit další zajímavý hook, který slouží pro předávání stavu v hlubokém stromu komponent. `useContext` má přístup k informacím zpřístupněným React kontexty, které jsou definované nad komponentou volající hook ve stromové struktuře.

2.4.3 TailwindCSS

Pro pochopení TailwindCSS [26] je nejdříve potřeba vysvětlit jak funguje CSS.

2.4.3.1 CSS

CSS (Cascading Style Sheets) [27] je jazyk určený pro stylování HTML dokumentů. Pravidlo CSS se skládá z deklarací. Deklarace se skládá z vlastnosti a hodnoty.

Výpis kódu 2.4 Ukázka CSS deklarace

```
1     box-sizing: border-box;
```

Pravidlo samotné se skládá ze selektoru a bloku deklarací. Selektor určuje, na jaké elementy se blok deklarací aplikuje a blok deklarací určuje, jaké styly se aplikují na vybrané elementy.

Výpis kódu 2.5 Ukázka CSS pravidla

```
* {
  margin: 0;
  padding: 0;
  box-sizing: border-box;
}
```

Samotné styly jsou buď definované v rámci souborů a následně je v HTML souboru vytvořený odkaz na soubor. Styly mohou také být definované přímo v HTML v značce „style“, nebo mohou být definované přímo na konkrétním elementu v atributu „style“.

2.4.3.2 Tailwind

Tailwind patře mezi utility knihovny učené k stylování webových stránek, úkol je změnit pohled na stylování aplikací, ne nabídka již vytvořených komponent, například Bootstrap nebo Material-UI. Tailwind pouze definuje minimalistické CSS třídy, které obsahuje nejčastěji jednu deklaraci 2.5. Důvodem, proč se využívá tento způsob, místo psaní deklarací přímo na komponentě v style atributu je, že přímo napsané deklarace nepodporují media query, což je standardní způsob, jak definovat responzivní styly na webu, přizpůsobující se velikosti obrazovky.

Obvykle bez použití knihoven se styly definují v rámci CSS souborů a poté jsou odkazem vloženy do HTML souborů, v tomto případě CSS třídy definují vzhled komponent. V dnešní době se weby již tak netvoří. Dnes se dělí uživatelské rozhraní na komponenty pomocí populárních webových frameworků, a tak se dospělo k tomu, že je vhodnější definovat styly přímo s komponentou, aby všechno bylo pohromadě. To dělá kód udržitelnějším, protože nemusíme udržovat globálně definované třídy, kde každá změna může narušit aplikaci, ale máme zaručené, že změna ve stylech ovlivní jenom konkrétní komponentu.

Výpis kódu 2.6 Porovnání CSS z Tailwind CSS

```
# CSS
.button{
    margin:0;
    padding:0;
    background-color:black;
    text-color:white;
}

<button class="btn">button</button>

# tailwind CSS React

function Button(){
    return(
        <button className="m-0␣p-0␣bg-black␣text-white">
            button
        </button>
    )
}
```

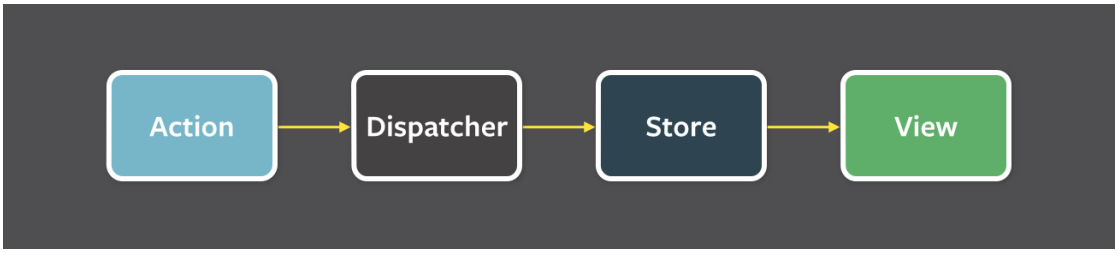
Tento příklad ilustruje, že v dnešní době nemá smysl přemýšlet nad zastaralým způsobem psaní CSS a definovat komponenty v něm, v dnešní době se již přemýšlí na úrovni komponentů v kódu.

Dalším hlavním důvodem udržitelnosti je, že není třeba udržovat vlastní dokumentaci CSS tříd, kde by vývojáři museli informovat ostatní o změnách. Při použití TailwindCSS taková potřeba není, protože základní styly jsou již nadefinované a vývojář může libovolně stylovat aplikaci bez toho, aby informoval ostatní o změnách. Také další výhodou je absence prohledávání velkých CSS souborů, když chceme něco změnit.

2.4.4 Redux toolkit

V dnešní době se používají tzv. reaktivní frameworky pro psaní klientské části. Reaktivita spočívá v tom, že při změně stavu aplikace se uživatelské rozhraní překresluje. Pro správu komplexního stavu aplikace vzniklo několik knihoven, jedna z těchto knihoven je Redux-Toolkit [28].

Knihovna využívá flux [29] architekturu, navrženou Metou. Akce je Pure-function (funkce bez vedlejších efektů) předaná dispatcheru, který vykoná změnu v storu a potom se projeví změny v uživatelském rozhraní.



Obrázek 2.1 Obrázek získan z Meta archivu

Redux-toolkit staví na jeho předchůdci Reduxu a zjednodušuje jeho rozhraní. Samotná knihovna slouží pro snadné udržování stavu aplikace.

2.4.5 NPM

Npm[30] je nástroj pro správu závislostí aplikace, které se vyvíjí v prostředí Node.js. Základní soubor, který udržuje informace o závislostech, je package.json, kde jsou vyjmenované balíčky a jejich akceptovatelné verze. Dále jsou zde metadata projektu a také zde lze definovat příkazy. Dalším důležitým souborem je package-json.lock [31], kde jsou verze závislostí, které jsme stáhli. Závislosti jsou umístěné v složce node-modules.

1. npm install - nainstaluje závislosti specifikované v pacakge.json.
2. npm run lint - zkontroluje kvalitu kódu nástrojem eslint.
3. npm run build - sestaví projekt.

2.4.6 i18next

Jeden z požadavků na aplikaci bylo podporovat dva jazyky – český a anglický, proto jsem využil knihovnu na lokalizaci i18next [32] a její adaptaci pro React. i18next nabízí rozhraní pro konfiguraci lokalizace a pomocné funkce pro překlad v rámci aplikace.

V rámci knihovny React se definuje JSON objekt obsahující překlady pro každý jazyk, objekty by měly obsahovat stejné klíče.

Pro překlad v rámci aplikace slouží React hook 2.4.2.4 `useTranslation`, nabízející funkci pro překlad a instanci i18next nabízející rozhraní pro změnu jazyka a získání jazyka.

2.4.7 Eslint

Eslint [33] je nástroj pro kontrolo kvality kódu v jazyku Javascript/Typescript.

2.5 Technologie na straně serveru

2.5.1 Kotlin

Kotlin [34] je programovací jazyk vyvinutý společností Jetbrains, která vytváří vývojářské nástroje a má vlastní jazykové laboratoře, patří mezi významné firmy v oblasti programovacích jazyků.

Důležitý pojem pro jazyk Kotlin je JVM, což je běhové prostředí, ve kterém běží programy zkompilované do bytecodu. Mezi JVM jazyky patří: Java, Scala, Kotlin a Groovy.

Kotlin byl vytvořen hlavně pro vývoj na android zařízení, ale používá se také pro vývoj webových aplikací.

Mezi základní charakteristiky jazyka Kotlin patří: staticky typovaný, objektově orientovaný (má podporu pro dědičnost, rozhraní, abstraktní třídy).

Jedna z hlavních výhod oproti Javě, ke které je často přirovnáván, je null-safety. Tento koncept nás chrání před vznikem tzv. NullPointer výjimek, protože typy proměnných zahrnují kontrolu, jestli proměnná může nabývat hodnoty null. Například v Jave je toto potřeba řešit Optional rozhraním, což je náročnější na používání a neobsahuje všechny výhody poskytované nativní implementací, například kód se nezkompiluje, pokud nezkontrolujeme, že hodnota není null, to jazyky bez null-safety povolují.

Další velkou výhodou jazyka Kotlin, je že lze jej libovolně mixovat s jazykem Java, na tomto principu závisí jeho možné využití v psaní aplikací pomocí frameworku Spring Boot.

Také je podle mého názoru jazyk značně syntakticky přívětivější než Java díky skvělému rozhraní, které nabízejí jednotlivé objekty, jenž je značně lepší než ekvivalentní v Javě.

Zásadní nevýhodou jazyka je, že aktuálně existuje jediný vývojářský nástroj, který má dobrou podporu pro tento jazyk – Intelij, vyvinutý stejnou společností jako samotný jazyk. To způsobuje závislost na této společnosti.

Kotlin je standardizovaný jazyk pro psaní backendu pro projekty spojené s Anketou ČVUT.

2.5.2 Spring Boot

Spring Boot je webový framework, který je nadstavbou na Spring frameworku a nabízí vývojářům předem nakonfigurované nastavení pro snadné použití. Tento framework je standardizovaný při použití v rámci školních projektů na systému Anketa ČVUT.

Základním principem využívaným v frameworku Spring Boot je návrhový vzor Dependency Injection [35]. 0bjekt si nevytváří závislosti sám, ale jsou mu předané, například v konstruktoru. Dependency injection je specializovaná forma IoC (Inversion of Control).

IoC funguje na principu konfigurace, kde se definuje, jak se má objekty sestavit. Dříve před tím, než vznikl Spring Boot se definovala konfigurace v XML souborech, Konkretně objektům vytvořeným v rámci konfigurace se říká v Spring frameworku beans.

Bean [36] lze definovat v XML souboru postaru, nebo v dnešní době se hlavně používá přístup anotací a scanování souborů za účelem nalezení anotovaných tříd.

Jak jsem zmínil, tak framework byl zejména zvolen za účelem standarizace technologií pro školní projekty, aby to bylo snadnější pro studenty a vyučující je udržovat. Také jsem měl s frameworkem předchozí zkušenosti ze školy a obecně je považovaný za nejlepší framework pro jazyk Kotlin/Java se skvělou podporou vývojářů a je využíván ve velkých firmách v České republice, takže není potřeba se obávat, že by skončila jeho podpora.

2.5.3 Hibernate

JPA je standardizované rozhraní pro ORM ve světě Javy a Hibernate je jenom jeho konkrétní implementací. Nástroj používá návrhový vzor Work of unit [37], který slouží k udržení informací o změnách a potom vše zapíše najednou, tím se zamezuje příliš velké množství synchronzací.

Hibernate využívá pro psaní dotázů HQL [38], dotazovací jazyk podobný SQL, samotné HQL se zkompiluje na SQL, výhodou je, že nám umožňuje psát mezi-platformní dotazy, fungující na různých databázových strojích. Také nám umožňuje chovat se k datům jako objektům, místo řádkům v databázi. Samotné HQL mi ale v rámci mých potřeb nebude stačit.

Projekt vyžaduje dynamickou tvorbu dotazů, kterou nelze docílit běžným použitím čistého SQL, nebo HQL. Je potřeba využít mocnější nástroj umožňující dynamickou tvorbu dotazů. V SQL/HQL bych musel psát dotaz pro každou možnost dotazu. Například pokud bych chtěl stránkovat dotaz a jindy ho nechtěl stránkovat, tak bych musel vytvořit pro tyto dva případy dotaz zvlášť, nebo bych chtěl filtrovat například podle uživatelského jména a příjmení a jindy jenom podle uživatelského jména. V takovém případě Hibernate nabízí Criteria API [39], umožňující tvořit type-safe dotazy dynamicky.

Výpis kódu 2.7 Ukázka tvorby dotazu pomocí Criteria API

```
val cb: CriteriaBuilder = this.entityManager.criteriaBuilder
val cq: CriteriaQuery<SurveyViewAggregationOutput> = (
 cb.createQuery(SurveyViewAggregationOutput::class.java)
)
val root: Root<SurveyView> = cq.from(SurveyView::class.java)
cq.select(
        cb.construct(
        SurveyViewAggregationOutput::class.java,
        cb.max(root[SurveyView_.TOTAL_SUBSCRIBED]),
        cb.min(root[SurveyView_.TOTAL_SUBSCRIBED]),
        cb.max(root[SurveyView_.TOTAL_FILLED]),
        cb.min(root[SurveyView_.TOTAL_FILLED]),
        cb.max(root[SurveyView_.PCT_FILLED]),
        cb.min(root[SurveyView_.PCT_FILLED]),
    ),
)
return this.entityManager.createQuery(cq).singleResult
```

Zásadní problém nastává při udržitelnosti těchto dotazů, jak vidíme v ukázce kódu jsou parametry vybírány pomocí třídy se sufixem „_“. Proto je potřeba využívat metamodel generátor od ORM Hibernate, generující třídy s konstantami podle názvů vlastností entit.

Výpis kódu 2.8 Ukázka vygenerované metamodel třídy

```
@StaticMetamodel(Faculty.class)
@Generated("org.hibernate.jpamodelgen.JPAMetaModelEntityProcessor")
public abstract class Faculty_ {

/**
* @see fit.cvut.anketa.reports.v2.entity.Faculty#facultyId
**/
public static volatile SingularAttribute<Faculty, Long> facultyId;

/**
* @see fit.cvut.anketa.reports.v2.entity.Faculty
**/
public static volatile EntityType<Faculty> class_;

public static final String FACULTY_ID = "facultyId";
}
```

2.5.4 Gradle

Gradle [40] je správce závislostí a pluginů pro aplikace v Kotlinu, tento build tool používá nový backend Ankety[41] na rozdíl od původního, který využíval Maven. Já jsem se osobně již s nástrojem Gradle setkal v rámci předmětů na škole, tak mi toto rozhodnutí vyhovovalo. Zásadní výhoda Gradle nástroje oproti konkurenčnímu nástroji Maven je možnost konfigurace v jazyku Kotlin, což zaručí jednotný jazyk napříč celým backendem reportovacího systému. Také z toho, co jsem se dočetl, je značně rychlejší při kompilaci projektu díky cachování a inkrementální kompilaci [42].

2.5.5 Nástroje pro zlepšení kvality kódu

V rámci projektu použiji dva nástroje pro zlepšení kvality kódu Kotliner a Detekt. Tyto nástroje jsou používány novou implementací backendu Ankety[41] a v rámci toho projektu se osvědčily, a tak pro dodržení jednoty jsem se rozhodl rovněž zavést tyto nástroje.

Kotliner [43] je Gradle plugin, který umí formátovat kód napsaný v Kotlinu a kontrolovat, zda kód splňuje specifikované standardy.

Detekt [44] je Gradle plugin podobný Kotlineru, který kontroluje kvalitu kódu podle předem nadefinovaných pravidel v konfiguraci.

2.5.6 Kotlin Symbol Processing

KSP[45] patří mezi techniky programování zvané meta programování, kde program využívá zdrojový kód jako data pro generování dalších dat, jedná se o podobný nástroj jako má Java pro zpracování anotací. KSP nabízí velice jednoduché objektově orientované rozhraní pro psaní jednoduchých compiler pluginů, samotné rozhraní se má podobat co nejvíce gramatice samotného jazyka. Hlavní stavební jednotkou je `SymbolProcessorProvider` – třída, která má za úkol vrátit instanci procesoru, další důležitá třída je samotný `SymbolProcessor`, starající se o nalezení potřebných symbolů a posledním stavebním kamenem je `SymbolVisitor`, třída využívající ná-vrhový vzor visitor, jejím účelem je navštívit získané symboly a získat z nich informace potřebné pro generování dalších souborů.

KSP vždy posbírá procesory, které jsou předané providery. Procesory se v každém kompilačním cyklu zavolají. Pokud se vygenerují další soubory, tak nastane další cyklus zpracování a tak to běží, dokud se generují další soubory.

Nástroj jsem se rozhodl použít pro generování souborů, protože jsem chtěl vytvořit aplikaci snadno rozšiřitelnou o další reporty a vím, že některé soubory je nutné vytvořit, ale sdílejí stejnou strukturu. Tak vznikla potřeba použít nástroj na usnadnění vývoje ve formě generování opakovaného kódu, abych dodržel Dont Repeat Yourself princip programování.

2.5.7 Kotlinpoet

Kotlinpoet[46] je další knihovna využitá při samotné implementaci reportu na backendu. Jedná se o knihovnu nabízející rozhraní využívající návrhový vzor Builder pro generování souborů v jazyce Kotlin, také obsahuje knihovnu pro použití v rámci KSP. Je velice jednoduchá na použití a není omezená funkcionalitou, lze vygenerovat libovolný Kotlin soubor. Patří mezi klíčové technologie mého řešení.

Výpis kódu 2.9 Ukázka generování anotace

```
AnnotationSpec.builder(RequestMapping::class)
    .addMember("%S", "/${path?.value.toString()}")
    .build()
```

2.5.8 Recharts

Pro tvorbu vizualizací bude použita knihovna Recharts, speciálně vytvořená pro využití v rámci Reactu, využívající skládací způsob tvorby grafů, na rozdíl od běžně známých knihoven jako je Chart-js. Následně jsou uvedeny ukázky z dokumentace knihoven, které ukazují, jak se vytváří graf v příslušné knihovně.

Odkaz na kód [47]

Výpis kódu 2.10 Ukázka definice grafu při použití knihovny Rechart

```
    <BarChart width={730} height={250} data={data}>
      <CartesianGrid strokeDasharray="3␣3" />
      <XAxis dataKey="name" />
      <YAxis />
      <Tooltip />
      <Legend />
      <Bar dataKey="pv" fill="#8884d8" />
      <Bar dataKey="uv" fill="#82ca9d" />
    </BarChart>
```

Odkaz na kód [48]

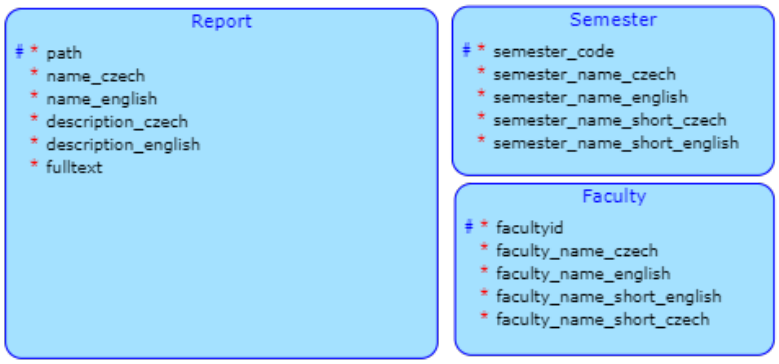
Výpis kódu 2.11 Ukázka definice grafu při využití Chart.js

```
    export const options = {
    responsive: true,
      plugins: {
        legend: {
          position: 'top' as const,
        },
      },
    };
    const labels = [...labels];
    export const data = {
      labels,
      datasets: [
        {...dataset1},
        {...dataset2},
      ],
    };
export function App() {
  return <Bar options={options} data={data} />;
}
```

Je zřejmé, že přístup knihovny Recharts je mnohem lidštější a snadněji pochopitelný, než přístup starších a nemoderních knihoven jako je Chart.js, celý graf lze definovat pomocí React komponent a také podporují velice dobré rozhraní pro vlastní rozšíření. Z mého hlediska je to aktuálně nejlepší knihovna na tvorbu grafu, se kterou jsem se setkal z hlediska syntaxe a to lze přisoudit tomu, že knihovna byla navržená přímo pro React, místo aby se definovala obecně.

2.6 Návrh databáze

Pro vytvoření aplikace jsem navrhl nový databázový model. Místo toho, abych uvažoval databázi anket, tak jsem jsem se rozhodl implementovat systém obecně, takže systém může obsluhovat snadno neomezený počet reportů.



Obrázek 2.2 Vytvořeno nástrojem portál DBS

Tabulky semestr a fakulty již byly vytvořené v databázi ankety, já jsem přidal pouze jedinou tabulku reports, která obsahuje metadata pro klienta, aby se mohl dotazovat na reporty. Samotné reporty lze vytvořit z libovolné tabulky, pohledu, nebo materializovaného pohledu.

2.7 Návrh UI

Jeden z úkolů této práce je navrhnout nové uživatelské rozhraní klientské části aplikace. Původní rozhraní nepůsobí příliš moderně a obsahuje hodně chyb a věcí, které působí nejasně. V této sekci popíši postup návrhu nového rozhraní a řešení nedostatků původního rozhraní.

Nejdříve bych popsal nástroj pro návrh uživatelského rozhraní Figma, jenž jsem se rozhodl použít v rámci této práce, jelikož mám z již zmíněným nástrojem bohaté zkušenosti ze školních projektů i z vlastních projektů.

2.7.1 Figma

Figma je nástroj napsaný v jazyce WebAssembly, aktuálně je nejlepší svého druhu. Účelem je snadná kolaborace při tvorbě interaktivních prototypů webových aplikací. Důvod pro využití nástroje jako Figma ve vývoji je možnost vytvořit si představu o rozhraní bez vynaložení velkého úsilí, které je třeba vynaložit při programování. Časová ztráta špatného návrhu v projektu je mnohonásobně větší, než špatného návrhu v návrhářském nástroji.

Základním principem návrhu rozhraní v nástroji Figma je rozdělení rozhraní na atomické komponenty, z nich skládat složitější komponenty, ještě složitější komponenty, až nakonec z nich vytvoříme jednotlivé stránky. Druhým zásadním principem je tvorba variant jednotlivých komponent, například nadpis první úrovně a nadpis druhé úrovně. Je to velice podobné tvorbě rozhraní v Reactu 2.4.2.

2.7.2 Základní pravidla UI návrhu

Před tím, než se pustím do popisu návrhu, tak zmíním pár základních principů návrhu uživatelského rozhraní, zveřejněných Figmou [49].

1. **Hierarchie:** První důležitý princip pro správný návrh je správná hierarchie, je třeba, aby důležité prvky byly vidět na první pohled a méně důležité prvky, aby nebraly důležité místo.
2. **Konzistence:** Uživatelské rozhraní musí být konzistentní, například modré tlačítko, pokud již jednou využité jako odkaz, tak by v celé aplikaci mělo být využité jako odkaz.
3. **Kontrast:** Zásadním principem je kontrast, například text na světlém pozadí by měl být tmavý a text na tmavém posadí by měl být světlý.
4. **Blízkost:** Uživatelé podvědomě považují prvky sobě blízké za funkčně podobné, proto je dobré seskupit prvky se stejným významem.

2.7.3 Volba barev

Jelikož aplikace spadá pod ČVUT, tak jsem chtěl zavést barvy, které vystihovaly ČVUT jako instituci. Tak jsem analyzoval grafický manuál ČVUT [50], kde lze najít barvy využívané ČVUT. Já jsem osobně se rozhodl použít hlavně tradiční modrou ČVUT rgb(0 101 189), další doporučené barvy jsem se rozhodl nepoužít, jelikož nejsou obvyklé u webových rozhraní, pro tmavý režim používám stejnou barvu, jenom z nižším nasycením.

Volba dalších barev byla podle průmyslově standardizovaných asociací barev. Zelená znamená úspěch, modrá slouží pro zobrazení informací, červená znamená chybu a oranžová/žlutá je asociovaná s varováním. Tento systém používají nejpopulárnější UI frameworky jako Bootstrap [51] a Material UI [52]. Samozřejmě, že lze použít i jiné barvy pro tyto účely, ale pro nejlepší uživatelskou zkušenost je dobré dodržet, co je již zaběhnuté. Například tlačítko vymazat účet v zelené barvě by mohlo někoho zmást.

Dále je již v průmyslu obvyklé volit odstíny námi vybraných barev. Barvy mají odstíny od 50, 100, 200, 300, 400, 500, 600, 700, 800 a 900, třeba Material UI začíná od 50, TailwindCSS [53] pro vlastní barevnou paletu začíná od 100. Pro tvorbu vlastní palety jsem využil metodu zveřejněnou youtouberem UX Tools [54], který navrhl vlastní systém volby barev. Systém spočívá ve využití formátu HSB, kde H je odstín barvy, S je nasycenost barvy a B je světlost barvy. Při volbě odstínu se mění jenom nasycenost a světlost. Začíná se od vysoké světlosti a nízké nasycenosti, postupně se ubírá světlost a zvětšuje nasycenost.

Pro tmavý model jsem použil méně syté barvy pokud původní zvolené byly příliš syté, tuto informaci jsem získal od youtoubera Malewicz [55] v jeho kurzu na návrh tmavého režimu.

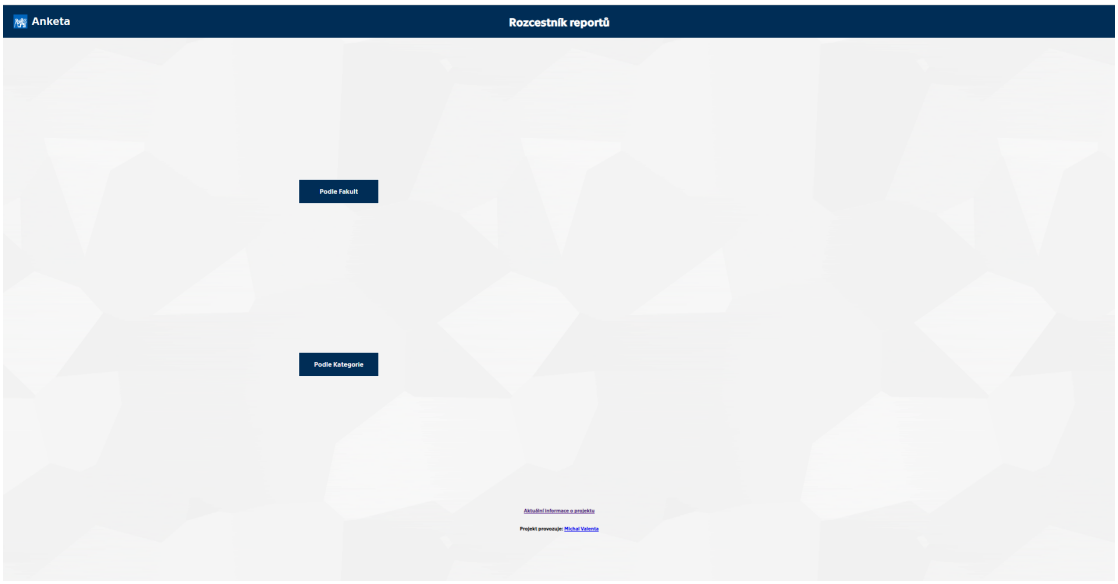
2.7.4 Volba zaoblení rohů

Jelikož jsem studentem fakulty ČVUT, tak často využívám webové stránky jako jsou KOS, Anketa a samotný web ČVUT využívají hranaté rohy, až na Projects, co využívá zaoblené rohy, proto jsem se rozhodl navrhnout rozhraní s hranatými rohy.

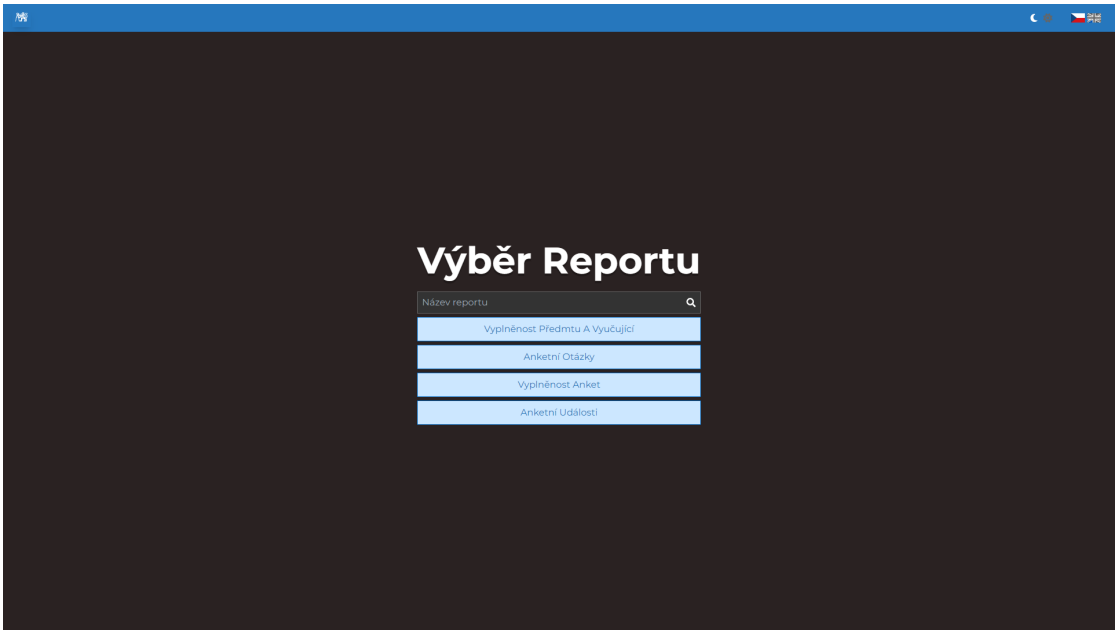
2.7.5 Návrh domovské stránky v1

Na hlavní stránce aktuálně využívané verze 2.3 jsou pouze dvě nevycentrovaná tlačítka, což velice špatně využívá volný prostor. Tlačítka výběr reportů umožňují pomocí akce „hover“ vybrat report. Můj návrh 2.4 řeší problém s nevyužitým místem tak, že se značně zvětší nadpis a seznam reportů se zobrazí přímo na stránce.

Další problémem původního návrhu je, že na stránce jsou dvě tlačítka se stejnou funkcionalitou, což může být matoucí pro uživatele, například uživatel neví na první pohled, jestli obě cesty vedou na stejnou stránku. Také proces na stránku reportů vyžaduje o jeden výběr víc, konkrétně výběr fakulty a reportu. Můj návrh zjednodušuje prvotní rozhodnutí na jedno kliknutí, a dokonce dává možnost uživateli filtrovat, což není možné v aktuální verzi.



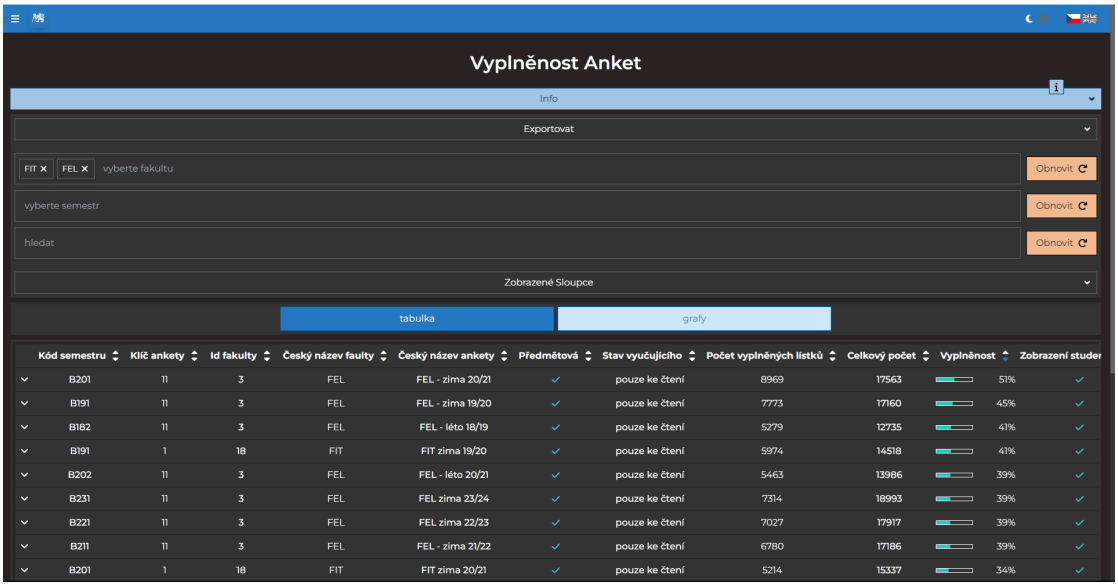
Obrázek 2.3 Screenshot z předchozí verze



Obrázek 2.4 Screenshot z nové verze

2.7.6 Návrh stránky reportu

Stránka reportů je druhá stránka aplikace. Stránka reportu je podstatně složitější než úvodní stránka 2.4.



Obrázek 2.5 Nová stránka reportu

2.7.7 Návrh tabulky

Tabulka v aktuální verzi vše zobrazuje ve formě textu, to působí velice jednotvárně, také nedostatek barvy působí nepřívětivě, proto jsem se rozhodl tabulku lehce oživit. Tabulku jsem navrhl tak, že pro různé druhy sloupců se obsah zobrazí jinak. Například pro sloupce obsahující buď ano, nebo ne, jsem se rozhodl obsah vyjádřit barevnou ikonou. Pro ano jsem zvolil zelenou barvu a pro ne jsem zvolil červenou barvu, což jsou barvy tradičně asociované s tímto významem. Pro delší texty jsem se rozhodl, že se nezobrazí celý obsah, aby se nevyskytovaly různě vysoké řádky.

Jelikož reporty mají velké množství sloupců, které nelze všechny zobrazit na obrazovku najednou, tak jsem se rozhodl přidat možnost zobrazit řádek na šířku obrazovky ve formátu karty.

2.7.8 Návrh akční sekce stránky reportu

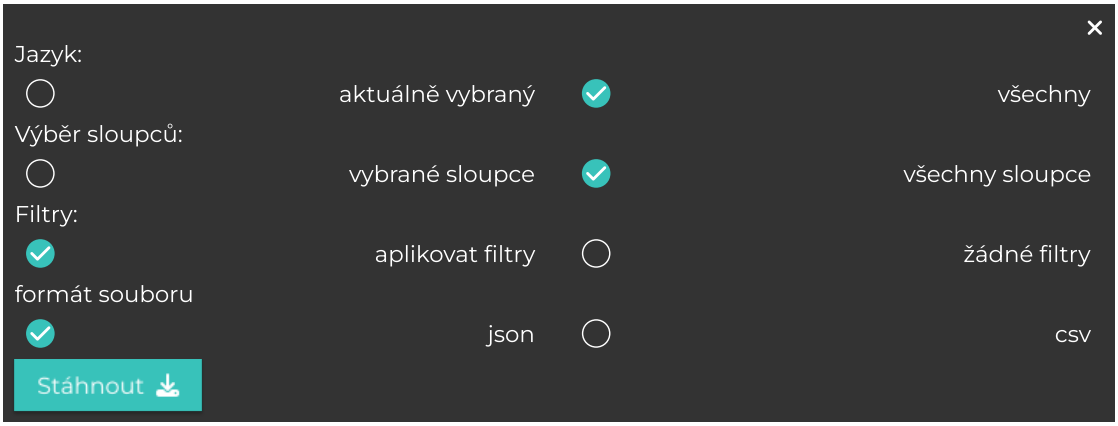
Jeden z nefunkčních požadavků je jednoduchost, hlavním úkolem bylo vymyslet takovou sekci, aby byla jednoduchá na použití a neměla ohromující efekt, že je tam příliš informací. V samotné akční sekci si může uživatel vybrat jaké sloupce si zobrazí, může exportovat zvolená data a filtrovat data.

Aby akční sekce nepůsobila příliš ohromující, je výběr sloupců a export skládací element, tak si může uživatel vybrat, co nechá zobrazené a zároveň při prvním otevření není zobrazeno příliš informací.

2.7.9 Návrh formuláře pro export

Export je navržený tak, aby dal uživateli kontrolu nad daty, které vyexportuje. Lze zvolit, jaký jazyk dat bude exportovaný, protože aplikace obsahuje data v angličtině a v češtině, tak je možné exportovat data v jednom z nich, nebo v obou jazycích. Dále lze zvolit, jestli se respektuje výběr sloupců uživatele, nebo chce vyexportovat všechny sloupce. Dále je možné zvolit, jestli uživatel chce aplikovat vybrané filtry, nebo exportovat všechna data. A pomocí posledního pole formuláře může uživatel zvolit formát dat, aktuálně je v plánu umožnit exportovat data ve formátu CSV

a JSON, v budoucnosti bych chtěl přidat podporu pro další formáty jako je Apache Parquet a Avro, což na rozdíl od řádkových formátů jako CSV jsou sloupcové formáty.



Obrázek 2.6 formulář export

2.7.10 Návrh volby sloupců

Jelikož aplikace je aktuálně vícejazyčná a aplikace obsahuje velké množství sloupců, tak jsem se rozhodl přidat možnost vybrat si, jestli se zobrazí sloupce všech jazyků, nebo se zobrazí sloupce z některého jazyka, tím uživateli můžeme zpřehlednit prohlížení dat. Dále aplikace nabízí volbu jednotlivých sloupců, pokud výběr podle jazyka není dostačující.

Obrázek 2.7 Volba sloupců

2.7.11 Návrh formuláře na vyhledávání

Jedním z nejdůležitějších rozhodnutí v návrh aplikace bylo navržení filtrování. Hlavní požadavek na filtrovaní byla jednoduchost, aby neměl uživatel problém filtrovat data bez technického vzdělání a bez znalosti syntaxe.

Během tvorby aplikace došlo k nespočetným změnám v návrhu formuláře. Podněty pro vylepšení jsem měl příležitost získat každý týden během konzultací a také při vlastním testování aplikace. Hlavními závěrem je důležitost snadného filtrování podle semestru a podle fakulty, proto jsem se rozhodl pro tyto dva sloupce vytvořit vlastní filtrovací pole. Dále stránka obsahuje hlavní filtrovací pole, o kterém se zmíním v následující kapitole. Poslední filtrovací pole je čistě textové a má za úlohu vykonat textové vyhledávání přes všechny sloupce.

2.7.12 Návrh hlavního filtrovacího pole

Nejdůležitějším vyhledávacím prvkem aktuální verze je pole, které jsem nazval hlavní. Pole má umožnit uživateli vyhledávat podle všech sloupců. Pole funguje na principu, že uživatel může

přidat libovolný počet filtrů. Filtry mají specifickou formu podle typu sloupce.

Pro filtrování podle data a času jsem se rozhodl dát uživateli na výběr mezi třemi možnostmi: datum a čas, který již nastal, datum a čas, co ještě nenastal a poslední možnost je interval. Důvod pro tuto volbu je to, že zadávání data a času může být časově náročné, proto jsem se rozhodl dát uživateli na výběr z méně náročných voleb.

Dále mezi způsoby filtrování patří pole, jenž má výběr z možností buď ano, nebo ne. Pole jsem realizoval pomocí prvku switch. Prvek switch není dostupný v samotném HTML5, proto jsem jej implementoval sám, samotný prvek je vizuální změnou zaškrtávacího políčka.

Následujícím možným filtrem je textové pole s našeptáváním. Pole s našeptáváním slouží pro hledání textových polí kratšího charakteru, nebo identifikátorů. Nápovědy umožní uživateli snadno vyhledat informace bez nutnosti, aby napsal celý název, nebo znal název přesně.

Dalším filtrovacím polem je výběrové pole, které se obecně v HTML5 nazývá select. Select slouží pro sloupce, které mají předem známé hodnoty, ale nejsou typu ano/ne.

Posledním filtrovacím polem implementovaným v aplikaci je číselný interval. Intervalové pole má za úkol prohledávat sloupce s vysokou kardinalitou, dobrým příkladem pole z vysokou kardinalitou je třeba vyplněnost, nebo počet vyplňujících studentů.

V rámci pole lze vybrat stejné pole vícekrát, to zvětšuje flexibilitu filtrování oproti předchozím dvěma verzím aplikace. Lze například vyfiltrovat více předmětů, nebo více vyučujících. Na filtry stejného pole se bude aplikovat logická spojka `OR` a na filtry obecně se aplikuje logická spojka `AND`. Nevylučují, že v budoucnu bude uživatel mít možnost si vybrat jakou spojku aplikuje na filtry různého typu.

Obrázek 2.8 Hlavní formulářové pole

Kapitola 3
Vývoj

3.1 Metodologie vývoje

Iterativní [56] vývojové metodologie fungují na principu iterací, kdy se každá jednotlivá iterace skládá z analýzy, návrhu, vývoje a testování. V rámci každé iterace se přidávají nové funkcionality, podle zpětné vazby z předchozí iterace. Až poslední iterace se považuje za hotový produkt.

Protipólem iterativního vývoje je vodopádový přístup, který se skládá z jedné velké iterace, která se skládá z pěti fází: požadavky, návrh, implementace, testování a údržba. Každá z jednotlivých fází prochází kontrolou kvality.

Iterativní metody mají určený časový interval na vytvoření nové verze. V rámci iterativního vývoje není nic „vytesané v kameni“, požadavky se mohou měnit z iterace na iteraci. Výhodou iterativního oproti vodopádovému přístupu je, že reaguje na zpětnou vazbu klienta a tak je větší pravděpodobnost, že výsledný produkt nebude k zahození. Vodopádový způsob vývoje se proto v dnešní době již využívá velice zřídka, protože riziko vytvoření něčeho nepoužitelného je příliš vysoké, pokud nemáme zcela jasné předem, jaké požadavky má produkt splňovat.

Svoji práci jsem vyvíjel iterativně, protože vedoucí práce je jeden budoucích uživatelů aplikace, tak v rámci každé konzultace jsem představil, co jsem vykonal a získal zpětnou vazbu, na kterou jsem mohl následující týden reagovat.

3.2 Verzovací model - Git flow

Projekt jsem vyvíjel verzovacím modelem Gitflow [57]. Gitflow má dvě hlavní větve, které nikdy nezanikají: `develop` a `master`. Dále má pomocné větve: `feature`, `hotfix` a `release`.

Feature větve obsahují novou funkcionalitu, vznikají z větve `develop` a při dokončení se slučují zpět do větve `develop`.

Release větve vznikají, když se vytváří nová verze aplikace, do větví se poté již nepřidává žádná funkcionalita, jenom záležitosti ohledně dokumentace a ošetřování chyb. Větev vzniká z `develop` větve a slučuje se do `master/main` větve, kde vznikne značka nové verze a sloučí se zpět i do `develop` větve.

Hotfix větve řeší chyby a vznikají z `main/master` větve a slučují se zpět do `main/master` větve a do `develop` větve.

3.3 Technologie nasazení

3.3.1 Gitlab

Gitlab [58] je dev-ops nástroj, ČVUT si udržuje vlastní instanci. Jeho hlavní funkcionalita je správa git repozitáře, další důležitá funkcionalita je CI/CD pipeline, kterou používám v projektu na nasazení aplikace. Pipeline se definuje v souboru gitlab-ci.yml, nejmenší jednotka v souboru je úloha (job), zde se definuje, co se má vykonat a jak. Největší jednotka je fáze (stage) určující pořadí kdy se vykonají úlohy v ní definované, které se vykonají paralelně.

Například máme fázi sestavení, kdy se projekt sestaví a předá do fáze nasazení. V fázi nasazení máme úlohu pro každé prostředí, do kterého chceme aplikaci nasadit.

3.3.2 Apache

Apache [59] je open-source HTTP server napsaný v jazyce c++. Projekt je již desítky let na trhu, to svědčí o jeho spolehlivosti a důvěryhodnosti. Úlohou HTTP serveru je obsluhovat velké množství uživatelů paralelně, posílat uživatelům soubory, cachovat soubory a mnoho další funkcionalit. Samotný Apache server je vysoce rozšiřitelný, nabízí velké množství modulů a umožňuje uživatelům psát si vlastní moduly.

3.3.3 Apache tomcat

Spring Boot framework, má jako výchozí web server nastavený Apache Tomcat [60], což je webový server napsaný v jazyce Java. Na rozdíl od Apache, který slouží pro posílání statického obsahu uživatelům, tak Apache Tomcat je aplikační server pro java aplikace.

3.4 Nasazení serveru

Před nasazením je třeba projekt připravit, aby podporoval různé konfigurace podle prostředí. To se v rámci frameworku Spring Boot řeší application-profil.properties soubory. Tento soubor by neměl obsahovat citlivá data přímo, ale měl by je získávat z proměnných prostředí. V souboru je potřeba specifikovat odkaz na databázi, přihlašovací údaje do databáze a další konfigurační možnosti, v rámci Spring Boot je správné mít pro každé prostředí vlastní profil a vlastní konfigurační soubor.

Nasazení serverové částí funguje na stejném principu jako v práci Oleksandra Chmela [41] [61], který pracoval ve své diplomové práci na refaktoring backendu Ankety ČVUT. Důvod využití stejného postupu je, že správce ankety upřednostňuje, aby projekty měly stejný způsob nasazení. Proto, abych mohl svoji serverovou část nasadit, bylo potřeba analyzovat způsob využitý výše zmíněnou prací, ale jelikož v práci není způsob dokumentován, tak jsem musel provést výzkum a zjistit jak to funguje. Způsob nasazení jsem přizpůsobil vlastní potřebě, tak jsem se jej rozhodl popsat.

Nasazení probíhá v rámci Gitlab CI/CD pipeline při události merge request na větvi develop.

Pipeline se skládá ze dvou fází 3.3.1. První fáze je fáze sestavení, v rámci ní se projekt sestaví a předá jar soubor v artefaktu do druhé fáze. Druhá fáze je nasazení a v ní se nejdříve inicializuje ssh klíčem ssh-agent, potom se zkopíruje jar soubor na server pomocí secure-copy a spustí se skript restartující službu běžící na serveru. Skript běží pod uživatelem anketa-reports, kterého jsem pro tento účel vytvořil. Restartování služby je privilegovaný příkaz, který potřebuje, aby byl specifikovaný pro daného uživatele v souboru sudoers, aby mohl běžet mimo uživatele root. Samotná služba je definovaná v složce /etc/systemd/system, kde se služby definují.

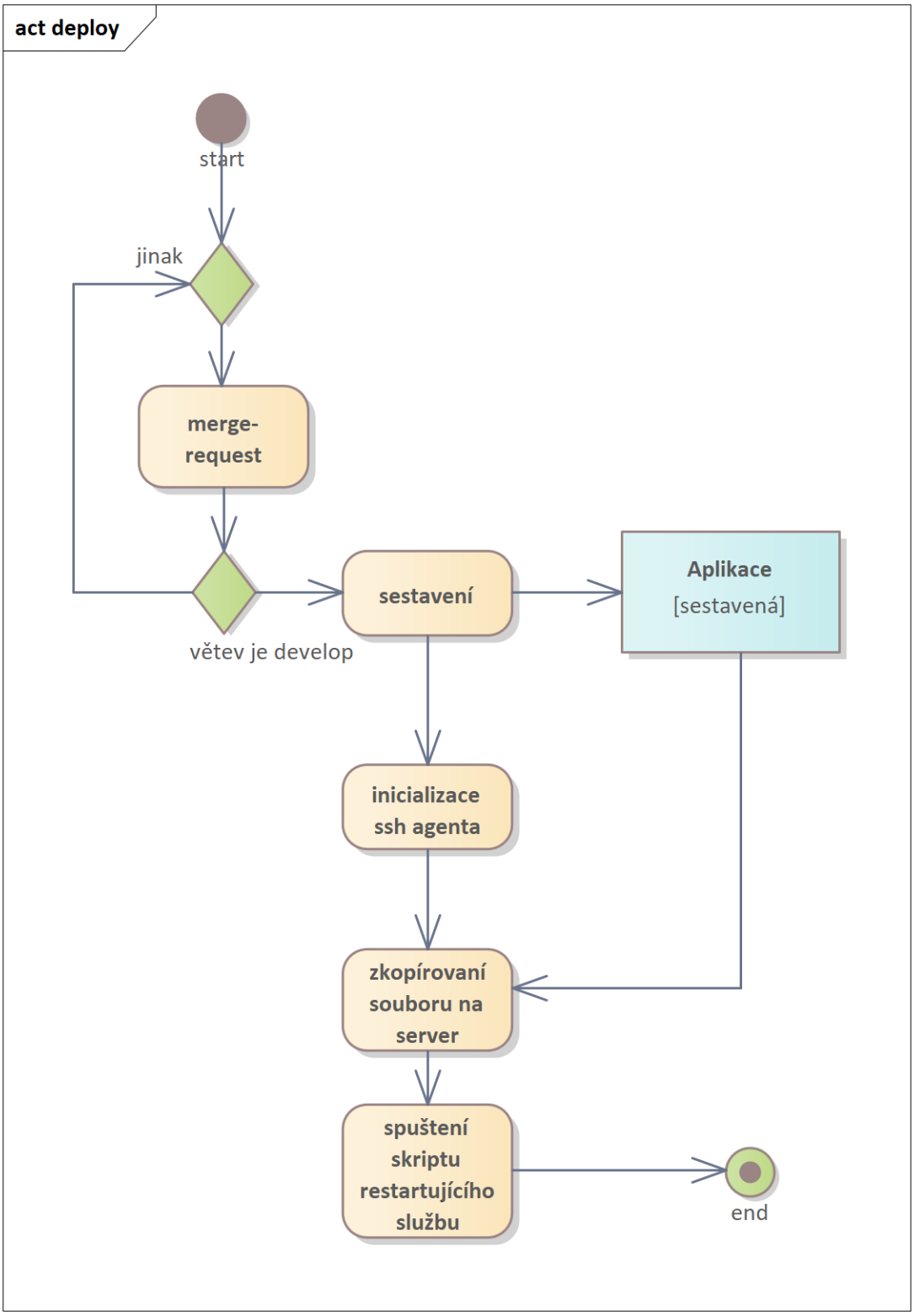
3.5 Nasazení klienta

Před samotným procesem nasazení, stejně jako v rámci serverové části, je potřeba projekt připravit, aby jej bylo možné nasadit na různá prostředí. Nutností, aby šla klientská část nasadit je, aby v rámci projektu bylo možné změnit url, kde se aplikace nachází a všechno fungovalo tak, jak v rámci vývoje. Tento požadavek například nesplňovala druhá verze ankety, obsahující „natvrdo“ napsané odkazy v požadavcích na server.

Nasazení klientské části probíhá velice podobně jako serverové části, zejména co se týká definice Gitlab pipeline.

Proces probíhá stejně, ale jak bylo zmíněno v 3.2, tak je klientská část vyvíjená v jiném repozitáři. Klient se nasazuje při merge requestu na větev develop. V rámci pipeline se projekt sestaví a pomocí artefaktu se předá do stage deploy, kde se inicializuje ssh-agent stejně jako u serverové části. Následně se spustí vzdáleně skript, který vyčistí složku od původních souborů. Tento krok je nutný, jelikož se názvy statických souborů generují náhodně a nepřepisují se při novém nasazení. Kdyby se nemazaly, tak by na serveru brzy bylo velké množství zbytečných souborů. Je sice možné nastavit, aby se názvy negenerovaly náhodně, ale jelikož se statické soubory předávají Apache serveru, který si soubory ukládá do cache, tak by bylo nutné cache při nasazení pročisťovat, avšak protože stejný Apache server využívají ostatní aplikace spojené s Anketou, tak by to nebylo vhodné.

Klient je SPA 2.1.1, tudíž je potřeba nadefinovat .htaccess soubor, říkající Apache serveru, že všechny požadavky se mají směrovat na index.html.



Obrázek 3.1 Diagram procesu nasazení

Kapitola 4
Implementace

4.1 Tvorba komponent rozhraní

Důležitou součástí mojí práce byla tvorba React komponentů, ze kterých se skládá rozhraní aplikace. Komponent má definovaná data, která očekává (v React terminologii se těmto datům říká `props`). Dále, aby komponent byl konfigurovatelný, tak musí přijímat Reactem definované rozhraní použitého HTML elementu. Toto umožňuje lepší znovupoužití, například kdyby uživatel chtěl změnit styl, tak má tuto možnost. Pro změnu stylů v mojí aplikaci používám knihovnu tailwind-merge, která nabízí funkci, jenž umožňuje konfigurovatelné přepisování stylů napsaných pomocí Tailwind-css. Další věcí, kterou může nabízet rozšiřitelný komponent, je možnost jej referencovat jeho rodičem, proto je potřeba použít Reactem vytvořený HOC 2.4.2.3 `forwardRef`, což se například využívá v mojí aplikaci pro buňky tabulky. To nám umožňuje získat informace o jednotlivých HTML elementech, jako například jejich výšku, šířku a další vlastnosti, také to lze využít pro zjištění zda se elementy překrývají.

Výpis kódu 4.1 Ukázka konfigurovatelného komponentu

```
export interface TableCellBaseProps extends
React.PropsWithChildren<React.HTMLProps<HTMLTableCellElement>> {
}

export interface TabelCellBasicProps extends TableCellBaseProps {
    textWrapperProps?: React.HTMLProps<HTMLDivElement>
}

const TableCellBasic =
forwardRef<HTMLTableCellElement, TabelCellBasicProps>
(function ({
    textWrapperProps = { className: '' },
    children,
    className: cellClassName, ...rest }: TabelCellBasicProps,
    ref
) {

    const { className: textWrapperClassname } = textWrapperProps;

    return (
        <td className={twMerge(
            'p-2', cellClassName
        )} {...rest} ref={ref}>
            <div className={twMerge(
                "flex␣items-center␣justify-center", textWrapperClassname
            )}>
                {children}
            </div>
        </td>
    )
})
```

4.2 Implementace reportu

4.2.1 Databázová část

Report je nejprve potřeba definovat v databázi jako pohled, materializovaný pohled, nebo tabulku, aplikace nepodporuje reporty využívající více tabulek najednou, ale to samozřejmě lze obejít tvorbou již zmíněného pohledu, nebo anotací nabízenou ORM Hibernate2.5.3 Subselect, která umožňuje definovat SQL dotaz přímo u entitní třídy jako parametr zmíněné anotace.

Jak jsem zmínil v návrhu, tak jsem se rozhodl použít knihovnu Oracle Text2.2 pro full-textové vyhledávání, rád bych ukázal, jak se takový index definuje a jak se takový index synchronizuje.

Full-textové vyhledávání po dohodě s vedoucím bude využité v reportu otázky, který obsahuje větší množství textu, také bude využité pro sloupec jméno vyučujícího a název předmětu.

Výpis kódu 4.2 Ukázka texového indexu přes jednu tabulku

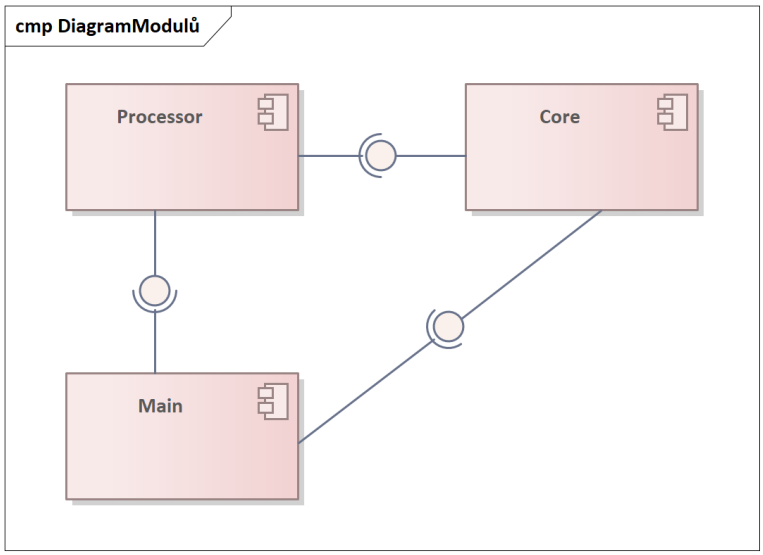
```
begin
    ctx_ddl.create_preference('question_store', 'MULTI_COLUMN_DATASTORE');
    ctx_ddl.set_attribute('question_store', 'columns',
                          'QUESTION_KEY,' ||
                          ' HAS_VALUE_ANSWER,' ||
                          ' HAS_TEXT_ANSWER,' ||
                          ' VALUE_QUESTION_CZ,' ||
                          ' VALUE_QUESTION_EN,' ||
                          ' TEXT_CONTEXT_CZ,' ||
                          ' TEXT_CONTEXT_EN,' ||
                          ' TEXT_EXPL_NORMAL_CZ,' ||
                          ' TEXT_EXPL_NORMAL_EN,' ||
                          ' TEXT_EXPL_BAD_CZ,' ||
                          ' TEXT_EXPL_BAD_EN,' ||
                          ' TEXT_QUESTION_NORMAL_CZ,' ||
                          ' TEXT_QUESTION_NORMAL_EN,' ||
                          ' TEXT_QUESTION_BAD_CZ,' ||
                          ' TEXT_QUESTION_BAD_EN,' ||
                          ' PLACEHOLDER_CZ, PLACEHOLDER_EN,' ||
                          ' MANDATORY, RELATETED_TO_TEACHER,' ||
                          ' MV_COURSE_ID_COURSE, PRIMARY');
end;

create index question_idx on QUESTION (VALUE_QUESTION_CZ)
indextype is ctxsys.context parameters
    ('datastore question_store');
```

Výpis kódu 4.3 Ukázka textového indexu na jeden sloupec

```
CREATE INDEX full_name_text_index ON mv_teacher (full_name)
    INDEXTYPE IS ctxsys.context
    PARAMETERS ('SYNC (ON COMMIT)');
```

4.2.2 Serverová část

Backend se skládá ze tří modulů: processor, core a main. Modul processor má za úkol zpracovat anotace nacházející se v modulu main a vygenerovat podle nich patřičné zdrojové kódy, core obsahuje veškerou logiku sdílenou napříč ostatními moduly a main obsahuje logiku, která není sdílená, například entitní třídy.



Obrázek 4.1 Diagram modulů

Aplikace využívá ORM Hibernate pro práci s databází. To má nevýhody a výhody, jednou z nevýhod je složitější přidávání nových reportů do aplikace, protože potřebujeme vytvořit třídu reprezentující tabulku v databázi, což značně omezilo moje možnosti, lepší řešení by bylo použít čisté SQL, které nezávisí na definici entitních tříd.

Tvorba reportu aktuálně spočívá v přidání nové třídy a vhodné anotaci konkrétní třídy. Report pro konfiguraci nabízí několik anotací, anotace je koncept v jazyku Kotlin pro přidání metadat třídám, proměnným, funkcím. Aktuálně jsou dostupné anotace: `Fulltext`, `ColumnMetadata`, `Vizualization`, `Option` a `Report`.

`Report` anotace slouží k označení tabulkové třídy pro zpracování v rámci kompilace, kdy se vytvoří pro danou třídu kontroler zpřístupňující report na webu a repozitář, který získává data z databáze. Také se za běhu vytvoří záznam v tabulce, upozorňující klientskou část aplikace na nový report. Anotace konfiguruje cestu, pod kterou bude report dostupný na webu, název, popis reportu a zda se povoluje full-textové vyhledávání.

`Fulltext` anotace slouží k označení sloupce, že má Oracle Fulltext index a je použitý při textovém prohledání reportu.

`ColumnMetadata` konfiguruje sloupec z databáze, jak se bude chovat na straně klienta a na straně serveru. Definuje název sloupce, popis sloupce, jak se bude sloupec filtrovat, jak se sloupec zobrazí v tabulce, v jakém jazyce je napsaný a jak se budou hledat nápovědy pro daný sloupec, také pokud sloupec má omezený počet hodnot, tak je možné specifikovat překlady, dále lze specifikovat jestli se ve výchozím stavu sloupec zobrazí a poslední možnost je škálování, které se aplikuje na číselné hodnoty, například převod na procenta.

Výpis kódu 4.4 Anotace metadat sloupce

```
@ColumnMetadata(
        nameCzech = "nazev␣predmetu",
        nameEnglish = "course␣name",
        language = Language.ANY,
        inputType = InputType.BASIC,
        columnType = ColumnType.BASIC,
        filteringType = FilteringColumnType.FULLTEXT,
        descriptionCzech = "nazev␣predmetu",
        descriptionEnglish = "course␣name",
    )
```

`Option` se používá v rámci již zmíněné `ColumnMetadata` pro definici překladů omezeného počtu hodnot. Například se překlad využívá pro filtrování pomocí výběrového pole, nebo při zobrazení hodnot v tabulce.

Výpis kódu 4.5 Ukázka použití anotace Option

```
@ColumnMetadata(
        nameCzech = "predmetova",
        nameEnglish = "is␣course␣survey",
        language = Language.ANY,
        inputType = InputType.YES_NO,
        columnType = ColumnType.YES_NO,
        filteringType = FilteringColumnType.BASIC,
        descriptionCzech = "jestli␣je␣anketa␣predmetova,␣nebo␣neni",
        descriptionEnglish = "is␣course␣survey",
        options = [
            Option(
                tag = "Y",
                tagAliasCzech = "predmetova",
                tagAliasEnglish = "course"
            ),
            Option(
                tag = "N",
                tagAliasCzech = "nepredmetova",
                tagAliasEnglish = "non-course"
            ),
        ],
)
```

Poslední anotace je `Vizualization` přiřazující reportu graf, který se zobrazí na straně klienta.

Aby bylo možné soubory generovat v rámci kompilačního kroku, tak bylo zapotřebí využít Kotlin Symbol Processing2.5.6. Proces generování probíhá ve dvou krocích, nejdříve se vygeneruje repozitář a v následujícím kompilačním cyklu se vygeneruje kontrolér. Pro generování souborů se používá knihovna KotlinPoet2.5.7. Generování by bylo také možné provést jinak, že by se vygeneroval pouze jeden repozitář a jeden kontroler, ale já jsem se rozhodl pro variantu, kdy každý report má vlastní kontroler, protože mi to přišlo přehlednější a výsledná funkcionalita by byla totožná.

Repozitář má jednotné rozhraní napříč všemi reporty.

1. **find:** je hlavní metoda vykonávající proces hledání.
2. **countQuery:** najde celkový počet objektů.
3. **generateOrders:** vytvoří řazení.
4. **generatePredicates:** vygeneruje predikáty.
5. **generateHints:** získá rady, co nejpodobnější dotazu.
6. **findBounds:** najde maximum a minimum číselné hodnoty, používám při vyhledávání rozsahem
7. **validateGroupData:** validuje a parsuje formát seskupení.

Dalším krokem je registrace reportů v databázi, což probíhá při startu aplikace, kde poslouchající třída zareaguje na tuto událost a pomocí runtime skanování anotací, jenž je realizované pomocí velice výkonné paralelizované knihovny Classgraph, využívající reflexi pro získaní informací o programových objektech, v mém případě hledám anotaci Report.

Díky KSP a Classgraph knihovně jsem využil dva typy zpracování anotací runtime a compile time.

Teď bych rád zmínil, jak jsem dokázal implementovat repozitář, aby podporoval operace jako je `group by`, stránkování, filtrování a vše najednou pro libovolný počet sloupců a libovolné entity, to považuji za jednu z nejzajímavějších součástí na své práci.

Protože používám ORM Hibernate a framework Spring, taky silně typovaný jazyk jako je Kotlin, tak nebylo nejjednodušší to vymyslet a obejít všechny pasti, které mi technologie nastražily.

Nejprve bylo potřeba vymyslet jak získat data obecně, když chceme z databáze všechny sloupce a když chceme libovolnou podmnožinu dat při využívaní operace `group by`. Criteria Api nabízí několik možností jak získat data, pokud se nespecifikuje nic, tak nám vrátí data jako entitní třídy. Dále lze specifikovat jinou třídu a použít klauzuli `construct` volající konstruktor dané třídy, nebo poslední možnost využití n-ticového dotazu, který vrátí odpověď jako n-tici hodnot bez informace o datech.

Výpis kódu 4.6 Získaní dat jako seznamu entitních tříd

```
val cb: CriteriaBuilder = this.entityManager.criteriaBuilder
val cq = cb.createQuery(Semester::class.java)
val root = cq.from(Semester::class.java)
```

Výpis kódu 4.7 Klauzule construct

```
val cb: CriteriaBuilder = this.entityManager.criteriaBuilder
val cq = cb.createQuery(SemesterWrapper::class.java)
val root = cq.from(Semester::class.java)
cq.select(cb.construct(SemesterWrapper.class, root["semesterCode"]))
```

Výpis kódu 4.8 Získání dat jako n-tice

```
val cb: CriteriaBuilder = this.entityManager.criteriaBuilder
val cq: CriteriaQuery<Tuple> = cb.createTupleQuery()
val root: Root<ENTITY> = cq.from(metadata.cls)

cq.multiselect(
*validatedGroup.map { root.get<Any>(it) }.toTypedArray(),
*validatedAggregation.map {
when (it.second) {
    AggregationType.AVG -> cb.avg(root.get(it.first))
    AggregationType.MIN -> cb.min(root.get(it.first))
    AggregationType.MAX -> cb.max(root.get(it.first))
    AggregationType.COUNT -> cb.count(root.get<Any>(it.first))
    }
}.toTypedArray()
)
```

Jelikož jsem chtěl podporovat různé pořadí získaných sloupců, různý počet získaných sloupců a nebyl nucen vymýšlet nějaké způsoby jako to zprovoznit s konstruktorem, tak jsem vybral jako variantu nejmenšího odporu dotaz na n-tice.

4.3 Klientská část

Klientská část byla vytvořená v knihovně React2.4.2. Jedná se o SPA2.1.1 webovou aplikaci, která se otevře při zadání příslušného URL.

Na začátku otevření aplikace se zobrazí úvodní stránka, do které se načtou semestry, fakulty a metadata reportů. Semestry a fakulty se používají pro filtrování. Z metadat se vygenerují odkazy.

Stránky reportů jsou implementované jako jedná stránka s dynamickým url parametrem, který obsahuje odkaz na příslušný kontroler. Data spojené s stránkou jsou uložená v rámci Redux Toolkit Slice2.4.4.

Výpis kódu 4.9 Formát uložení dat reportu

```
export const createFilterInputSchema =
<K extends z.ZodTypeAny>(cols: K) => {
    return z.object({
        type: z.nativeEnum(InputTypeV2),
        min: z.optional(z.coerce.number()),
        max: z.optional(z.coerce.number()),
        scaling: z.optional(z.coerce.number()),
        inputType: z.optional(z.nativeEnum(DateFindInputType)),
        start: z.optional(z.coerce.string()),
        end: z.optional(z.coerce.string()),
        value: z.optional(z.coerce.string()),
        options: z.optional(z.array(z.object({
            tag: z.coerce.string(),
            tagAliasCzech: z.coerce.string(),
            tagAliasEnglish: z.coerce.string()
        }))),
    })
}

export const createInputSchema = <K extends z.ZodTypeAny>(cols: K) => {
    return z.object({
        query: z.coerce.string(),
        expression: z.array(createFilterInputSchema(cols)),
        semesterCode: z.array(z.coerce.string()),
        facultyId: z.array(z.coerce.string()),
        hinted: z.optional(cols),
        hintQuery: z.coerce.string()
    }).extend(PagingInputSchema.shape)
}

const ReportFormSliceSchema = z.object({
    args: FilterInputSchema,
    metadata: z.record(z.string(), ColumnDataSchema),
    columnView: z.nativeEnum(ColumnView),
    path: z.string()
})

type ReportFormSlice = z.infer<typeof ReportFormSliceSchema>
```

Na ukázce je vidět celý formát dat uložených na straně klienta. Poslední objekt v ukázce kódu je ten nejdůležitější, reprezentuje strukturu stavu stránky reportu, formát je napsaný při použití knihovny Zod, důvodem je možnost validace podle schématu za běhu aplikace, to je vyžadováno při získávání stavu z url, nebo z local-storage, protože uživatel tam může provést změny narušující strukturu očekávanou aplikací.

1. **args:** filtry reportu
2. **metadata:** metadata aktuálně zobrazeného reportu
3. **columnView:** určuje jazyk zobrazených sloupců
4. **path:** cesta ke kontroleru reportu

První objekt v ukázce reprezentuje formát filtru hlavního vyhledávacího pole2.7.12, jelikož Redux Toolkit Slice2.4.4 povoluje uložení dat pouze jako objekt a není povolená dědičnost jako u tříd, tak jsem musel přidat do definice všechny možné proměnné týkající všech filtrů, jako nepovinné.

Druhý objekt reprezentuje formát filtrování dat na straně klienta.

1. **query:** textový všeobecný dotaz použitý při full-textovém vyhledávání.
2. **expression:** výraz vytvořený hlavním filtrovacím. polem2.7.12
3. **semesterCode:** pole obsahující id fakulty, což je jedno z polí nutně obsažených v reportu.
4. **facultyId:** další pole nutně obsažené v reportu.
5. **hinted:** sloupec, který žádá o nápovědy pro uživatele,
6. **hintQuery:** dotaz určený k vyhledání nápověd.

Filtry se při přechodu na stránku reportů mohou načíst buď z URL, z local-storage, nebo se nastaví na výchozí prázdné. Největší prioritu má URL, potom local-storage a jako poslední možnost je výchozí stav, data se v local-storage verzují, takže když se dohodne nějaká změna formátu, tak vývojář může data zneplatnit a vynutit, aby uživatelé začali používat nový formát. Také při přechodu na stránku s reporty po průzkumu se nastaví aktuální semestr jako výchozí filtr, to se vykoná pomocí `extrareducers` nabízených knihovnou Redux-Toolkit2.4.4, které poslouchají, jestli byl požadavek na získání semestrů úspěšně proveden.

Algoritmus pro správné načtení informací pro report je náročnější než kontrola, jestli mám data v url nebo local-storage, jelikož je stránka univerzální a úložiště reporty sdílejí, tak při změně stránky na jiný report je potřeba aktuální data vyměnit, aby se načetl správný report. Samotný proces jsem zakreslil pomocí diagramu aktivit pro lepší pochopení.

Dále bylo potřeba vyřešit implementaci tabulky tak, aby podporovala libovolný počet sloupců všech dostupných již zmíněných typů. Proto jsem použil funkci Javascriptu `Object.keys` na metadata a následně jsem použil funkci map, která iteruje přes všechny klíče a vytvořil jsem vhodný počet sloupců. Zda je sloupec zobrazený určuje atribut `selected` v metadatech sloupce a hodnota `columnView` uložená v Redux-Toolkit slice stránky reportu. Hlavní filtrovací pole funguje podobně jako tabulka, kde se také předají všechny klíče z metadat.

Výpis kódu 4.10 Formát uložení dat reportu

```
{Object.keys(metadata)
.filter((key) => isColumnSelected(key)).map((col, idx) => (
    <Fragment key={String(col)}>
        ...
    </Fragment>
))}
```

Zajímavé je tak řešení resetování stránkování při změně filtru, jelikož je velké množství různých akcí 2.4.4, tak by bylo potřeba do každé nakopírovat kód nastavující stránku na nulu, já jsem místo toho použil `extrareducers` 2.4.4, které poslouchají, jestli nenastane akce měnící filtry, pokud ano, tak se resetuje číslo stránky.

Výpis kódu 4.11 Formát uložení dat reportu

```
builder.addMatcher(isAnyOf(
    reportFormSlice.actions.resetColumns,
    reportFormSlice.actions.resetSemester,
    reportFormSlice.actions.resetFaculty,
    reportFormSlice.actions.onChangeQuery,
    reportFormSlice.actions.updateChoiceArray,
    reportFormSlice.actions.changePageSize,
    reportFormSlice.actions.unpage,
    reportFormSlice.actions.sort,
    reportFormSlice.actions.addBasicExpression,
    reportFormSlice.actions.addBasicExpression,
    reportFormSlice.actions.addSelectExpression,
    reportFormSlice.actions.addYesNoExpression,
    reportFormSlice.actions.addExpression,
    reportFormSlice.actions.updateExpression,
    reportFormSlice.actions.removeExpression,
    reportFormSlice.actions.initMetadata
    ), (state) => {
    resetPagingAcition(state)
})
```

Poslední zajímavostí na straně klienta jsou vizualizace. Již zmíněné nástroje pro reporting podporují vizualizace, proto jsem se rozhodl alespoň přidat minimalistickou podporu pro ně a zároveň díky tomu, že můj projekt je méně obecný, přidat něco navíc, co již zmíněné nástroje nepodporují.

Aktuálně aplikace podporuje dva typy vizualizací – sloupcový graf a čárový graf, které jsem považoval za nejužitečnější pro zobrazení. Pro tvorbu grafu slouží již zmíněná anotace, aktuálně je možnost se vyjádřit omezenější, protože formát je napsaný tak, aby v něm bylo možné vytvořit vizualizaci snadno a rychle.

Důležité pro pochopení, jak se dají vytvořit formáty na vizualizace, je porozumět, jaká data graf potřebuje v jakém formátu, a také jak funguje SQL.

Čárový a sloupcový graf jsou podobné grafy vzhledem k jejich potřebám na data, v obou případech máme body na x-ové ose a y-ové ose, v případě čárového grafu se všechny body propojí křivkou. U sloupcového grafu se zobrazí jako sloupce, ale každý se dá využít v jiných případech.

Zjednodušeně by se tvorba grafů dala popsat jako provedení operace `group by` nad určitými sloupci a agregace nad jinými sloupci. Vyjadřovací schopnost mého formátu je podobná grafickému rozhraní Apache Supersetu, ale moje implementace obsahuje navíc propracovanější formátování popisků.

1. **each:** určuje sloupce, které se použijí pro tvorbu více sloupců/čar. Pokud `each` obsahuje sloupce `facultyId` a `isCourseSurvey`, tak se vytvoří čárové grafy pro všechny kombinace sloupců.

2. **`x`:** specifikuje sloupce použité v rámci klauzule `group by`, sloupce s parametru `each` se k nim přidají. Hodnoty sloupců jsou zobrazené na x-ové ose. pokud není žádný specifikovaný sloupec v `each`, tak se sloupce v x chovají, jako kdyby byly součástí `each`. Sloupce v `each` se nesmějí nacházet v `x`.

3. **`y`:** určuje metriky, které se zobrazí na grafu. Například: `fillRate:AVG` najde průměr sloupce `fillRate`, metrik je možné zobrazit neomezené množství. Aktuálně jsou podporované agregační operace, minimum, maximum, průměr a počet. Každý sloupec může mít až všechny čtyři metriky najednou. Není možné mít sloupce v `y` a zároveň v `x` nebo `each`.

4. **additional:** v tomto poli se specifikují další metriky, které nemají být zobrazené v grafu, pomocí toho sloupce lze obejít omezení SQL databází povolující vracet sloupce, které používají agregační funkci, nebo se nacházejí v klauzuli `group by`, na sloupec se aplikuje agregace `min`.
5. **formatX a formatTickX:** formátování je jedna z výhod mojí aplikace oproti ostatním, pomocí těchto parametrů lze specifikovat zobrazený formát pro x-ovou souřadnici. O definici formátu se zmíním později.
6. **formatY a formatTickY:** definují formátování na y-ové ose, jelikož jsou podporovány aktuálně pouze sloupcový a čárový graf, tak mám podporu pro formátování čísel. Aktuálně lze specifikovat počet desetinných míst a také lze přidat na příklad značku procenta k číslu, abychom uživateli dali najevo o jakou jednotku se jedná.
7. **labelXCzech, labelYCzech, labelXEnglish, labelYEnglish:** dávají možnost označit osy.
8. **sort:** zde lze specifikovat řazení využité při tvorbě vizualizace, například pokud bychom chtěli, aby se vizualizace řadila podle průměrné vyplněnosti, můžeme zadat `fillRate:AVG,DESC`.
9. **titleCzech, titleEnglish:** specifikují název vizualizace.
10. **limit:** určuje, kolik záznamů se získá z databáze pro vytvoření vizualizace.

Následně bych rád zmínil jak funguje formátování a jeho využití. Na ukázce je vidět základní princip formátování, proměnné obsažené v řádku výsledku je možné využít pomocí syntaxe, kde proměnná je ohraničená závorkami předcházenými zavináčem. Pokud proměnná končí zavináčem znamená, že se jedná o překládanou proměnnou a musí se dodržet určitá pravidla, že proměnné končí na suffix Czech, pokud je v českém jazyce a suffix English, pokud je v angličtině. Jestliže proměnná obsahuje již zmíněné `options` předefinující pojmenování určitých hodnot, tak se využije předefinování.

Výpis kódu 4.12 Ukázka formátování 1

```
formatX = "@{facultyName@}-@{isCourseSurvey}",
```

Pokud se specifikuje hvězdička místo názvu proměnné, tak se využije text obsažený v označení, což lze využít například v případě, že máme sloupcový graf s agregací nad jedním sloupcem.

Výpis kódu 4.13 Ukázka formátování 2

```
formatX = @{*},
```

Zde je možné vidět formátování čísla na nula desetinných míst a zakončení znakem procenta. Aktuálně pro y sloupce se ignoruje použitý text a formátuje se pouze číslo, protože jsem neviděl potřebu toho podporovat víc.

Výpis kódu 4.14 Ukázka formátování 3

```
tickFormatY = "@{label:0}␣%",
```

Pro tvorbu vizualizací byla použita knihovna Recharts 2.5.8, speciálně vytvořená pro využití v rámci Reactu, využívající skládací způsob tvorby grafů, na rozdíl od běžně známých knihoven jako je Chart-js. Pro moje potřeby jsem nadefinoval obecné komponenty pro sloupcový a čárový graf, aby podporovaly všechny zmíněné konfigurace. Nadefinoval jsem, také vlastní komponent nápovědy.

Knihovna Recharts 2.5.8 vyžaduje data ve specifickém formátu, proto jsem vytvořil funkci transformující data do vhodného formátu, protože je to náročná operace, tak pro optimaliazci používám hook 2.4.2.4 `useMemo`, který si pamatuje výstup napříč překreslováními.

Také jsem definoval vlastní parsery zpracovávající vlastní formátování.

4.4 Vizualizace a reporty

Další součástí mojí práce bylo vytvořit požadované reporty v nově implementovaném systému. Toto díky snadnému rozhraní na implementaci reportů bylo velice snadné, což je za mě velice pozitivní hodnocení pro můj systém, stačilo vytvořit entitní třídy potřebných pohledů, případně vytvořit identifikační třídy, pokud se identifikátor skládal z více částí a vhodně vše anotovat podle potřeb.

4.4.1 Vyplněnost předmětu - celková

Přidal jsem nový report, který zobrazuje data o vyplněnosti předmětu ze všech semestrů, aktuálně zpřístupňuje data pouze z již ukončených semestrů.

Pro report jsem vytvořil tři vizualizace. Porovnání vyplněnosti předmětu během semestrů pomocí čárového grafu. Průměrnou vyplněnost předmětu vizualizovanou sloupcovým grafem a průměrnou vyplněnost předmětu podle katedry a semestru, také vizualizovanou sloupcovým grafem. 4.3 4.4

4.4.2 Vyplněnost předmětu a vyučující - celková

Další nový report, který jsem přidal, zobrazuje data o vyplněnosti předmětu ze všech semestrů. Tento report jako předchozí zobrazuje pouze data z již ukončených anket.

Tento report má také tři vizualizace. Počet předmětů, které vyučující vyučoval jako sloupcový graf. Vyučující s nejlépe hodnocenými předměty v průměru také jako sloupcový graf a vyplněnost předmětu podle vyučujících vizualizovaná pomocí sloupcového grafu.

4.4.3 Vyučující a předměty

Další report je vyučující a jejich předměty, tyto data se získávají v reálném čase, tak je možně vidět data i z aktuální ankety, tento report patří mezi původní reporty zobrazené v předchozím systému.

4.4.4 Vyplněnost předmětů

Vyplněnost předmětů, podobné jako report s vyučujícími a předměty zobrazuje informace z aktuálního semestru. Nevýhodou reportu je, že neobsahuje informace o vyplnění z předchozích semestrů.

4.4.5 Vyplněnost anket

Vyplněnost anket patří mezi nově přidané reporty a slouží k porovnání vyplněnosti napříč všemi semestry. Tento report obsahuje čtyři vizualizace.

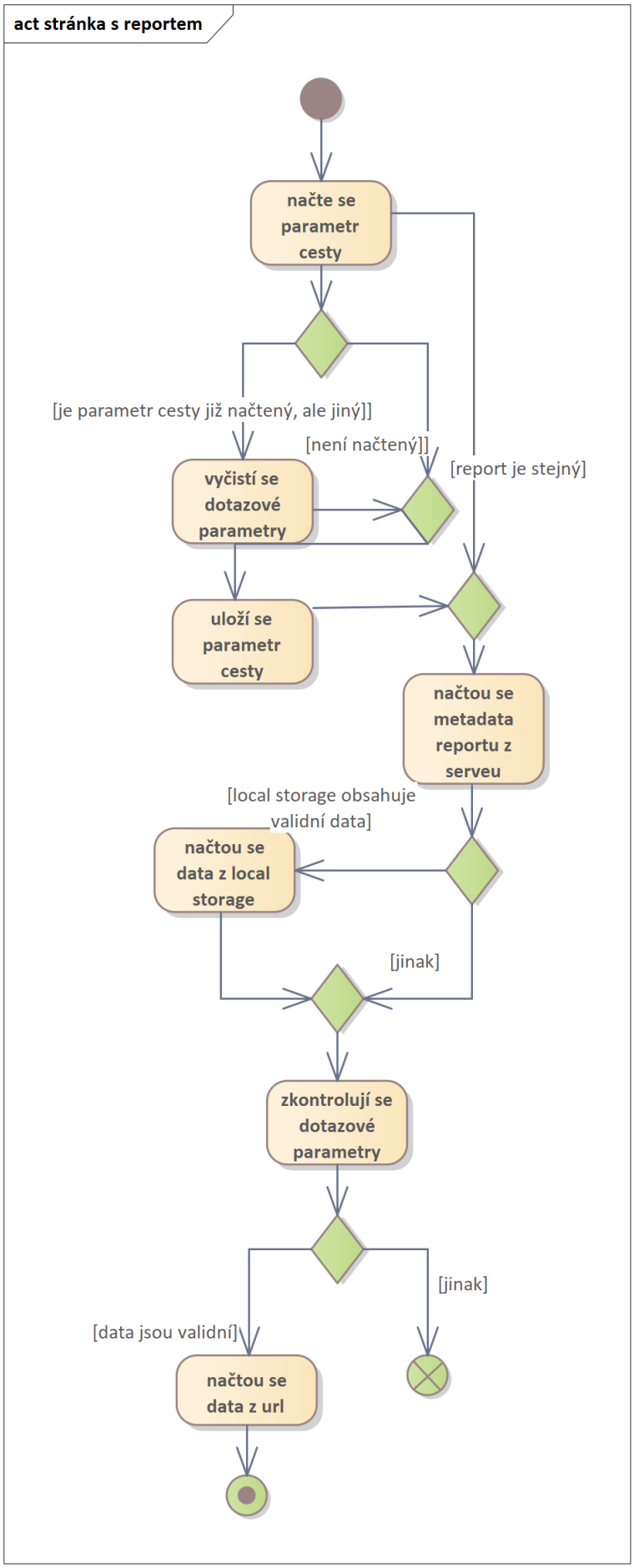
První vizualizace porovnává vyplněnosti anket napříč semestry pomocí čárového grafu. Druhá vizualizace porovnává počet lístků, třetí vizualizace porovnává počet vyplněných lístků a čtvrtá poslední vizualizace kombinuje dvě předchozí vizualizace do jednoho grafu. 4.5

4.4.6 Anketní události

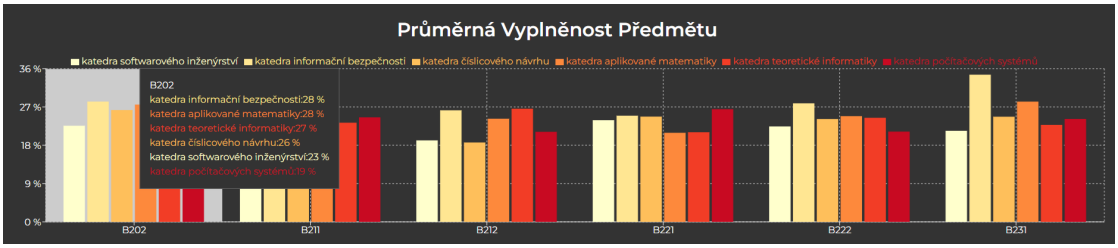
Anketní události je další z nových reportů, který obsahuje informace o stavu ankety a kdy se bude měnit. Pro report jsem nevytvořil žádné vizualizace, jelikož tam nebylo nic vhodného na vizualizaci.

4.4.7 Anketní otázky

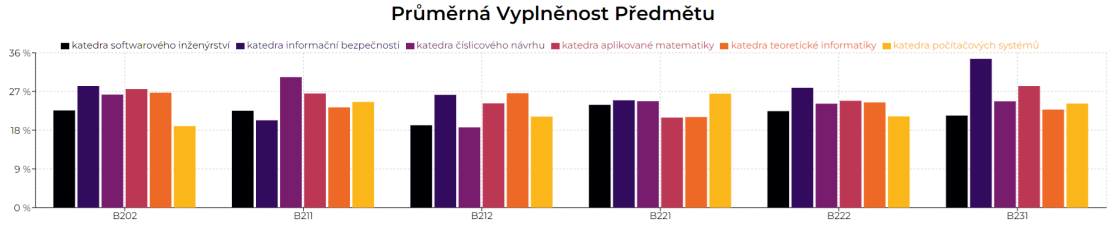
Poslední report je report anketních otázek, obsahující informace o otázkách v jednotlivých anketách, tento report neobsahuje žádné vizualizace. Rozdílem oproti ostatním reportům je možnost prohledávání otázek jednotným vyhledávacím polem.



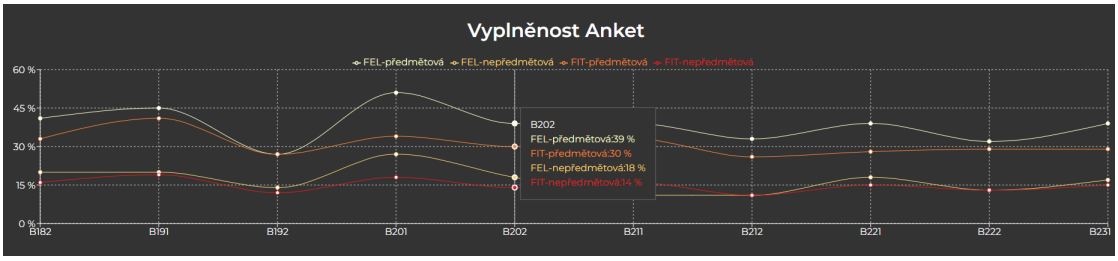
Obrázek 4.2 Diagram načtení stránky



Obrázek 4.3 Sloupcový graf tmavý



Obrázek 4.4 Sloupcový graf světlý



Obrázek 4.5 Sloupcový graf světlý

Kapitola 5
Testování

5.1 Testovaní dotazů na backendu

Pro testovaní na backendu používám testovací framework Junit. Osobně využívám několik druhů testů. Jednotkové testy a integrační testy.

Jednotkové testy se zaměřují na testování nedělitelných jednotek kódu, například funkce. V případě, že funkce využívá jiné části systému, tak je potřeba mockovat. Mockování je programově předem nadefinované chování objektu bez nutnosti využití reálné instance objektu/funkce. Tento typ testování využívám například pro parser dotazu.

Dále mám v aplikaci definované integrační testy testující repositáře propojené s databází. V mém případě jsem se rozhodl použít H2 databázi, což je SQL databáze v paměti. Databázi H2 je velice jednoduché využívat v rámci Gitlab pipeline, jediná nevýhoda v mém případě je, že nelze testovat specifické funkcionality databáze Oracle.

Pro zkoušení specifických funkcí databáze Oracle jsem si vytvořil Docker kontejner, ale kvůli svojí velikosti a náročnosti na konfiguraci jej není snadné použít v rámci CI/CD.

5.2 Testování na straně klienta

Na straně klienta používám k testování testovací framework Vitest. Na straně klienta hlavně používám jednotkové testy na komplexnější funkce, kde může dojít k chybám.

5.3 Uživatelské testování

Uživatelské testování probíhalo na konzultacích, jelikož vedoucí práce patří mezi uživatele aktuálního reporting systému, tak jsem vždy předvedl, co jsem měl aktuálně vytvořené a tak jsem mohl průběžně získávat informace o tom, co je potřeba změnit a zlepšit. Aplikace je velice jednoduchá, hlavní funkcionalitou je vyhledávaní, a proto nebylo potřeba vytvářet žádné složité scénáře. Aplikace má další funkcionality, ale není náročná z uživatelského hlediska na pochopení a použití.

Kapitola 6
Závěr

Cílem této práce bylo analyzovat aktuální tvorbu reportů Anketa ČVUT, shromáždit požadavky na další typy reportů, navrhnout nový systém s novou vizuální podobou pro zobrazování původních a nových reportů, implementovat jej, řádně otestovat a nasadit na školní infrastrukturu.

Nový systém má výrazně vylepšenou vizuální podobu a na rozdíl od předchozí verze podporuje angličtinu. Nově implementované funkcionality umožňují v rámci rozhraní exportovat data, podporují napovídání při psaní textových filtrů. Nově lze data procházet data napříč semestry a je podporováno zobrazení dat z více fakult najednou. Dále jsem vytvořil vylepšený filtrovací systém, který nabízí filtrovaní podle typu sloupce. Také lze volit zobrazené sloupce podle jazyka, nebo manuálním výběrem. Hlavním přínosem je, že se již reporty negenerují periodicky pomocí skriptu, ale data jsou získávaná v reálném čase z databáze.

Jako možné vylepšeni do budoucna bych chtěl přidat administrátorské rozhraní pro vytváření dalších reportů. Další novou funkcionalitu, kterou bych chtěl přidat, je podpora nových formátů v rámci exportu.

Domnívám se, že nová aplikace výrazně usnadní práci s reporty Ankety ČVUT.

Funkcionality	Anketa aktuální	Anketa v2	Moje práce
stránkování			✓
řazení			✓
export dat			✓
uložení filtrů v local storage			✓
zobrazení všech dat			✓
zobrazení dat z aktuálního semestru	✓	✓	✓
api pro získání dat		✓	✓
snadná rozšiřitelnost			✓
možnost nasadit	✓		✓
zobrazení dotazu v url			✓
podpora více jazyků		✓	✓

Tabulka 6.1 Tabulka porovnání funkcionalit

Bibliografie

1. GLÖCKNER, Sara. *Portal Systems - Reporting* [online]. 2022. [cit. 2024-04-24]. Dostupné z: https://www.portalsystems.de/en/wiki/reporting/.
2. *Superset - The Apache Software Foundation* [online]. 2023. [cit. 2024-04-24]. Dostupné z: https://superset.apache.org/docs/faq/.
3. ELASIC [online]. [B.r.]. [cit. 2024-04-24]. Dostupné z: https://www.elastic.co/guide/en/kibana/current/index.html.
4. VALENTA, Michal. *Anketa Reports* [online]. 2024. [cit. 2024-04-24]. Dostupné z: https://anketa.cvut.cz/reports/stav/stav_anketa_muvs.html.
5. SANICKÁ, Kristýna. *Anketa Reports* [online]. 2024. [cit. 2024-04-24]. Dostupné z: https://gitlab.fit.cvut.cz/anketa-cvut-v3/reports-frontend.
6. VALENTA, Šimon. *Anketa Reports* [online]. 2024. [cit. 2024-04-24]. Dostupné z: https://gitlab.fit.cvut.cz/anketa-cvut-v3/reports-backend.
7. ORACLE. *Oracle* [online]. [B.r.]. [cit. 2024-04-24]. Dostupné z: https://www.oracle.com/database/.
8. BERNHOUER, David. *Architektury - BI-TWA.21* [online]. 2022. [cit. 2024-04-24]. Dostupné z: https://courses.fit.cvut.cz/BI-DBS/materials/slides/pres-les10-fyzicke-ulozeni-dat.pdf.
9. OLUWATOSIN, Haroon Shakirat. Client-server model. *IOSR Journal of Computer Engineering*. 2014, roč. 16, č. 1, s. 67–71.
10. ORACLE. *Oracle levenshtein* [online]. [B.r.]. [cit. 2024-04-24]. Dostupné z: https://docs.oracle.com/en/database/oracle/oracle-database/19/arpls/UTL_MATCH.html#GUID-E7017694-4DE1-4692-B61E-8CFEEA5FF6B8.
11. ELASTIC. *Elastic Levenshtein* [online]. [B.r.]. [cit. 2024-04-24]. Dostupné z: https://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl-fuzzy-query.html.
12. VALENTA, Michal. *Fyzický pohled na data a vývoj aplikací nad databází* [online]. 2022. [cit. 2024-04-24]. Dostupné z: https://courses.fit.cvut.cz/BI-DBS/materials/slides/pres-les10-fyzicke-ulozeni-dat.pdf.
13. ORACLE. *Indexing with Oracle Text* [online]. [B.r.]. [cit. 2024-04-24]. Dostupné z: https://docs.oracle.com/en/database/oracle/oracle-database/19/arpls/UTL_MATCH.html#GUID-E7017694-4DE1-4692-B61E-8CFEEA5FF6B8.
14. ORACLE. *Oracle text* [online]. [B.r.]. [cit. 2024-04-24]. Dostupné z: https://www.oracle.com/database/technologies/appdev/oracletext.html.

15. ORACLE. *DBMS_SEARCH Package* [online]. [B.r.]. [cit. 2024-04-24]. Dostupné z: https://docs.oracle.com/en/database/oracle/oracle-database/23/ccref/dbms_search-package.html#GUID-7916E58F-9C8B-457D-A23C-37AF4904F51F.
16. ORACLE. *Oracle contains* [online]. [B.r.]. [cit. 2024-04-24]. Dostupné z: https://docs.oracle.com/en/database/oracle/oracle-database/19/ccref/oracle-text-CONTAINS-query-operators.html.
17. ORACLE. *Oracle like* [online]. [B.r.]. [cit. 2024-04-24]. Dostupné z: https://docs.oracle.com/cd/B13789_01/server.101/b10759/conditions016.htm.
18. FUSE.JS. *Fuse.js* [online]. [B.r.]. Dostupné také z: https://www.fusejs.io.
19. ELASTIC. *Elastic* [online]. [B.r.]. [cit. 2024-04-24]. Dostupné z: https://www.elastic.co/guide/index.html.
20. MICROSOFT. *Typescript* [online]. [B.r.]. [cit. 2024-04-24]. Dostupné z: https://www.typescriptlang.org/.
21. META. *React* [online]. 2024. [cit. 2024-04-24]. Dostupné z: https://react.dev/.
22. MOZILLA. *Introduction to the DOM* [online]. 2023. [cit. 2024-04-24]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction.
23. COMMUNITY, React. *React Reconciliation* [online]. [B.r.]. Dostupné také z: https://legacy.reactjs.org/docs/reconciliation.html.
24. COMMUNITY, React. *Lifecycle of Reactive Effects* [online]. [B.r.]. Dostupné také z: https://react.dev/learn/lifecycle-of-reactive-effects.
25. *Higher-Order Components* [online]. 2024. [cit. 2024-04-24]. Dostupné z: https://legacy.reactjs.org/docs/higher-order-components.html.
26. *Tailwindcss* [online]. 2024. [cit. 2024-04-24]. Dostupné z: https://tailwindcss.com/docs/installation.
27. MOZILLA. *CSS Syntax* [online]. 2023. [cit. 2024-04-24]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/CSS/Syntax.
28. *Redux Toolkit* [online]. 2023. [cit. 2024-04-24]. Dostupné z: https://redux-toolkit.js.org/introduction/getting-started.
29. META. *Flux architektura* [online]. 2023. [cit. 2024-04-24]. Dostupné z: https://facebookarchive.github.io/flux/docs/in-depth-overview/.
30. KARRYS, Luke et al. *https://docs.npmjs.com/about-npm* [online]. 2023. [cit. 2024-04-24].
31. WILLIAMSON, Erik et al. *https://docs.npmjs.com/cli/v10/configuring-npm/package-lock-json* [online]. 2023. [cit. 2024-04-24].
32. *react-i18next documentation* [online]. 2024. [cit. 2024-04-24]. Dostupné z: https://react.i18next.com/.
33. ESLINT. *https://eslint.org/* [online]. 2023. [cit. 2024-04-24].
34. JETBRAINS. *Kotlin docs* [online]. 2024. [cit. 2024-04-24]. Dostupné z: https://kotlinlang.org/docs/home.html.
35. VMWARE. *The IoC container* [online]. 2024. [cit. 2024-04-24]. Dostupné z: https://docs.spring.io/spring-framework/docs/3.2.x/spring-framework-reference/html/beans.html.
36. VMWARE. *Spring Beans* [online]. 2024. [cit. 2024-04-24]. Dostupné z: https://docs.spring.io/spring-framework/reference/core/beans/introduction.html.
37. FOWLER, Martin. *Work of unit* [online]. [B.r.]. [cit. 2024-04-24]. Dostupné z: https://martinfowler.com/eaaCatalog/unitOfWork.html.

38. KING, Gavin. *A Guide to Hibernate Query Language* [online]. 2024. [cit. 2024-04-24]. Dostupné z: https://docs.jboss.org/hibernate/orm/6.5/querylanguage/html_single/Hibernate_Query_Language.html.
39. KING, Gavin. *Criteria* [online]. 2024. [cit. 2024-04-24]. Dostupné z: https://docs.jboss.org/hibernate/orm/6.4/userguide/html_single/Hibernate_User_Guide.html#criteria.
40. *Gradle* [online]. 2024. [cit. 2024-04-24]. Dostupné z: https://docs.gradle.org/current/userguide/userguide.html?_gl=1*1barmnh*_ga*MTg1NDE4NDIwNi4xNzE0MTIzNDI1*_ga_7W7NC6YNPT*MTcxNTc4MzUwNy4xMS4xLjE3MTU3ODM4NzguMzcuMC4w.
41. CHMEL, Olexandr. *Anketa ČVUT* [online]. [B.r.]. [cit. 2024-04-24]. Dostupné z: https://dspace.cvut.cz/handle/10467/113754.
42. *Gradle vs Maven* [online]. 2024. [cit. 2024-04-24]. Dostupné z: https://gradle.org/maven-vs-gradle/.
43. MAILEN, Jeremy. *Kotliner* [online]. 2024. [cit. 2024-04-24]. Dostupné z: https://github.com/jeremymailen/kotlinter-gradle.
44. DETEKT. *Detekt* [online]. 2024. [cit. 2024-04-24]. Dostupné z: https://detekt.dev/.
45. JETBRAINS. *Kotlin Symbol processing* [online]. [B.r.]. [cit. 2024-04-24]. Dostupné z: https://kotlinlang.org/docs/ksp-overview.html.
46. JETBRAINS. *Kotlinpoet* [online]. [B.r.]. [cit. 2024-04-24]. Dostupné z: https://square.github.io/kotlinpoet/.
47. GROUP, Recharts. *BarChart* [online]. 2024. [cit. 2024-04-24]. Dostupné z: https://recharts.org/en-US/api/BarChart.
48. *reactchartjs/react-chartjs-2: vertical - CodeSandbox* [online]. 2024. [cit. 2024-04-24]. Dostupné z: https://codesandbox.io/p/devbox/reactchartjs-react-chartjs-2-vertical-jebqk?embed=1&file=%2FApp.tsx.
49. FIGMA. *5 essential UI design principles—and how to use them* [online]. [B.r.]. [cit. 2024-04-24]. Dostupné z: https://www.figma.com/resource-library/ui-design-principles/.
50. ČVUT. *Grafický manuál ČVUT* [online]. [B.r.]. [cit. 2024-04-24]. Dostupné z: https://www.cvut.cz/logo-a-graficky-manual.
51. OTTO, Mark et al. *Bootstrap colors* [online]. 2023. [cit. 2024-04-24]. Dostupné z: https://getbootstrap.com/docs/5.3/utilities/colors/#colors.
52. MUI. *Material UI colors* [online]. 2024. [cit. 2024-04-24]. Dostupné z: https://mui.com/material-ui/customization/palette/.
53. TAILWIND. *Customizing colors* [online]. 2024. [cit. 2024-04-24]. Dostupné z: https://tailwindcss.com/docs/customizing-colors.
54. TOOLS, UX [video]. UX Tools, 2022 [cit. 2024-04-24]. Dostupné z: https://www.youtube.com/watch?v=yYwEnLYT55c.
55. MALEVICZ. *Dark Mode UI Course 1* [video]. 2021. [cit. 2024-04-24]. Dostupné z: https://www.youtube.com/watch?v=CZqcnxLd978.
56. PETERSEN, Kai; WOHLIN, Claes; BACA, Dejan. The waterfall model in large-scale development. In: *Product-Focused Software Process Improvement: 10th International Conference, PROFES 2009, Oulu, Finland, June 15-17, 2009. Proceedings 10*. Springer, 2009, s. 386–400.
57. ATTLASIAN. *Gitflow workflow* [online]. 2024. [cit. 2024-04-24]. Dostupné z: https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow.

58. GITLAB. *Gitlab* [online]. 2024. [cit. 2024-04-24]. Dostupné z: https://about.gitlab.com/get-started/.

59. APACHE. *Apache HTTP server project* [online]. 2024. [cit. 2024-04-24]. Dostupné z: https://httpd.apache.org/.

60. APACHE. *Apache Tomcat* [online]. 2024. [cit. 2024-04-24]. Dostupné z: https://tomcat.apache.org/.

61. VMWARE. *Spring Boot instalace* [online]. 2024. [cit. 2024-03-21]. Dostupné z: https://docs.spring.io/spring-boot/docs/current/reference/html/deployment.html#deployment.installing.

Obsah příloh

- `readme.txt` ..... stručný popis obsahu média
- `src`
  - `impl` ..... zdrojové kódy implementace
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF