VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

MODERNÍ ARCHITEKTURY WEBOVÝCH APLIKACÍ

MODERN WEB APPLICATION ARCHITECTURES

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. PAVOL MALÍK**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. RADEK BURGET, Ph.D.**
SUPERVISOR

BRNO 2019

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav informačních systémů (UIFS) Akademický rok 2018/2019

Zadání diplomové práce

Student: **Malík Pavol, Bc.**
Program: Informační technologie Obor: Informační systémy
Název: **Moderní architektury webových aplikací**
Modern Web Application Architectures
Kategorie: Informační systémy

Zadání:

1. Seznamte se se současnými způsoby návrhu webových aplikací klient-server se zaměřením na aplikace s tlustým klientem (rich client).
2. Prostudujte a srovnejte technologie pro tvorbu webových aplikačních rozhraní. Zaměřte se na technologie GraphQL a REST. Zmapujte podporu GraphQL na různých serverových i klientských platformách.
3. Po dohodě s vedoucím zvolte vhodnou aplikační doménu a navrhněte architekturu aplikace využívající technologie GraphQL pro aplikační rozhraní.
4. Zvolte vhodné technologie a implementujte serverovou část navržené aplikace s aplikačním rozhraním GraphQL a klientskou část s využitím webových technologií.
5. Proveďte testování vytvořené aplikace
6. Zhodnoťte dosažené výsledky, zejména pak výhody a nevýhody technologie GraphQL oproti používaným alternativám.

Literatura:

- Banks, A., Porcello, E.: Learning GraphQL, O'Reilly Media, Inc., 2018
- Žára, O.: JavaScript - Programátorské techniky a webové technologie, Computer Press, 2015

Při obhajobě semestrální části projektu je požadováno:

- Body 1 až 3

Podrobné závazné pokyny pro vypracování práce viz http://www.fit.vutbr.cz/info/szz/

Vedoucí práce: **Burget Radek, Ing., Ph.D.**
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2018
Datum odevzdání: 22. května 2019
Datum schválení: 30. října 2018

Abstrakt

Táto práca oboznamuje čitateľa so základnými spôsobmi návrhu klient-server aplikácií a technológiami pre tvorbu aplikačných programových rozhraní (API). Porovnáva najmä moderné prístupy – REST a GraphQL, ktoré sa v poslednom čase stali najpopulárnejšími. Výsledným produktom práce je webová aplikácia pre osobný rozvoj zamestnancov vo firme zaoberajúcej sa vývojom *softwaru*. Aplikácia poskytuje funkcie ako správu oblastí osobného rozvoja, či plánovanie stretnutí. Samozrejmosťou je správa zamestnancov, registrácia a prihlasovanie používateľov. Aplikácia využíva technológiu GraphQL pre aplikačné rozhranie, programovací jazyk Java spolu so Spring Boot pre implementáciu serverovej časti a Angular pre klientskú časť. Pre autentizáciu a autorizáciu je použitý *framework* Spring Security, pre databázu bol zvolený systém MySQL.

Abstract

This thesis informs reader about main ways of client-server application design and technologies for building an Application Programming Interface (API). It compares mainly modern approaches – REST and GraphQL, which became recently popular. The final product of thesis is a web application for self-improvement of employees in software company. Application provides features like self-improvement activities management or planning of meetings. Employees management, user registration and log in are obvious. Application uses GraphQL technology as an application interface, Java programming language together with Spring Boot for server implementation and Angular for client implementation. Spring Security framework is used for authentication and authorization, MySQL was chosen for database part.

Kľúčové slová

klient-server, bohatý webový klient, osobný rozvoj, API, GraphQL, REST, Java, Spring, Spring Boot, Spring Security, Angular, MySQL

Keywords

client-server, rich web client, self-improvement, API, GraphQL, REST, Java, Spring, Spring Boot, Spring Security, Angular, MySQL

Citácia

MALÍK, Pavol. *Moderní architektury webových aplikací.* Brno, 2019. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Radek Burget, Ph.D.

Moderní architektury webových aplikací

Prehlásenie

Prehlasujem, že som túto diplomovú prácu vypracoval samostatne pod vedením pána Ing. Radka Burgeta, Ph.D. Uviedol som všetky literárne pramene a publikácie, z ktorých som čerpal.

.....
Pavol Malík
20. mája 2019

Poďakovanie

Moje poďakovanie patrí Ing. Radkovi Burgetovi, Ph.D. za podporu, odborné vedenie, konzultácie a poskytnuté rady pri tejto diplomovej práci. Tiež by som sa chcel poďakovať Vladovi Sudorovi za rady, konzultácie a uskutočnenie testovania v rámci firmy. Moja vďaka patrí aj rodine za podporu a trpezlivosť.

Obsah

Kapitola 1

Úvod

Webové aplikácie sa v poslednom desaťročí stali veľmi obľúbenými. Pomocou webového prehliadača, ktorý je dnes dostupný takmer na každom zariadení (počítač, mobil, tablet, smart TV a pod.), môžu koncoví používatelia odosielať e-maily, rezervovať letenky, kupovať či predávať tovar, počúvať hudbu, zdieľať a upravovať fotografie, chatovať, vytvárať konferenčné hovory a mnoho ďalších činností. Na to všetko stačí mať len webový prehliadač a zariadenie pripojené k internetu.

Spočiatku si používatelia webu vystačili so statickými stránkami a jednoduchými používateľskými rozhraniami. Avšak spolu s vývojom nových programovacích jazykov, webových prehliadačov, mobilných zariadení, výkonnejších procesorov rástli aj očakávania používateľov v interakcii a použiteľnosti webových aplikácií. Tieto požiadavky vyústili vo vývoj nových technológií a rozšírenie možností webových prehliadačov.

Do popredia sa dostali moderné architektúry aplikácií ovplyvnené prístupmi návrhu ako REST či GraphQL, ktoré si dávajú za úlohu určitým spôsobom uľahčiť komunikáciu medzi klientom a serverom. Akým spôsobom pracujú, ich výhody a nevýhody sú uvedené v rámci tejto práce. Pre demonštráciu technológie GraphQL je vytvorená webová aplikácia využívajúca jej hlavné prvky, ktorej postupný vývoj je v tejto práci opísaný. Aplikácia sa zameriava na osobný rozvoj zamestnancov v *softwarovej* firme a zahŕňa prvky ako plánovanie stretnutí či správu jednotlivých oblastí osobného rozvoja. Registrácia a prihlasovanie používateľov sú tiež jej súčasťou.

Jednou z hlavných bodov práce je oboznámiť čitateľa so súčasnými spôsobmi návrhu aplikácií typu klient-server, ktoré sú uvedené v kapitole 2. Zameriava sa najmä na typ bohatý webový klient, keďže výsledná aplikácia je práve tohto typu.

Kapitola 3 poskytuje základný prehľad webových aplikačných rozhraní a technológií pre ich tvorbu. Diskutuje najmä dve súčasné technológie, a to REST a GraphQL, ktoré následne porovnáva.

Kapitola 4 sa zaoberá špecifikáciou požiadaviek, ich analýzou, plánom vývoja a návrhom výslednej aplikácie, ktorý je rozdelený na viacero častí.

Implementácia aplikácie je kvôli zložitosti rozdelená na zvlášť kapitoly, ktoré sú organizované podobným spôsobom. Serverová (kapitola 5) a klientská (kapitola 6) časť aplikácie pozostávajú zo základného opisu technológií použitých pre vývoj, štruktúry, nastavenia technológie GraphQL, hlavných prvkov a nakoniec vybraných funkcií.

Prácu uzatvára kapitola pre testovanie aplikácie (7), kde sú uvedené postupy použité pri testovaní, vrátane záverečných testov vykonaných zamestnancami firmy, pre ktorú je aplikácia cielená, ako aj ďalšími námetmi na jej vylepšenie.

Kapitola 2

Spôsoby návrhu aplikácií klient-server

Posun od neinteraktívnych terminálových aplikácií k interaktívnym bohatým klientom zahŕňal nové programovacie jazyky, nové *hardwarové* platformy a väčšie vstup/výstup možnosti [13]. Vo svete vývoja webu bola evolúcia o niečo pomalšia, no aj tam došlo k posunu. Logika bola presunutá zo servera na klienta, čo umožnilo zvýšenú interakciu na strane klienta bez nutnosti neustálej komunikácie so serverom.

Myš ako vstupné zariadenie tiež zohrala dôležitú úlohu v evolúcii aplikácií. Pomocou nej boli používatelia schopní manipulovať s netextovými prvkami obrazovky, čo sa uplatnilo hlavne pri programoch pracujúcich s multimédiami (napr. *editory* obrázkov či videí). To, čo nebolo možné s terminálovými aplikáciami, sa stalo skutočným s príchodom bohatých klientov. Podobná evolúcia sa deje aj dnes: až doteraz bolo nereálne si predstaviť napr. úpravu obrázkov vo webovej aplikácii, pretože by to vyžadovalo neustále znovunačítanie stránok pri každej operácii. Avšak moderné webové aplikácie využívajú spracovanie aj na strane klienta, čím odbremeňujú server od práce a zároveň zvyšujú interaktivitu.

Táto kapitola opisuje jednotlivé typy klientov podľa hrúbky (inteligencie), ich výhody či nevýhody. Literatúra sa v tejto oblasti líši – klienti rovnakého typu teda môžu byť pomenovaní rôznymi názvami. Možno však rozlíšiť tri základné typy klientov:

- tenký klient – väčšina logiky je vykonávaná na serveri
- bohatý klient – väčšina logiky je vykonávaná u klienta
- bohatý webový klient – logika je rozdelená medzi klienta a server

2.1 Tenký klient

Aplikácie typu tenký klient (*thin client*) sú tiež známe ako webové aplikácie, webové používateľské rozhranie či internetové/intranetové aplikácie [13]. Jednou z typických definícií tenkého klienta je, že sa jedná o zariadenie, ktoré nemá lokálny pevný disk a prístup k dátam a aplikáciám získava od servera, cez sieť [1]. Rovnako sa dá nazvať aj kedysi používaný počítačový terminál, ktorý bol pripojený na centrálny server. Obecne však platí, že tenkí klienti sa spoliehajú na server, kde je vykonávaná väčšina ich výpočtov [15].

Vo svete webu bežia aplikácie typu tenký klient vo webovom prehliadači, z čoho plynie viacero obmedzení [18, 13]. Komponenty aplikácie nie sú vykresľované priamo na obrazovku

– namiesto toho sa pre vykreslenie používajú značkovacie jazyky HTML a CSS, ktorým prehliadač „rozumie“. Pôvodný koncept HTML neobsahoval multimédiá, ani nepodporoval vysoký stupeň interaktivity, takže väčšina týchto prvkov musela byť pridaná pomocou zásuvných modulov (*plug-ins*). Nebolo však možné určiť, či sú tieto moduly nainštalované u klienta a aká je ich presná funkcia.

Klasické webové aplikácie vo veľkej miere využívajú sieť, pretože prehliadač zobrazuje obrazovku vytvorenú na serveri, čo má za následok spomalenie interakcie medzi používateľom a aplikáciou. Navyše, celá obrazovka musí byť zakaždým vyžiadaná od servera. Hlavnou nevýhodou klientov tohto typu teda je, že nedokážu fungovať bez sieťového spojenia a sú ovplyvnení službami, ktoré sieťové spojenie poskytuje. Aj pri rýchlej sieti je prenesené veľké množstvo dát, čo redukuje výkon aplikácie.

O interakciu používateľa s aplikáciou sa opäť stará server, ktorý musí zabezpečiť vykreslenie požadovaného HTML a spracovať dáta potvrdené používateľom pomocou formulárov. Bežné interaktívne prvky ako ťahaj a pusti (*drag-and-drop*), naspäť a dopredu (*undo-redo*) a pod., nemusia byť dostupné, čo má negatívny vplyv na použiteľnosť aplikácie [18].

Výhodou webových aplikácií je multiplatformovosť[1]. Často môžu byť používané na rôznych zariadeniach ako mobiloch či tabletoch – v zásade všade, kde je nainštalovaný webový prehliadač. Ďalšou výhodou je, že citlivé dáta sú uložené len na serveri a iba časť z dát, ktoré používateľ potrebuje je prevedená z centrálneho úložiska cez sieť.

Kvôli obmedzenej funkčnosti sú tenkí klienti väčšinou používaní len na zobrazovanie dát, ako napr. telefónnych zoznamov či informácií o pobočkách. Tieto typy aplikácií nevyžadujú toľko funkčnosti, nakoľko o údržbu a aktualizácie dát sa zvyčajne starajú správcovia databáz.

2.2 Bohatý klient

Oproti tenkým klientom, bohatí klienti (*rich clients*) nebežia vo webovom prehliadači, ale natívne na operačnom systéme používateľa [13]. Alternatívnymi názvami bohatého klienta sú: hrubý klient (*thick client*), desktopová aplikácia, natívna aplikácia alebo tučný klient. Väčšina aplikácií tohto typu je napísaná v jazykoch Java, C++ alebo .NET. Typicky využívajú lokálny *hardware*, systémové zdroje ako pamäť, pevný disk a vstupné/výstupné zariadenia, ku ktorým majú neobmedzený prístup[2].

Hrubí klienti poskytujú rozsiahlu sadu prvkov optimalizovaných tak, aby fungovali pre rôzne prípady použitia. Často obsahujú oveľa viac prvkov ako používatelia v skutočnosti potrebujú pre svoju prácu. Vzhľad a mechanizmy týchto aplikácií sú väčšinou prispôsobené operačnému systému, na ktorom pracujú, vďaka čomu sú jednoduchšie pre pochopenie.

Ďalším prvkom je rozšíriteľnosť pomocou zásuvných modulov. Jedná sa o doplnky poskytované predajcami tretích strán, ktoré je možné pridať do API hrubých klientov pre rozšírenie o potrebnú funkciu.

Manipulácia s dátami je vykonávaná lokálne. Ak je dostupné sieťové pripojenie, je typicky použité len na stiahnutie potrebných dát, ktoré sú následne spracovávané lokálne a v prípade potreby poslané späť na server. Tento spôsob umožňuje aplikáciám pracovať *offline* bez nutnosti sieťového spojenia.

Napriek neuveriteľnej funkčnosti majú tieto aplikácie určité obmedzenia. Väčšina z nich je samostatná a pracuje na klientskom počítači takmer bez znalosti svojho prostredia [18].

[1] podpora viacerých platforiem

[2] určitá funkčnosť ako modifikácia pamäte používanej inými aplikáciami môže byť obmedzená, kvôli ochrane systému pred hrozbami

Prostredím sa myslia ostatné počítače alebo služby v sieti, rovnako ako ostatné aplikácie na používateľskom počítači. Hrubí klienti síce poskytujú vysokokvalitný, responzívny používateľský zážitok (*user experience*) a dobrú podporu rôznych platforiem, no sú však veľmi zložité pre nasadenie a údržbu. Navyše vyžadujú výkonný procesor alebo veľa pamäti, ale zväčša nevyužívajú výkonný *hardware* [13]. Každá aplikácia typu hrubý klient musí byť inštalovaná, udržiavaná a aktualizovaná na každej stanici (používateľskom počítači) zvlášť. Ako rastie zložitosť aplikácií a klientských platforiem, takisto aj problémy spojené s nasadením aplikácie na klientský počítač spoľahlivo a bezpečne [18]. Jedna aplikácia môže jednoducho narušiť inú, ak dôjde k nasadeniu nekompatibilnej zdieľanej knižnice či komponentu.

2.3 Bohatý webový klient

Bohatý webový klient (*rich web client*) je tiež známy ako bohatá internetová aplikácia (*rich Internet application*), AJAX klient, Web 2.0 klient a tučný tenký klient [13]. Od čias éry Web 2.0 sa staré technológie webových klientov začali rýchlo vyvíjať a pojem webový klient sa tak zásadne zmenil. Hlavný podiel mal na tom jazyk JavaScript, vďaka ktorému môžu webové aplikácie meniť statický obsah a pracovať s prvkami stránky. Tiež použitím technológie AJAX je možná komunikácia so serverom bez nutnosti obnovenia stránky, čo znamená, že používatelia môžu pokračovať v práci, zatiaľ čo aplikácia získava dáta zo serveru na pozadí.

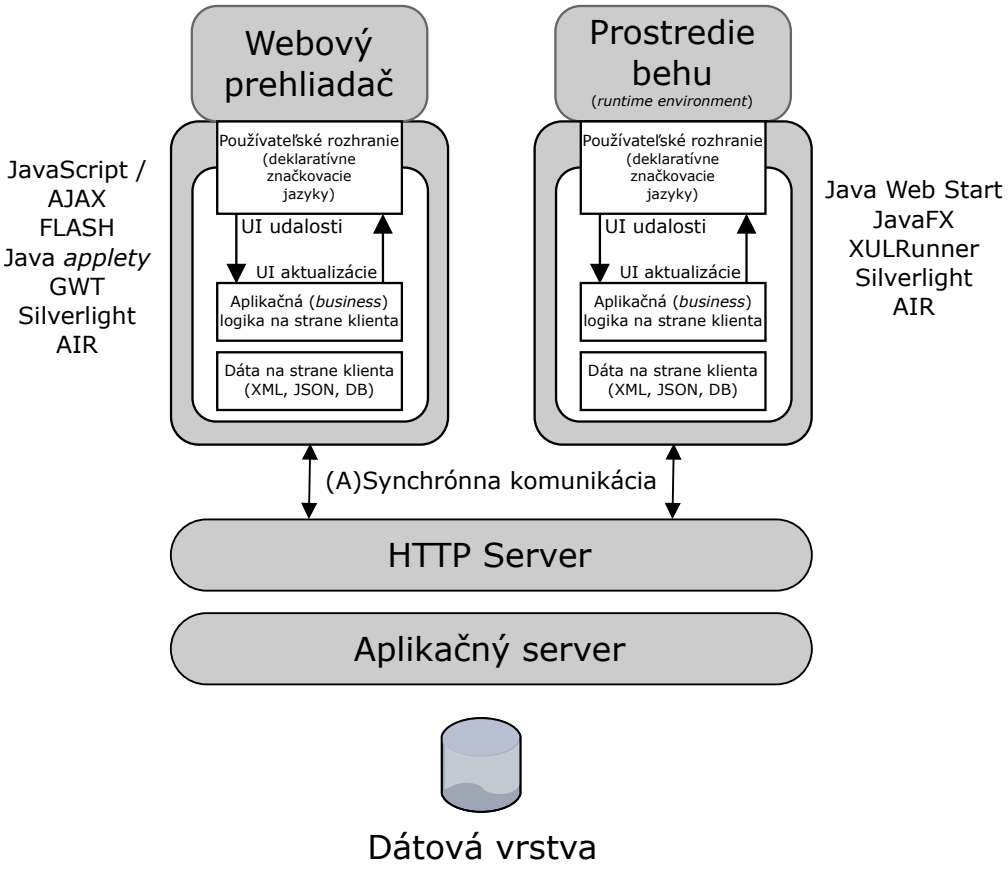
Bohaté internetové aplikácie poskytujú koncovým používateľom vysokú mieru interakcie použitím rôznych mechanizmov rozhrania (napr. natívna podpora multimédií, efekty, animácie a pod.) [9]. Oproti tenkým klientom je práca medzi klientom a serverom flexibilnejšia, keďže jej časť, pôvodne vykonávanú serverom, teraz zabezpečuje klient – konkrétne sa jedná o časť aplikačných dát a výpočtovej logiky. Tento aspekt je pravdepodobne hlavným dôvodom úspechu bohatých internetových aplikácií: použiť web ako *back-end* si uchováva všetky výhody otvorenej, nízkorozpočtovej architekúry bez nutnosti inštalácie, zatiaľ čo zvyšujúca sa výpočetná sila klientov zabezpečuje kvalitu interakcie, ktorú moderné desktopové aplikácie a operačné systémy poskytujú používateľom.

Základná architektúra bohatých internetových aplikácií je zobrazená na obrázku 2.1: systém sa skladá z webového aplikačného servera a skupiny používateľských aplikácií bežiacich na klientských strojoch. Tieto aplikácie sú implementované buď v rámci prehliadača za použitia rôznych technológií ako JavaScript, Flash animácie, Java *applety*, alebo mimo prehliadač ako binárne súbory, ktoré sa dajú stiahnuť z webu a interpretovať v špecifickom behovom prostredí (*runtime environment*).

Mať aplikačné behové prostredie aj na strane klienta prináša niekoľko nových príležitostí:

- riadiaca (*control*) logika môže byť implementovaná na strane klienta alebo servera
- aplikačná (*business*) logika môže byť rozdelená medzi klienta a server
- dáta môžu byť uložené na strane klienta, aj na strane servera
- komunikácia môže byť flexibilnejšia – klient-server a server-klient
- klient môže pracovať aj bez pripojenia k serveru

Bohatá internetová aplikácia predstavuje spektrum nových architektonických vzorov, možností návrhu a implementačných jazykov. V zásade akákoľvek udalosť vyvolaná používateľom môže byť riešená lokálne u klienta, delegovaná na server, alebo riešená na oboch vrstvách.



Obr. 2.1: Architektúra bohatej internetovej aplikácie [9]

Niekto považuje dobre navrhnuté aplikácie alebo aplikácie s vizuálnymi efektami za bohaté aplikácie, keďže nie je jednoduché z pohľadu používateľa rozhodnúť, či sa jedná o bohatú webovú aplikáciu. Ako uvádza F. Lange vo svojej knihe ([13]), základným pravidlom je: „Ak webová aplikácia používa JavaScript pre načítanie dát asynchrónne, je to bohatá webová aplikácia.“

Existuje množstvo iných typov klientov, ďalšie kapitoly však budú venované hlavne vývoju bohatých internetových aplikácií.

Kapitola 3

Technológie pre tvorbu web API

V 60. rokoch minulého storočia začali vývojári s tvorbou štandardných knižníc pre prvé procedurálne jazyky [11]. Knižnice zdieľali s ostatnými vývojármi, ktorí mohli využiť ich štandardnú funkcionalitu bez nutnosti poznania vnútorného kódu.

Následne v 70. a 80. rokoch, spolu so vznikom sieťou spojených počítačov, prišli prvé sieťové API poskytujúce služby, ktoré vývojári mohli využívať pomocou tzv. vzdialeného volania procedúr (*Remote Procedure Call*, ďalej len RPC). RPC bolo zahájené klientom, ktorý odoslal žiadosť na vzdialený počítač, aby vykonal určitú činnosť a ten spätne zaslal klientovi odpoveď [20]. Počítače tohto typu sa síce líšili od dnešných klientov a serverov, ale princíp toku informácií zostal rovnaký: klient požaduje dáta, server posiela odpoveď.

Počas 90. rokov chcela väčšina spoločností štandardizovať spôsob, akým sú API tvorené a vystavované [11]. Štandardy ako CORBA (*Common Object Request Broker Architecture*), COM (*Component Object Model*) či DCOM (*Distributed Component Object Model*) sa usilovali o vystavenie služieb cez web. Ich hlavným problémom bola zložitá správa a konfigurácia.

Koncom 90. rokov prišli otvorenejšie a štandardnejšie spôsoby prístupu k vzdialeným službám cez web (webové API). V prvom rade SOAP (*Simple Object Access Protocol*) a XML (*Extensible Markup Language*), následne REST (*Representational State Transfer*) a JSON (*JavaScript Object Notation*), vďaka ktorým sa prístup k službám stal jednoduchším a štandardizovanejším, bez závislostí na klientskom kóde alebo programovacom jazyku.

V tejto kapitole sa čitateľ oboznámi so základnou definíciou API a jeho hlavnými vzormi (kapitola 3.1). Podstatná časť je venovaná moderným vzorom – REST a GraphQL, ktoré sa v dnešnej dobe využívajú vo väčšine webových aplikácií.

Kapitola 3.2 uvádza kľúčové princípy, ktoré by mala REST aplikácia spĺňať, ďalej reprezentáciu dát (zdroje) a opis základných HTTP metód používaných v tomto API vzore.

Kapitola 3.3 je venovaná definícií základných typov a ich väzieb, ktoré môžu vznikať v GraphQL schéme. Okrem toho sú tu spomenuté dopyty a mutácie – významné prvky technológie GraphQL. Všetko je doplnené konkrétnymi príkladmi pre lepšie pochopenie problematiky. Záver kapitoly sa zaoberá podporou GraphQL na rôznych klientských a serverových platformách.

Posledná kapitola (3.4) zhŕňa poznatky kapitol, ktoré jej predchádzajú a uvádza hlavné rozdiely medzi dnes najpopulárnejšími prístupmi pre tvorbu API, výhody a nevýhody použitia technológie GraphQL.

3.1 API

Aplikačné programové rozhranie (*Application programming interface*, ďalej len API) je rozhranie, ktoré program poskytuje iným programom, ľuďom a v prípade webových API celému

svetu (vďaka internetu) [11]. API sú zakladnými stavebnými blokmi, ktoré umožňujú vzájomnú spoluprácu pre hlavné podnikové (*business*) platformy na webe. Majú za úlohu napr. zdieľanie dát pre predpoveď počasia z dôveryhodného zdroja ako *National Weather Service* stovkám aplikácií, ktoré sa špecializujú na ich prezentáciu. API spracovávajú naše kreditné karty a umožňujú spoločnostiam nepretržite zbierať naše peniaze bez toho, aby sme sa museli zaoberať detailami finančných technológií a im zodpovedajúcim právam a správnym nariadeniam. API sú kľúčovými komponentmi škálovateľných a úspešných internetových spoločností ako Amazon, Stripe, Google či Facebook.

API sú potrebné pri výmene informácií s poskytovateľmi dát, ktorí dokážu riešiť špecifické problémy, takže ľudia a spoločnosti nemusia tieto problémy riešiť sami. Napríklad, ak chcú na svojej stránke využívať interaktívne mapy, nemusia „objavovať už vynájdené“, ale použiť Google Maps API. Podobne je to aj s Facebook Login, *chatbot* či ďalšími.

Existuje viacero API vzorov (*paradigms*) – rozhraní, ktoré odhaľujú *back-endové* dáta služby iným aplikáciám. Medzi dnes najpopulárnejšie patria: REST, GraphQL, RPC, WebHooks či WebSockets. Dve z nich – REST a GraphQL sú bližšie opísané v nasledujúcich kapitolách.

3.2 REST

Representational state transfer alebo REST je architektonický štýl pre návrh distribuovaných internetových aplikácií [24]. Aplikácie, ktoré sa držia nasledujúcich obmedzení či princípov[1] sú považované za tzv. RESTful aplikácie:

- **klient-server** – činnosti by mali byť rozdelené medzi klientov a servery, vďaka čomu sa môžu vyvíjať nezávisle
- **bezstavovosť** – komunikácia medzi klientom a serverom by mala byť bezstavová
- **vrstvený systém** – viacero hierarchických vrstiev ako brány (*gateways*), *firewally* či *proxy* môže existovať medzi klientom a serverom
- ***cache*** – odpovede od servera musia byť označené ako tzv. „kešovateľné“ (*cacheable*) alebo „nekešovateľné“ (*noncacheable*)
- **jednotné rozhranie** – všetky interakcie medzi klientom, serverom a sprostredkujúcimi komponentmi sú založené na jednotnosti ich rozhraní
- **kód na požiadanie** – klienti môžu rozšíriť svoju funkčnosť stiahnutím a spustením kódu na požiadanie (skripty, Java *applety*, Silverlight a pod.)

Vyššie uvedené obmedzenia nediktujú, aké špecifické technológie majú byť použité pre vývoj aplikácií. Namiesto toho, je možné pri ich dodržaní vytvoriť škálovateľné, prenosné, spoľahlivé a výkonnejšie aplikácie.

Zdroje

Zdroje sú zakladným konceptom REST-u. Zdrojom je čokoľvek, k čomu možno pristupovať alebo s čím možno manipulovať – videá, obrázky, používateľské profily, ale aj hmotné veci ako ľudia či zariadenia. Zdroje sa typicky vzťahujú k iným zdrojom, ako napr. objednávka

[1] podľa dizertačnej práce Roya Fieldinga – dostupné z: https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

a produkty v nej. Tiež je možné ich zoskupovať do kolekcií. Bývajú reprezentované rôznymi formátmi ako XML, JSON, HTML vo forme textu, ale aj binárne formáty ako PDF, JPEG či MP4. Web poskytuje jednotný identifikátor zdroja (ďalej len URI[2]), ktorý sa používa pre jednoznačné označenie týchto zdrojov. Tabuľka 3.1 uvádza príklady URI a zdroje, ktoré reprezentujú.

URI	Opis zdroja
`http://blog.example.com/posts`	Reprezentuje kolekciu príspevkov blogu
`http://blog.example.com/posts/1`	Reprezentuje príspevok blogu s ID `1`
`http://blog.example.com/posts/1/comments`	Reprezentuje kolekciu komentárov súvisiacich s príspevkom blogu, ktorý má ID `1`
`http://blog.example.com/posts/1/comments/245`	Reprezentuje komentár s ID `245`

Tabuľka 3.1: URI a opis zdroja [24]

HTTP metódy

Princíp jednotného rozhrania určuje interakcie medzi klientom a serverom pomocou niekoľkých štandardizovaných operácií či metód. HTTP štandard[3] poskytuje osem HTTP metód, ktoré umožňujú klientom pristupovať a manipulovať so zdrojmi. Najdôležitejšie sú:

- **GET** – používa sa pre získanie reprezentácie zdroja. Napr. GET na `http://blog.example.com/posts/1` vráti reprezentáciu príspevku blogu s ID `1`. Oproti tomu GET na `http://blog.example.com/posts` získa kolekciu všetkých príspevkov blogu. Jednoduchosť tejto metódy je často zneužitá na vykonávanie operácií ako vymazávanie alebo aktualizovanie reprezentácií zdrojov. Takéto použitie však narúša sémantiku HTTP štandardu.
- **HEAD** – v prípadoch, kedy klient potrebuje zistiť, či daný zdroj existuje, ale nezaujíma ho jeho reprezentácia, by bolo použitie metódy GET príliš „ťažkopádne“. Vhodnejšou variantou je metóda HEAD, ktorá získa len hlavičku súvisiacu so zdrojom, ale nie jeho reprezentáciu.
- **DELETE** – ako názov napovedá, táto metóda sa používa pre vymazanie zdroja. Pri obdržaní požiadavky server vymaže daný zdroj. Ak by proces vymazávania mal trvať dlhšiu dobu, server to typicky oznámi prostredníctvom zaslania potvrdzujúcej správy, že požiadavku obdržal a pracuje na tom.
- **PUT** – metóda PUT umožňuje klientovi modifikovať stav zdroja. Klient modifikuje stav zdroja a odošle aktualizovanú reprezentáciu na server použitím PUT metódy. Pri obdržaní požiadavky server nahradí stav zdroja novým stavom.

  Táto metóda sa používa aj pre vytvorenie nového zdroja, je však nutné, aby klient poznal URI nového zdroja.

[2] Uniform Resource Identifier
[3] https://www.ietf.org/rfc/rfc2616.txt

- **POST** – používa sa hlavne pre vytváranie zdrojov, typicky v kolekciách. Napr. POST na `http://blog.example.com/posts/` vytvorí nový príspevok blogu v kolekcií príspevkov (*posts*).

  Oproti metóde PUT nie je nutná znalosť URI. Server je zodpovedný za priradenie nového ID zdroju a príslušného URI. Vo vyššie uvedenom príklade by aplikácia spracovala POST požiadavku a vytvorila nový zdroj na `http://blog.example.com/posts/123`, kde `123` je serverom vygenerované ID.

- **PATCH** – metóda PUT vyžaduje zaslanie celej reprezentácie zdroja, čo v určitých prípadoch nemusí byť vyhovujúce. Preto je lepšie použiť metódu PATCH, ktorá umožňuje čiastočné aktualizácie zdroja.

CRUD a HTTP metódy

Aplikácie pracujúce s dátami typicky používajú termín CRUD, ktorého iniciály v angličtine označujú štyri základné perzistentné funkcie: vytvorenie (*Create*), čítanie (*Read*), aktualizácia (*Update*) a odstránenie (*Delete*). S týmito operáciami sú často mylne spájané štyri HTTP metódy: GET, POST, PUT a DELETE. Typickú asociáciu ukazuje tabuľka 3.2.

Operácia	**HTTP metóda**
Create	POST
Read	GET
Update	PUT
Delete	DELETE

Tabuľka 3.2: Častá (a nie celkom správna) asociácia medzi CRUD operáciami a HTTP metódami [24]

Vyššie uvedené vzťahy sú správne pre operácie *Read* a *Delete*, avšak nie úplne správne pre *Create*/*Update* a POST/PUT. Ako už bolo spomenuté, metóda PUT môže byť použitá pre vytvorenie, tiež metóda PATCH pre aktualizáciu zdroja. Roy Fielding vo svojom blogu[4] uviedol, že aj metóda POST môže byť použitá pre aktualizácie. Z hľadiska návrhu API je teda skôr dôležité použiť správne metódy pre danú operáciu, než ich jednoducho mapovať v pomere 1:1 s CRUD.

3.3 GraphQL

V roku 2012 sa Facebook rozhodol prestavať svoje mobilné aplikácie, kvôli nedostatočnému výkonu a častým haváriám [20]. S vtedajším RESTful serverom a FQL (Facebook verzia SQL) dátovými tabuľkami potreboval Facebook vylepšiť spôsob, akým boli dáta posielané klientom. Výsledkom bol GraphQL, ktorý Facebook spočiatku používal interne. Následne v roku 2015 bol verejne sprístupnený a prijatý mnohými poskytovateľmi API ako GitHub, Yelp a Pinterest [11], ale aj spoločnosťami IBM, Intuit či Airbnb [20].

GraphQL je dopytovací jazyk na úrovni aplikačnej vrstvy, ktorý bol navrhnutý pre interpretáciu reťazca zo servera/klienta a vrátenie príslušných dát v zrozumiteľnej, stabilnej

[4] http://roy.gbiv.com/untangled/2009/it-is-okay-to-use-post

a predikovateľnej forme [5]. To zabezpečuje pomocou jednoduchých, ľahko pochopiteľných dopytov a príkazov. Ešte predtým, ako možno takéto dopyty vytvárať, je nutné špecifikovať typy.

Typy

Hlavnou jednotkou GraphQL je typ (*type*), ktorý reprezentuje vlastný objekt a tieto objekty opisujú hlavné prvky aplikácie [20]. Napr. sociálne siete pozostávajú z Používateľov a Príspevkov. Typy reprezentujú aplikačné dáta.

Typ obsahuje polia (*fields*) predstavujúce dáta, ktoré súvisia s každým objektom. Každé pole vracia špecifický typ dát, tj. celé číslo (*integer*), reťazec (*string*), ale aj vlastný typ alebo zoznam typov.

Kolekcia typov tvorí schému, ktorú možno definovať pomocou jazyka SDL (*Schema Definition Language*). Schémy sú typicky ukladané v textových dokumentoch a sú používané klientmi aj servermi pre validáciu GraphQL dopytov. Ukážka typu v SDL je uvedená v kóde 3.1.

```
type Photo {
  id: ID!
  name: String!
  url: String!
  description: String
}
```

Kód 3.1: Ukážka typu v jazyku SDL

Typ `Photo` predstavuje fotografiu – v zložených zátvorkách sú uvedené polia oddelené od ich typu dvojbodkou. Výkričník špecifikuje, že dané pole je nenulovateľné (*non-nullable*) – polia označené týmto spôsobom musia v každom dopyte vrátiť nejaké dáta. Pole `id` je typu `ID` (jedná sa o tzv. skalárny typ), ktorý sa používa, ak má byť vrátený unikátny identifikátor.

Skalárne typy

GraphQL obsahuje viacero skalárnych typov: `Int`, `Float`, `String`, `Boolean`, `ID`. Skalárny typ nie je objektový typ – neobsahuje polia. Tiež je možné špecifikovať vlastný skalárny typ uvedením kľúčového slova `scalar` (kód 3.2).

```
scalar DateTime

type Photo {
  id: ID!
  name: String!
  url: String!
  description: String
  created: DateTime!
}
```

Kód 3.2: Ukážka skalárneho a objektového typu v jazyku SDL

Vymenované typy

Vymenovaný typ alebo *enum* je skalárny typ, ktorého polia vracajú obmedzenú množinu reťazcov (kód 3.3).

```
enum PhotoCategory {
  SELFIE
  PORTRAIT
  LANDSCAPE
}
```

Kód 3.3: Ukážka vymenovaného typu v jazyku SDL

Ostatné typy

Okrem vyššie uvedených typov existuje množstvo ďalších, ktoré GraphQL definuje. Patria medzi ne hlavne: rozhrania, vstupné typy (*input*) či *uniony*. Niektoré z nich ešte budú v tejto práci spomenuté, ich príklady však tu už uvedené nie sú. Ďalšie informácie ohľadom typového systému GraphQL možno nájsť na oficiálnej stránke špecifikácie GraphQL[5].

Zoznamy a väzby

Pri tvorbe GraphQL schém je možné definovať polia, ktoré vracajú zoznamy akéhokoľvek GraphQL typu. Možno ich vytvoriť obalením daného typu do hranatých zátvoriek. Napr. `[String]` definuje zoznam reťazcov, `[Photo]` zasa zoznam fotografií (vlastných typov).

Schopnosť spájať dáta a dopytovať viacero typov súvisiacich dát je veľmi dôležitým prvkom. Práve tvorbou zoznamov vlastných objektových typov dochádza k spojeniu objektov, čím medzi nimi vznikajú rôzne druhy väzieb:

- **1:1** – definovaním poľa vracajúceho vlastný objektový typ, vzniká väzba 1:1 medzi dvoma objektami. Príkladom môže byť pole `ownedBy` určujúce väzbu medzi používateľom a fotografiou, ktorú vlastní (kód 3.4). Toto pole je definované v type `Photo` a špecifikuje, že fotografia môže mať len jedného vlastníka.
- **1:M** – táto väzba vzniká, ak rodičovský objekt obsahuje pole vracajúce zoznam vlastných objektových typov. Napr. v prípade, že používateľ vlastní viacero fotografií, stačí pridať pole `ownedPhotos` do typu `User`, ktoré vracia zoznam fotografií (kód 3.5).
- **M:N** – niekedy situácia vyžaduje prepojiť zoznamy objektov s inými zoznamami objektov. Pre vytvorenie väzby M:N je nutné do oboch typov pridať polia vracajúce zoznam druhého typu. Kód 3.6 zobrazuje prípad, kedy sa používateľ môže nachádzať vo viacerých fotografiách a zároveň na fotografii môže byť označených viacero používateľov.

```
type User {
  id: ID!
  name: String!
}
```

```
type Photo {
  ...
  ownedBy: User!
}
```

Kód 3.4: Ukážka väzby 1:1 medzi objektovými typmi `User` (vľavo) a `Photo` (vpravo) v jazyku SDL. Kvôli prehľadnosti sú niektoré polia vynechané

[5] https://graphql.github.io/graphql-spec/June2018/#sec-Type-System

```
type User {
  ...
  ownedPhotos: [Photo!]!
}
```

```
type Photo {
  ...
  ownedBy: User!
}
```

Kód 3.5: Ukážka väzby 1:M medzi objektovými typmi `User` (vľavo) a `Photo` (vpravo) v jazyku SDL. Kvôli prehľadnosti sú niektoré polia vynechané

```
type User {
  ...
  inPhotos: [Photo!]!
}
```

```
type Photo {
  ...
  taggedUsers: [User!]!
}
```

Kód 3.6: Ukážka väzby M:N medzi objektovými typmi `User` (vľavo) a `Photo` (vpravo) v jazyku SDL. Kvôli prehľadnosti sú niektoré polia vynechané

Dopyty

Pre získanie dát z API je možné použiť operáciu dopyt (*query*). Zaslaním dopytu na GraphQL server sú vyžiadané jednotky dát pomocou polí, ktoré sa mapujú na im zodpovedajúce polia v JSON odpovedi obdržanej od servera.

`Query` je jedným z viacerých koreňových typov – jedná sa o špeciálny typ, ktorého polia sa mapujú na operácie. Tieto polia sú prístupné ako koreňové polia pre dopyt a možno ich v konkrétnom dopyte volať.

Pre demonštráciu GraphQL dopytu a následnej odpovede možno využiť ukážkové *Star Wars* API[6] – API typu REST „obalené“ (*wrapped*) GraphQL. Kód 3.7 zobrazuje niekoľko polí v type `Query` tohto API. V kóde 3.8 je znázornený dopyt (vľavo) a odpoveď naň (vpravo). Sú požadované dáta osoby (s ID 4), konkrétne jej meno (`name`), pohlavie (`gender`) a farba očí (`eyeColor`). Odpoveď vo formáte JSON má rovnaký tvar ako vytvorený dopyt a obsahuje len dáta, ktoré boli požadované.

```
type Query {
  film(id: ID, filmID: ID): Film
  person(id: ID, personID: ID): Person
}
```

Kód 3.7: Ukážka koreňového typu `Query` v SWAPI[6]

```
query {
  person(personID: 4) {
    name
    gender
    eyeColor
  }
}
```

```
{
  "data": {
    "person": {
      "name": "Darth Vader",
      "gender": "male",
      "eyeColor": "yellow"
    }
  }
}
```

Kód 3.8: Ukážka GraphQL dopytu pre získanie osoby (vľavo) a odpovede naň (vpravo)[6]

[6] https://graphql.org/swapi-graphql

V rámci dopytu sú typicky špecifikované polia obalené do zložených zátvoriek. Takéto bloky sa nazývajú výberové množiny (*selection sets*) a ich polia priamo súvisia s GraphQL typmi. Dopyt môžno prispôsobovať podľa potreby, pridávať či odoberať príslušné polia. Pridaním poľa `filmConnection`, je získaný názov každého filmu, v ktorom sa žiadaná osoba ukázala (kód 3.9). Dopyt je vnorený a ak sa vykoná, môže prechádzať súvisiace objekty, tj. pre dva typy dát stačí jedna HTTP požiadavka.

```
query {
  person(personID: 4) {
    name
    gender
    eyeColor
    filmConnection {
      films {
        title
      }
    }
  }
}
```

```
{
 "data": {
   "person": {
     "name": "Darth Vader",
     "gender": "male",
     "eyeColor": "yellow",
     "filmConnection": {
       "films": [
         {
           "title": "A New Hope"
         },
         {
           "title": "The Empire
               Strikes Back"
         },
         {
           "title": "Return of the
               Jedi"
         },
         {
           "title": "Revenge of the
               Sith"
         }
       ]
     }
   }
 }
}
```

Kód 3.9: Ukážka GraphQL dopytu pre získanie osoby (vľavo) a odpovede naň (vpravo)[7]. Oproti kódu 3.8 obsahuje dopyt polia pre získanie filmov, v ktorých sa žiadaná osoba ukázala

Mutácie

Čítanie dát v GraphQL zabezpečujú dopyty, pre modifikáciu dát sa používajú mutácie (*mutations*). Sú to teda operácie, ktoré vykonávajú zápis nových dát či aktualizáciu už existujúcich dát. Definíciou sú podobné dopytom – majú svoj názov a môžu obsahovať výberové množiny vracajúce objektové alebo skalárne typy.

Rovnako ako `Query`, aj `Mutation` je koreňový objektový typ. Schéma API špecifikuje polia prístupné v tomto type. Aj keď SWAPI možno využiť len pre posielanie dopytov, ako by mohli vyzerať niektoré jeho mutácie ukazuje kód 3.10.

```
type Mutation {
  deleteFilm(id: ID, filmID: ID): Boolean
  updatePersonName(id: ID, personID: ID, name: String): Person
}
```

Kód 3.10: Ukážka koreňového typu `Mutation`

Ako názov napovedá, prvá mutácia je určená k vymazaniu filmu podľa ID. Je znázornená v kóde 3.11 vľavo a neobsahuje výberovú množinu, keďže jej návratovým typom je `Boolean`. Podobne ako pri dopytoch, by odpoveď bola vo formáte JSON.

Druhá mutácia slúži k zmene mena osoby s konkrétnym ID. Pretože jej návratovým typom je objekt, oproti prvej mutácii obsahuje aj výberovú množinu (kód 3.12), kde je uvedené pole `name` (meno). V odpovedi by sa už nachádzalo aktualizované meno danej osoby.

```
mutation {
  deleteFilm(filmID: 3)
}
```

```
{
  "data": {
    "deleteFilm": true
  }
}
```

Kód 3.11: Ukážka GraphQL mutácie pre vymazanie filmu (vľavo) a odpovede na ňu (vpravo)

```
mutation {
  updatePersonName(personID: 4,
     name: "John Vader"): {
    name
  }
}
```

```
{
  "data": {
    "updatePersonName": {
      name: "John Vader"
    }
  }
}
```

Kód 3.12: Ukážka GraphQL mutácie pre úpravu mena (vľavo) a odpovede na ňu (vpravo)

Subscriptiony

Tretím typom operácie v GraphQL je *subscription*, ktorý sa používa v prípade, že klient chce zisťovať zmeny dát v reálnom čase. V tejto práci však *subscription* nie je použitý, preto je tu uvedený len pre úplnosť.

GraphQL vs. SQL

SQL (*Structured Query Language*) je dopytovací jazyk používaný pre prístup, správu a manipuláciu dát v databáze. SQL zaviedol myšlienku prístupu k viacerým záznamom pomocou jedného príkazu, a to nielen použitím ID, ale aj iných kľúčov. `SELECT`, `INSERT`, `UPDATE` a `DELETE` sú príkazmi, ktoré používa SQL. Napísaním jedného dopytu možno získať spojené dáta naprieč rôznymi tabuľkami v databáze.

Podobne ako SQL, možno použiť GraphQL mutácie pre zmenu alebo vymazanie dát. Aj keď „QL“ v SQL a GraphQL má rovnaký význam (*Query Language*), sú tieto tech-

nológie úplne odlišné. SQL dopyty sú posielané databáze, GraphQL dopyty zasa danému API. SQL dáta sú uložené v dátových tabuľkách, dáta GraphQL môžu byť uložené prakticky kdekoľvek: v databázach (aj rôznych), súborových systémoch, REST API, dokonca aj iných GraphQL API. SQL je dopytovací jazyk pre databázy, GraphQL je dopytovací jazyk pre internet. Syntax je taktiež odlišná: namiesto `SELECT` používa GraphQL `Query` pre vyžiadanie dát. `INSERT`, `UPDATE` a `DELETE` sú v GraphQL alternatívou jediného dátového typu: `Mutation`.

Klienti a servery

GraphQL je len špecifikáciou pre komunikáciu medzi klientom a serverom. Špecifikácia opisuje možnosti a charakteristiky jazyka, tj. nediktuje, aký konkrétny jazyk má byť použitý, ako majú byť dáta skladované alebo akých klientov podporovať.

Architektúra zostáva teda na vývojároch, čo viedlo k vytvoreniu nástrojov, ktoré im zrýchľujú prácu a zlepšujú efektivitu a výkon aplikácií. Tieto nástroje sa označujú ako GraphQL klienti a majú za úlohu aj riešenie sieťových dopytov či *cache* dát. Najznámejšími GraphQL klientmi sú Relay a Apollo.

Relay[8] je Facebook klient, ktorý pracuje s React a React Native. Relay sa zameriava na spojenie medzi React komponentmi a dátami, ktoré sú načítané z GraphQL serveru. Je používaný spoločnosťami ako Facebook, GitHub, Twitter a pod.

Apollo[9] klient bol vyvinutý firmou Meteor Development Group a je komplexným nástrojom pre GraphQL. Podporuje všetky hlavné *frontend* platformy a je nezávislý na *frameworkoch*. Apollo tiež vyvíja nástroje, ktoré pomáhajú s tvorbou GraphQL služieb (zvýšenie výkonu *backend* služieb) a nástroje pre monitorovanie výkonnosti GraphQL API. Spoločnosti Airbnb, CNBC, The New York Times či Ticketmaster využívajú Apollo klienta v produkcii.

Vyššie uvedení klienti sú určení pre jazyk JavaScript, no GraphQL podporuje mnoho ďalších jazykov, ktoré možno použiť pre vývoj na strane klienta. Okrem toho existuje veľké množstvo serverových knižníc pre často používané jazyky pre tvorbu serverovej časti webových aplikácií. Zoznam najpopulárnejších je vypísaný na oficiálnej stránke GraphQL[10].

3.4 GraphQL vs. REST

Na základe predchádzajúcich kapitol možno vyvodiť hlavné výhody GraphQL oproti populárnym alternatívam, ako napr. REST či RPC. GraphQL umožňuje klientom vnorovať dopyty a získavať tak dáta naprieč rôznymi zdrojmi pomocou jednej požiadavky [11]. Redukcia množstva HTTP volaní, ktoré je nutné odoslať na server, prináša zvýšenie rýchlosti v prípade mobilných aplikácií využívajúcich GraphQL, aj za podmienok pomalšieho sieťového pripojenia.

S dopytmi súvisí aj množstvo dát, ktoré je posielané medzi klientom a serverom. Kvôli charakteru REST-u a systémom, ktoré túto architektúru používajú, vyúsťujú REST API často v problém označovaný v angličtine ako *overfetching*, resp. *underfetching* [5]. *Overfetching* nastáva, ak je získaných viac dát, ako bolo potreba, naproti tomu *underfetching* je opakom, tj. bol získaný nedostatok dát pri dopytovaní. Pri zvyšovaní zložitosti API rastie aj zložitosť zdrojov, čo môže byť problémom. V GraphQL je možné pridávať nové polia a typy bez vplyvu na existujúce dopyty [11]. Oproti REST či RPC API vedia poskytova-

[8] https://facebook.github.io/relay/
[9] https://www.apollographql.com/
[10] https://graphql.org/code/

telia GraphQL API jednoduchšie zistiť, aké polia klienti používajú, na základe čoho môžu odstrániť polia, ktoré nie sú často využívané.

GraphQL má oproti REST-u silný typový systém, ktorý pomáha počas vývoja zabezpečiť, že dopyt je platný a syntakticky správny. Typy sú overované voči definovanej schéme, ktorú možno preskúmať aj z pohľadu klienta [5]. Schopnosť preskúmať API je veľmi dôležitá, najmä pre začínajúcich vývojárov či používateľov so systémom funkcií jednoduchých pre pochopenie a preskúmanie. Aj keď existujú externé riešenia ako Swagger, ktoré pomáhajú s prieskumom REST API, GraphQL možno preskúmať prirodzeným spôsobom [11]. Prichádza s nástrojom GraphiQL, ktorý umožňuje písať, overovať a testovať GraphQL dopyty či mutácie v rámci webového prehliadača (viac v kapitole 7).

Aj keď GraphQL má veľa pozitív, jednou z jeho nevýhod je náročnosť, ktorú pridáva pre poskytovateľa API [5, 20]. Server potrebuje dodatočne spracovávať zložité dopyty a overovať parametre, optimalizácia výkonu GraphQL dopytov môže byť tiež náročná. V rámci jednej firmy je jednoduché predpovedať určité prípady a odhaliť tak nedostatky, avšak pri práci s externými zamestnancami sa prípady stanú náročnými pre pochopenie a optimalizáciu. Pre jednoduché API je REST vyhovujúcou alternatívou, avšak ako dáta naberajú na zložitosti, vyžadujú väčšinou klienti rôzne tvary dopytov, kde vynikajú práve výhody GraphQL.

Kapitola 4

Analýza a návrh aplikácie

Jedným z hlavných cieľov tejto práce je vytvorenie webovej aplikácie využívajúcej technológie GraphQL pre aplikačné rozhranie. Po dohode s vedúcim práce bola zvolená aplikácia pre osobný rozvoj zamestnancov vo firme zaoberajúcej sa vývojom *softwaru*.

Táto kapitola sa zaoberá špecifikáciou požiadaviek, ich analýzou a nakoniec návrhom celkovej aplikácie. Najskôr boli spísané požiadavky na výslednú aplikáciu po niekoľkých konzultáciách so zodpovednou osobou vo firme. Následne prebehla analýza požiadaviek, v ktorej boli definovaní aktéri. Plán vývoja bol rozdelený na niekoľko iterácií a pre každú bol navrhnutý diagram prípadov použitia, spolu so špecifikáciami vybraných prípadov použitia. Nasledoval návrh GraphQL schémy, kde sú určené základné typy a vzťahy medzi nimi. Navrhnutá architektúra je rozdelená do niekoľkých častí, ktoré sú tiež opísané. Databáza je navrhnutá prostredníctvom ER diagramu a poslednou kapitolou je návrh používateľského rozhrania, rozdelený na viacero etáp, ku ktorým sú uvedené príslušné obrázky a prístupy súvisiace s návrhom.

4.1 Špecifikácia požiadaviek

Softwarová spoločnosť potrebuje vytvoriť aplikáciu pre osobný rozvoj svojich zamestnancov. Firma vedie niekoľko pobočiek, ktoré sídlia v rôznych mestách a majú svoj špecifický názov.

Zamestnanci pracujúci na konkrétnych pobočkách sú rozdelení do viacerých tímov, ktoré majú svojho vedúceho (*leadera*). Každý zamestnanec má pridelenú určitú pozíciu na úrovni jeho skúseností. Medzi ich osobné údaje patrí titul, meno, dátum narodenia, e-mail a telefónne čísla – aspoň dve. Požaduje sa adekvátne zobrazenie hierarchie spoločnosti, ideálne vo forme organizačnej štruktúry. Okrem správy pobočiek, zamestnancov a pozícií sa od aplikácie očakáva aj vyhľadávanie zamestnancov podľa kľúčových slov.

Osobný rozvoj môže prebiehať vo viacerých smeroch. Aplikácia by mala podporovať rozvoj v zručnostiach, technológiách a jazykoch, a to v odlišných úrovniach:

- **zručnosti** – mäkké zručnosti alebo *soft skills*, ako napr. kreativita, komunikácia, tímová spolupráca atď., aspoň 5 úrovní
- **technológie** – môžu zahŕňať konkrétne programovacie jazyky či *frameworky*, aspoň 5 úrovní podľa skúseností
- **jazyky** – rôzne typy cudzích jazykov, znalosť rozdelená podľa CEFR[1]

[1] https://www.cambridgeenglish.org/exams-and-tests/cefr/

Zamestnanci si môžu špecifikovať ciele, týkajúce sa vyššie uvedených oblastí, ktoré budú plniť. Pri jednotlivých cieľoch je nutné zaznamenávať názov, opis a progresivitu, tj. na koľko percent približne má daný zamestnanec splnený cieľ. Ciele sú evidované v rámci konkrétneho plánu, ktorý má svoj názov a opis.

Aplikácia by mala ďalej umožniť plánovanie stretnutí. Tie prebiehajú systémom „jeden na jedného“ (*one on one*), tj. vedúci a člen tímu. Na určitý dátum a čas naplánuje vedúci stretnutie v konkrétnej miestnosti, kde je následne preberaný pokrok v jednotlivých cieľoch, na ktorých daný zamestnanec pracuje. Na konci stretnutia sú spísané poznámky – očakáva sa interaktívny textový editor s možnosťou formátovania textu. Vedúci tímu by mal mať možnosť, v prípade svojej neprítomnosti alebo vyťaženosti, zmeniť vedúceho plánu. Okrem plánovania stretnutí na konkrétny dátum, by malo byť súčasťou aplikácie aj generovanie stretnutí, a to na dennej, týždennej či mesačnej báze. V rámci plánu by mali byť stretnutia chronologicky zoradené.

Alternatívou k plánom je možnosť vytvárania hodnotení. Tie sú podobné plánom, avšak neprebiehajú formou stretnutí. Každé hodnotenie obsahuje hodnotiteľa, ktorý môže jednorazovo zhodnodtiť pokrok v stanovených cieľoch hodnoteného a následne spísať poznámky k jednotlivým cieľom či celému hodnoteniu. Aplikácia by mala umožniť generovať hodnotenia pre viacerých zamestnancov pre zjednodušenie tvorby hodnotení.

Podľa množstva znalostí oblastí osobného rozvoja a ich úrovne je potrebné určiť skóre zamestnancov, ktoré môže byť dôležité najmä z hľadiska rozdeľovania zamestnancov do tímov, prideľovania koncoročných odmien či samotnej motivácie zamestnancov. Skóre spolu s vyššie uvedenými aspektmi by sa malo dať prehľadne zobraziť.

Výsledná aplikácia by mala podporovať správu používateľov a prideľovanie rolí. Jeden používateľ môže mať pridelených viacero rolí, ktoré určujú jeho práva. Každý z nižšie uvedených typov používateľa si môže zobraziť pobočky, vyhľadať zamestnanca spoločnosti a zobraziť si jeho profil. Tiež by sa mu mali zobrazovať jeho hodnotenia v prípade, že je v pozícii hodnotiteľa. V tom prípade by mal mať právo tieto hodnotenia upravovať.

Člen tímu má právo prehliadať si všetky svoje plány a zobraziť si ich detail, tj. všetky plánované stretnutia a ciele v rámci plánu.

Vedúci tímu má schopnosť vytvárať a spravovať plány členov svojho tímu. Správa plánu v tomto prípade zahŕňa aj správu jednotlivých stretnutí, ktoré sú súčasťou plánu. Ciele sú definované v rámci tvorby plánu. Vedúci taktiež môže upraviť oblasti osobného rozvoja svojho tímového člena.

Organizáciu tímov, tj. pridávanie/vymazávanie zamestnancov do/z tímov, má za úlohu manažér alebo admin. Oproti adminovi môže manažér u zamestnanca upravovať len jeho osobné údaje, prípadne členov tímu. Manažér má ako jediný možnosť generovať a prideľovať hodnotenia, a to akémukoľvek zamestnancovi. Ďalej si môže zobraziť organizačnú štruktúru spoločnosti.

Admin má právo správy jednotlivých oblastí osobného rozvoja (zručností, technológií a jazykov), ako aj pobočiek či pozícií. Pri zamestnancoch môže spravovať všetky údaje, najmä ich práva.

Aplikácia by mala byť intuitívna, používateľsky prívetivá a schopná behu na rôznych druhoch zariadení (počítač, mobil, tablet). Tiež by mala byť lokalizovaná do slovenského a anglického jazyka.

4.2 Analýza požiadaviek

Zo špecifikácie požiadaviek možno definovať nasledujúcich aktérov a ich úlohy v rámci aplikácie:

- **Používateľ** – abstraktný aktér, predstavuje obecne každého z používateľov, slúži najmä pre definíciu spoločných prípadov použitia ostatných aktérov
- **Člen tímu** – reprezentuje aktéra, ktorý má možnosť vykonávať nemodifikujúce akcie (s výnimkou hodnotení), týkajúce sa hlavne svojho osobného rozvoja
- **Vedúci tímu** – aktér, ktorého hlavnou úlohou je kontrola svojich členov tímu v dodržiavaní stanovených termínov a riadenie ich osobného rozvoja
- **Manažér** – predstavuje aktéra, ktorý zabezpečuje príjimanie nových zamestnancov a organizáciu tímov v spoločnosti, ako aj správu hodnotení
- **Admin** – aktér so schopnosťou meniť práva používateľov a tiež správy oblastí osobného rozvoja

Definovaní aktéri a ich prípady použitia sú zobrazení v diagrame prípadov použitia na obr. 4.2.

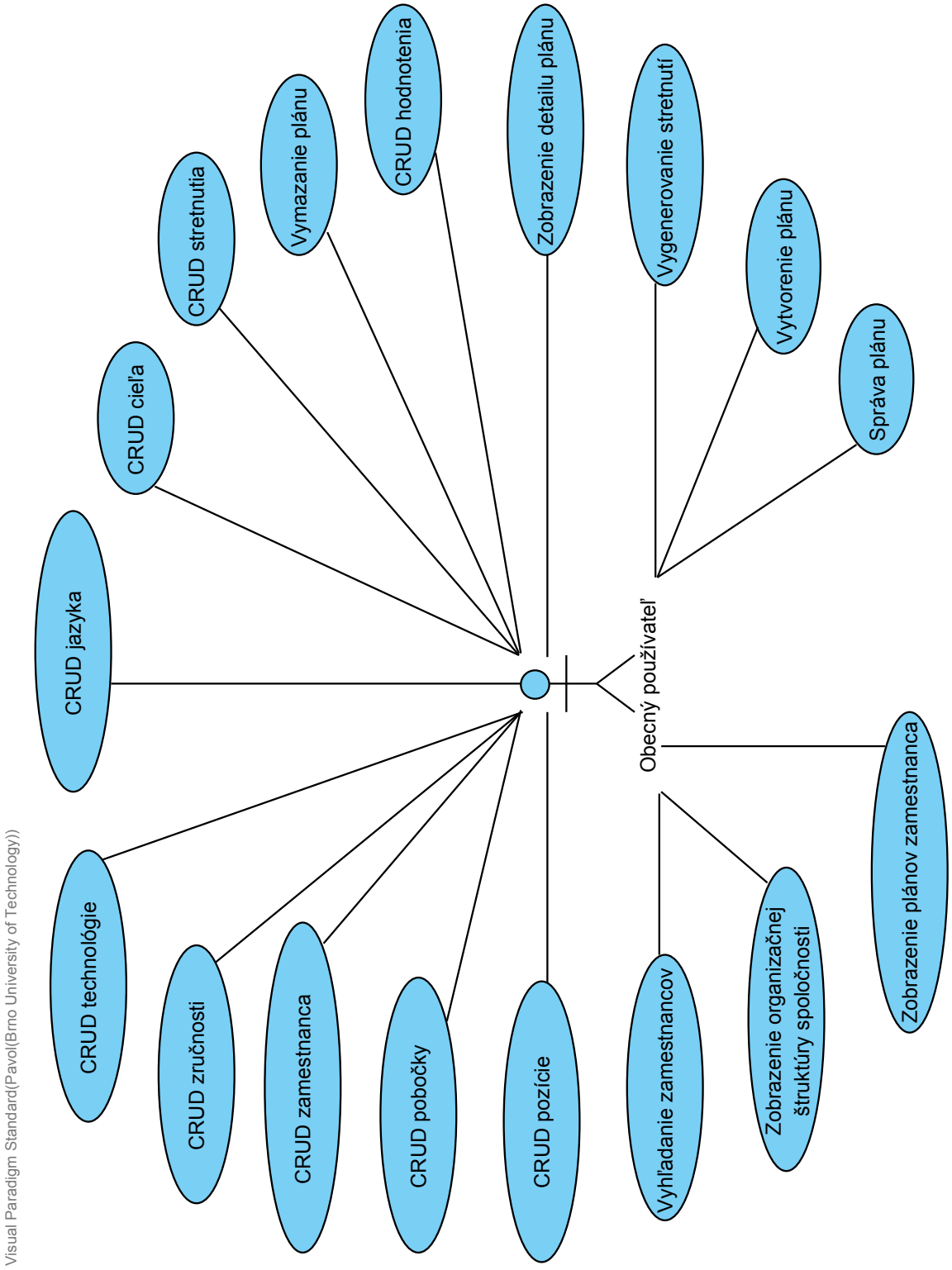
4.3 Plán vývoja

Plán vývoja aplikácie je rozdelený na dve iterácie. Z dôvodu prehľadnosti sú v prvej iterácii definované takmer všetky potrebné funkčné požiadavky podľa spomínaných kritérií. Na obr. 4.1 je znázornený diagram prípadov použitia pre prvú iteráciu. Vystupuje tu len jeden aktér predstavujúci obecného používateľa.

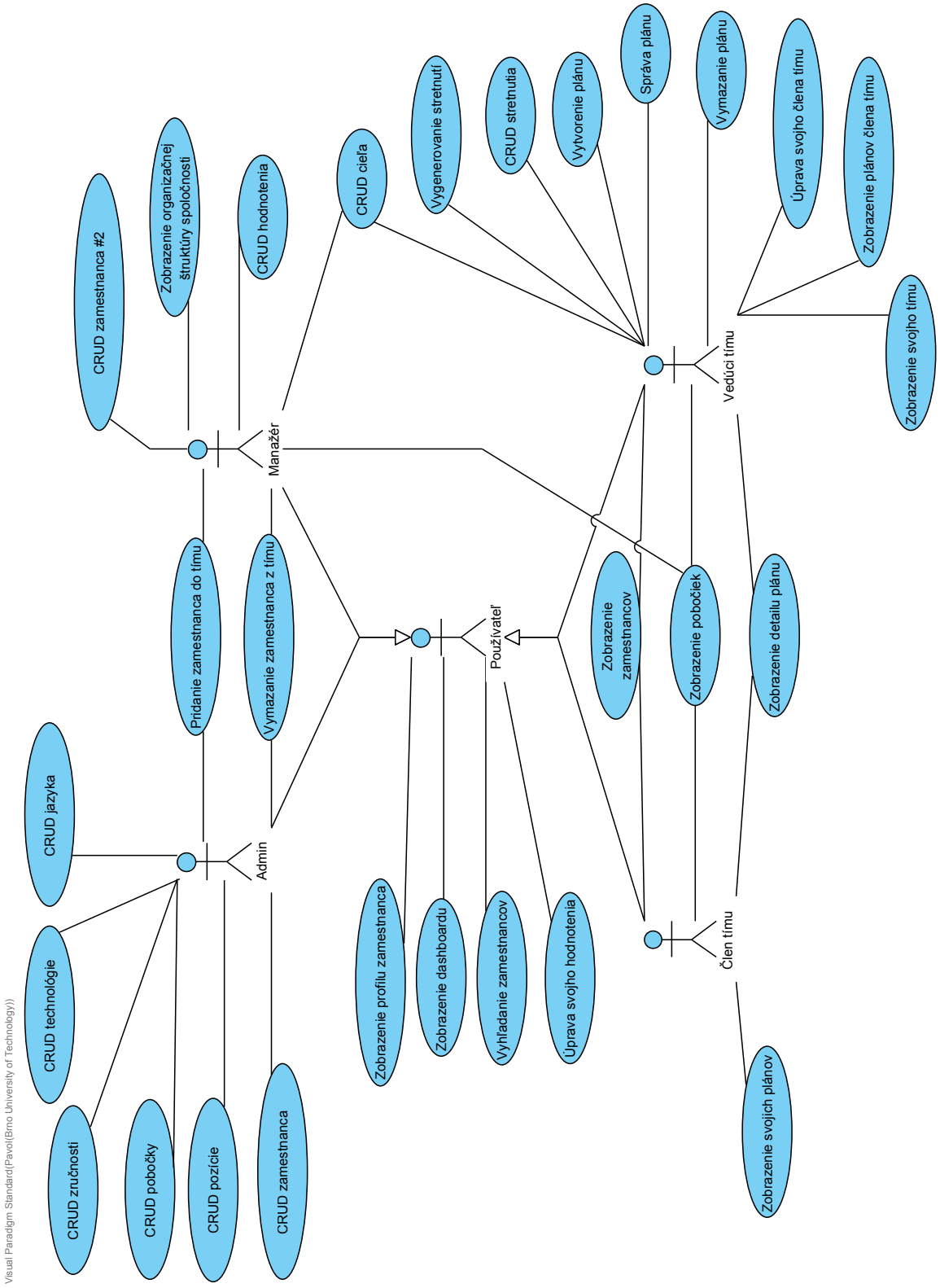
Následne v druhej (a zároveň poslednej) iterácii sú prípady použitia rozšírené o správu používateľov, ich autentizáciu a autorizáciu a niektoré ďalšie funkcie súvisiace už s konkrétnymi rolami používateľov. Diagram prípadov použitia pre túto iteráciu zobrazuje obr. 4.2.

Keďže detaily niektorých prípadov použitia nemusia byť na prvý pohľad jasné, je nutné ich upresniť, ideálne pomocou šablón nazývaných „špecifikácie prípadov použitia“, ktoré sú podstatným rozšírením diagramu prípadov použitia. V tabuľkách 4.1 – 4.4 sú špecifikované vybrané prípady použitia druhej iterácie (od prvej iterácie by sa líšili len aktérmi, ktorí by tu vystupovali).

Prípady „CRUD zamestnanca“ a „CRUD zamestnanca #2“ (obr. 4.2) majú podobný názov, ale líšia sa hlavne tým, že v druhom prípade môže používateľ (manažér) u daného zamestnanca pracovať len s jeho osobnými údajmi. Inak sú tieto prípady analogické a z tohto dôvodu je špecifikovaný len „CRUD zamestnanca“ (tab. 4.3).



Obr. 4.1: Diagram prípadov použitia prvej iterácie – CRUD zahŕňa akcie pre zobrazenie, vytvorenie, úpravu a vymazanie



Obr. 4.2: Diagram prípadov použitia zohľadňujúci jednotlivé roly v aplikácii – CRUD zahŕňa akcie pre zobrazenie, vytvorenie, úpravu a vymazanie

Názov	Vygenerovanie stretnutí
Opis	Vygeneruje stretnutia v rámci daného plánu.
Primárni aktéri	Vedúci tímu
Sekundárni aktéri	–
Predpoklady	1. Používateľ je prihlásený. 2. Používateľ má pridelené práva vedúceho tímu.
Akcie pre spustenie	Používateľ vyberie „Opakujúce sa stretnutia“ pri tvorbe plánu.
Hlavný tok	1. Systém zobrazí formulár s možnosťami opakovaného stretnutia (pravidelnosť, od, do a pod.). 2. Používateľ vyplní požadované údaje. 3. Systém overí zadané údaje. 4. Systém vygeneruje stretnutia podľa zadaných údajov.
Následné podmienky	Sú vytvorené nové stretnutia.
Alternatívny tok	–
Výnimky	1. Storno 2. Nesprávne údaje

Tabuľka 4.1: Špecifikácia prípadu použitia „Vygenerovanie stretnutí“

Názov	Vyhľadanie zamestnancov
Opis	Vyhľadá zamestnancov podľa zadaných kľúčových slov.
Primárni aktéri	Člen tímu, Vedúci tímu, Manažér, Admin
Sekundárni aktéri	–
Predpoklady	1. Používateľ je prihlásený. 2. Používateľ má pridelené práva člena tímu, vedúceho tímu, manažéra alebo admina. 3. V zozname zamestnancov musí existovať aspoň jeden zamestnanec.
Akcie pre spustenie	Používateľ zadá kľúčové slová do vyhľadávacieho poľa pri zozname zamestnancov.
Hlavný tok	1. Systém zobrazí zamestnancov, ktorí vyhovujú zadaným podmienkam.
Následné podmienky	Zobrazí sa zoznam hľadaných zamestnancov.
Alternatívny tok	–
Výnimky	1. Storno vyhľadávania

Tabuľka 4.2: Špecifikácia prípadu použitia „Vyhľadanie zamestnancov“

Názov	CRUD zamestnanca
Opis	Umožní vykonanie CRUD akcií nad zamestnancom.
Primárni aktéri	Admin
Sekundárni aktéri	–
Predpoklady	1. Používateľ je prihlásený. 2. Používateľ má pridelené práva admina.
Akcie pre spustenie	Používateľ vyberie položku „Zamestnanci“ v menu aplikácie.
Hlavný tok	1. Systém zobrazí zoznam zamestnancov s možnosťou „Pridať nového zamestnanca“ a možnosťami „Upraviť zamestnanca“ a „Vymazať zamestnanca“ pri jednotlivých zamestnancoch. 2. Ak používateľ vyberie „Pridať nového zamestnanca“ 2.1 Systém zobrazí formulár s možnosťou úpravy členov tímu, roly zamestnanca, jeho osobných údajov a oblastí osobného rozvoja. 2.2 Používateľ vyplní potrebné údaje. 2.3 Systém overí zadané údaje. 2.4 Systém vytvorí nového zamestnanca. 3. Ak používateľ vyberie pri zamestnancovi „Upraviť zamestnanca“ 3.1. Systém zobrazí formulár podobný ako pri 2.1 s predvyplnenými údajmi vybraného zamestnanca. 3.2. Používateľ upraví potrebné údaje. 3.3. Systém overí zadané údaje. 3.4. Systém upraví zvoleného zamestnanca. 4. Ak používateľ vyberie pri zamestnancovi „Vymazať zamestnanca“ 4.1. Systém zobrazí dialógové okno vyžadujúce potvrdenie akcie. 4.2. Pri potvrdení systém odstráni vybraného zamestnanca.
Následné podmienky	Zamestnanec je vytvorený, upravený alebo odstránený zo systému.
Alternatívny tok	1 – 2 Používateľ si môže zamestnanca vyhľadať.
Výnimky	1. Storno vytvorenia zamestnanca 2. Storno úpravy zamestnanca 3. Storno vymazania zamestnanca 4. Nesprávne údaje

Tabuľka 4.3: Špecifikácia prípadu použitia „CRUD zamestnanca“

Názov	Správa plánu
Opis	Umožní plánovanie stretnutí zamestnanca.
Primárni aktéri	Vedúci tímu
Sekundárni aktéri	–
Predpoklady	1. Používateľ je prihlásený. 2. Používateľ má pridelené práva vedúceho tímu.
Akcie pre spustenie	Používateľ vyberie položku „Správa plánu“ pri konkrétnom pláne zamestnanca.
Hlavný tok	1. Systém zobrazí stretnutia plánu s možnosťou výberu konkrétneho stretnutia a s možnosťami „Pridať nové stretnutie“ a „Vymazať vybrané stretnutie“. 2. Ak používateľ vyberie existujúce stretnutie 2.1 Systém zobrazí formulár s možnosťou úpravy údajov stretnutia a percent cieľov stretnutia. 2.2 Používateľ vyplní potrebné údaje. 2.3 Systém overí zadané údaje. 2.4 Systém upraví vybrané stretnutie. 3. Ak používateľ vyberie „Pridať nové stretnutie“ 3.1 Systém zobrazí formulár s výberom dátumu nového stretnutia. 3.2 Používateľ vyplní potrebné údaje. 3.3 Systém overí zadané údaje. 3.4 Systém vytvorí nové stretnutie. 4. Ak používateľ vyberie „Vymazať vybrané stretnutie“ 4.1. Systém zobrazí dialógové okno vyžadujúce potvrdenie akcie. 4.2. Pri potvrdení systém odstráni vybrané stretnutie.
Následné podmienky	Stretnutia plánu sú vytvorené, upravené alebo odstránené zo systému.
Alternatívny tok	–
Výnimky	1. Storno správy plánu 2. Storno vytvorenia stretnutia 3. Storno vymazania stretnutia 4. Nesprávne údaje

Tabuľka 4.4: Špecifikácia prípadu použitia „Správa plánu“

4.4 Návrh GraphQL schémy

Ako už bolo spomenuté, výsledná aplikácia bude využívať technológiu GraphQL pre aplikačné rozhranie. Preto je možné na základe špecifikácie a analýzy požiadaviek na výslednú aplikáciu určiť, s akými typmi bude pracovať. Oproti REST API, kde sa väčšinou navrhuje kolekcia koncových bodov, je lepšie sa na GraphQL API pozerať ako na kolekciu typov. Autori knihy *Learning GraphQL* ([20]) odporúčajú použiť tzv. *schema first* návrh, tj. najskôr začať s návrhom schémy – kolekcie typov. K tomuto účelu možno využiť jazyk SDL spomínaný v kapitole 3.3.

Ako prví sú spomenutí zamestnanci – každý z nich má pridelený titul, meno, dátum narodenia, e-mail, skóre a dve telefónne čísla. Tieto údaje možno reprezentovať poliami, ktoré vracajú vstavané skalárne typy. Člen tímu môže mať navyše vedúceho, tj. obecne zamestnanca. Pole `supervisor` viaže objekty typu `Employee` na samých seba. Kód 4.1 ukazuje, ako vyzerá tento typ v jazyku SDL.

```
type Employee {
    id: ID!
    title: String
    firstName: String!
    lastName: String!
    dateOfBirth: Date
    email: String!
    score: Int!
    phoneNumber1: String
    phoneNumber2: String
    supervisor: Employee
}
```

Kód 4.1: Typ reprezentujúci zamestnanca v jazyku SDL

Zamestnanci pracujú na pobočkách, ktoré sa nachádzajú v rôznych mestách a majú svoj názov. Pobočky teda možno zastúpiť typom `Branch` znázorneným v kóde 4.2. Keďže na jednej pobočke môže pracovať viacero zamestnancov, obsahuje tento typ pole `employees` vracajúce zoznam zamestnancov. Aby bolo možné dostať sa k pobočke cez daného zamestnanca, je nutné mu pridať pole vracajúce typ `Branch`.

```
type Employee {
    ...
    branch: Branch!
}
```

```
type Branch {
    id: ID!
    name: String!
    address: String!
    employees: [Employee]
}
```

Kód 4.2: Väzba medzi zamestnancom (vľavo) a pobočkou (vpravo)

Osobný rozvoj je rozdelený na tri oblasti: zručnosti (typ `Skill`), technológie (typ `Technology`) a jazyky (typ `Language`), ktoré ukazuje kód 4.3. Každá obsahuje názov a zoznam zamestnancov, pretože tú istú oblasť môže ovládať viacero zamestnancov. Pre podporu niekoľkých oblastí u zamestnanca treba do typu `Employee` pridať zoznamy pre jednotlivé oblasti.

```
type Skill {
    id: ID!
    name: String!
    employees: [Employee]!
}
```

```
type Technology {
    id: ID!
    name: String!
    employees: [Employee]!
}
```

```
type Language {
    id: ID!
    name: String!
    employees: [Employee]!
}
```

Kód 4.3: Typy reprezentujúce oblasti osobného rozvoja

Každý zamestnanec však môže mať odlišnú úroveň u jednotlivých oblastí, na čo je vhodné využiť tzv. väzobný typ (*through type*) bližšie špecifikujúci vzťah medzi zamestnancom a danou oblasťou [20]. V kóde 4.4 sú zobrazené tieto väzobné typy, pričom každý má vlastnú úroveň vyjadrenú vymenovaným typom (kód 4.5).

```
type SkillEmployee {
    level: Level!
    skill: Skill!
    employee: Employee!
}
```

```
type TechnologyEmployee {
    seniority: Seniority!
    technology: Technology!
    employee: Employee!
}
```

```
type LanguageEmployee {
    proficiency: Proficiency!
    language: Language!
    employee: Employee!
}
```

Kód 4.4: Väzobné typy reprezentujúce vzťah medzi oblasťami osobného rozvoja a zamestnancami

```
enum Level {
    BEGINNER
    LOWER_INTERMEDIATE
    UPPER_INTERMEDIATE
    LOWER_ADVANCED
    UPPER_ADVANCED
}
```

```
enum Seniority {
    TRAINEE
    JUNIOR
    MEDIOR
    SENIOR
    EXPERT
}
```

```
enum Proficiency {
    A1
    A2
    B1
    B2
    C1
    C2
}
```

Kód 4.5: Vymenované typy reprezentujúce úrovne jednotlivých oblastí

Následne je nutné prepojiť oblasti, ako aj zamestnancov so zodpovedajúcimi väzobnými typmi (kód 4.6, 4.7).

```
type Skill {
    ...
    skillEmployees: [
        SkillEmployee]!
}
```

```
type Technology {
    ...
    technologyEmployees: [
        TechnologyEmployee]!
}
```

```
type Language {
    ...
    languageEmployees: [
        LanguageEmployee]!
}
```

Kód 4.6: Výsledné typy reprezentujúce oblasti osobného rozvoja s väzbou na väzobné typy

```
type Employee {
    ...
    skillEmployees: [SkillEmployee]
    technologyEmployees: [TechnologyEmployee]
    languageEmployees: [LanguageEmployee]
}
```

Kód 4.7: Typ reprezentujúci zamestnanca s pridanými väzbami na väzobné typy

Pre pozíciu zamestnancov je definovaný typ `Position` (kód 4.8), ktorý obsahuje jej názov (pole `name`) a zoznam zamestnancov, keďže rovnakú pozíciu môže mať viacero zamestnancov. Jeden zamestnanec však musí pracovať len na jednej pozícii, čo je vyjadrené poľom `position` v type `Employee`. Pre úroveň skúseností pri pozícii je využitý vymenovaný typ `Seniority` (kód 4.5). Pole vracajúce tento typ je ale uvedené u zamestnanca, aby sa predišlo redundancii (keby bol u pozície, bolo by nutné uchovávať objekty s rovnakým názvom, ale odlišnou úrovňou).

```
type Employee {
    ...
    positionSeniority: Seniority!
    position: Position!
}
```

```
type Position {
    id: ID!
    name: String!
    employees: [Employee]
}
```

Kód 4.8: Väzba medzi zamestnancom (vľavo) a pozíciou (vpravo)

Typ `Goal` zastupuje jednotlivé ciele zamestnancov a obsahuje názov, opis a percentá (kód 4.9). Plán obsahujúci názov a opis môže evidovať viacej cieľov, čo predstavuje typ `Plan` a pole `goals`. Pri cieli je potom uvedený plán, ku ktorému patrí.

```
type Goal {
    id: ID!
    name: String!
    description: String
    percentage: Int!
    plan: Plan
}
```

```
type Plan {
    id: ID!
    name: String!
    description: String!
    goals: [Goal]!
}
```

Kód 4.9: Väzba medzi cieľom (vľavo) a plánom (vpravo)

Pri stretnutiach sa požaduje zaznamenávať miestnosť, dátum a čas. Okrem toho treba doplniť aj dĺžku trvania stretnutia a opis, resp. poznámky, ktoré bývajú na konci spísané. Keďže stretnutia sú podobne ako ciele evidované v rámci určitého plánu, obsahuje typ `Meeting` okrem vyššie uvedených údajov aj väzbu na plán a naopak (kód 4.10).

```
type Meeting {
    id: ID!
    description: String
    date: Date!
    startsAt: String!
    duration: String!
    meetingRoom: String!
    plan: Plan!
}
```

```
type Plan {
    ...
    meetings: [Meeting]!
}
```

Kód 4.10: Väzba medzi stretnutím (vľavo) a plánom (vpravo)

Zamestnanci pracujú na cieľoch a ich progres je preberaný na stretnutiach. Percentá uvedené v type `Goal` sú aktuálne, tj. na koľko percent je daný cieľ aktuálne splnený. Je

však tiež potrebné zachytiť, na koľko percent bol cieľ splnený ku konkrétnemu stretnutiu, čiže akúsi históriu progresu cieľa. K tomu sa ponúka využiť väzobný typ medzi stretnutím a cieľom, v ktorom sa budú zaznamenávať percentá. Kód 4.11 vyjadruje tento vzťah.

```
type Goal {
    ...
    goalHistories: [
        GoalHistory]
}
```

```
type GoalHistory {
    percentage: Int!
    meeting: Meeting!
    goal: Goal!
}
```

```
type Meeting {
    ...
    goalHistories: [
        GoalHistory]!
}
```

Kód 4.11: Väzba medzi cieľom (vľavo) a stretnutím (vpravo) pomocou väzobného typu (v strede)

Ďalej sa vyžaduje ukladať informácie o generovaní stretnutí. Jedná sa hlavne o rozsah generovania, tj. začiatočný a konečný dátum, periodicita opakovania (každý n-tý deň, týždeň či mesiac) a parametre, ktoré sa nastavia každému vygenerovanému stretnutiu (najmä začiatok, dĺžka trvania a miestnosť). Tieto informácie sa ukladajú hlavne kvôli možnosti neskoršieho pregenerovania stretnutí. Kód 4.12 zobrazuje doplnené polia v pláne.

```
type Plan {
    ...
    recurring: Boolean!
    recurrence: Recurrence
    repeatEvery: Int
    startDate: Date
    endDate: Date
    startsAt: String
    duration: String
    meetingRoom: String
}
```

```
enum Recurrence {
    DAILY
    WEEKLY
    MONTHLY
}
```

Kód 4.12: Plán doplnený o polia pre generovanie stretnutí (vľavo) a vymenovaný typ pre ich pravidelnosť (vpravo)

Následne zostáva vytvoriť prepojenie medzi zamestnancom a jeho plánom, ako aj vedúcim, ktoré je uvedené v kóde 4.13.

```
type Employee {
    ...
    plan: [Plan]!
}
```

```
type Plan {
    ...
    employee: Employee!
    supervisor: Employee!
}
```

Kód 4.13: Väzba medzi zamestnancom (vľavo) a plánom (vpravo)

Hodnotenia sú podobné plánom, zaznamenáva sa u nich však len názov, opis a dátum. V rámci hodnotení môže byť upravovaných niekoľko cieľov, pri generovaní však budú ciele pridelené viacerým hodnoteniam. Z tohto dôvodu je medzi hodnotením a cieľom podobný

vzťah, ako medzi stretnutím a cieľom, tj. je nutné medzi nimi vytvoriť väzobný typ (kód 4.14).

```
type Assessment {
    id: ID!
    name: String!
    description: String!
    endDate: Date
    assessmentGoals: [
        AssessmentGoal]!
}
```

```
type AssessmentGoal {
    percentage: Int!
    description: String
    assessment: Assessment!
    goal: Goal!
}
```

Kód 4.14: Väzba medzi hodnotením (vľavo) a cieľom pomocou väzobného typu (vpravo)

Do hodnotenia sú následne pridané polia pre hodnotiteľa (`assessor`) a hodnoteného (`assessee`). Pre možnosť priamo získať hodnotenia zamestnanca ako hodnotiteľa, sa pri zamestnancovi uvádza pole `assessments` (kód 4.15).

```
type Employee {
    ...
    assessments: [Assessment]!
}
```

```
type Assessment {
    ...
    assessor: Employee!
    assessee: Employee!
}
```

Kód 4.15: Väzba medzi zamestnancom (vľavo) a hodnotením (vpravo)

Boli vymenované a opísané typy (a väzby medzi nimi) tvoriace schému zodpovedajúcu prvej iterácii vývoja aplikácie. V poslednej iterácii je schéma rozšírená o typ `User`, ktorý oddeľuje prihlasovacie údaje spolu s rolami od zamestnanca, čo umožňuje jednoduchšiu manipuláciu s účtom. Typ `User` a typ `Employee` sú teda vo vzťahu 1:1 (kód 4.16).

```
type Employee {
    ...
    user: User!
}
```

```
type User {
    id: ID!
    login: String!
    password: String!
    roles: [Role]!
    employee: Employee!
}
```

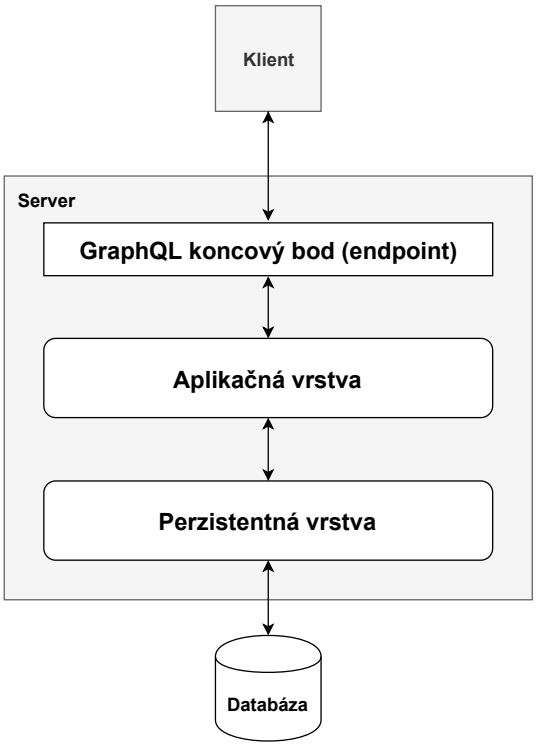
```
enum Role {
    MEMBER
    LEADER
    MANAGER
    ADMIN
}
```

Kód 4.16: Väzba medzi zamestnancom (vľavo) a používateľom (v strede) a vymenovaný typ pre roly používateľa (vpravo)

4.5 Návrh architektúry

Výslednú architektúru aplikácie možno obecne rozdeliť na 3 hlavné časti: klienta, server a databázu (obr. 4.3). Klient komunikuje so serverom prostredníctvom GraphQL konco-

vého bodu (*endpoint*), tj. všetky požiadavky sú naň smerované[2]. Server sa skladá z niekoľkých vrstiev. Tento prístup umožňuje rozdeliť zodpovednosti medzi jednotlivé vrstvy, vďaka čomu je aplikácia jednoduchá na zostavenie a údržbu [24]. Každá vrstva interaguje s nižšou vrstvou použitím presne definovaného kontraktu.



Obr. 4.3: Obecný návrh architektúry aplikácie

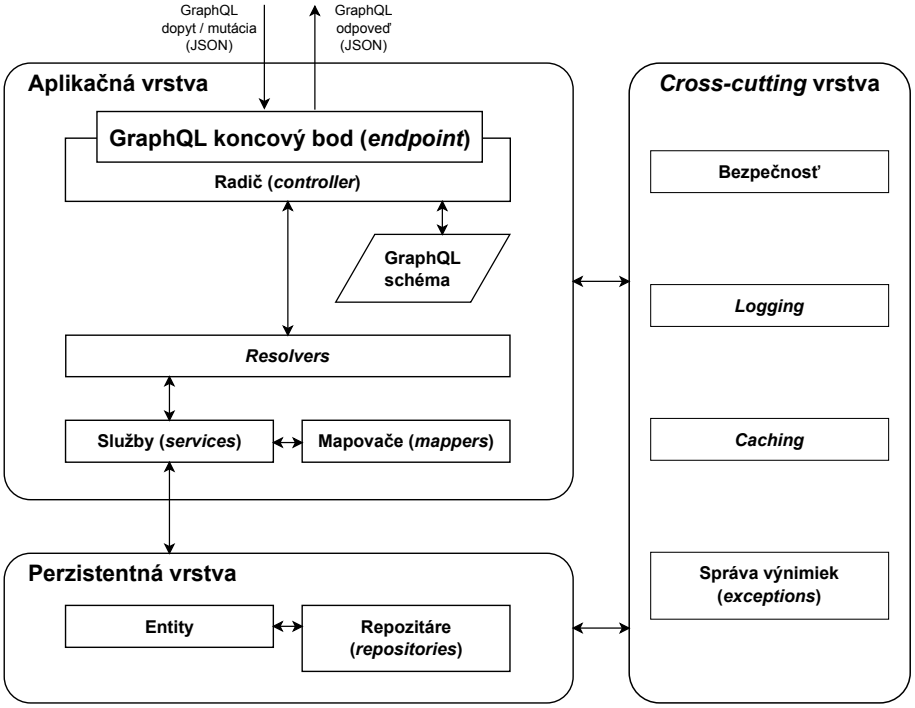
Detailnejší návrh architektúry servera je zobrazený na obrázku 4.4. Aplikačná vrstva je zodpovedná za príjem a spracovanie požiadaviek od klienta. K tomu účelu využíva radič, ktorý oproti REST architektúre „počúva“ požiadavky typicky na jednom koncovom bode [20]. Požiadavka je spracovaná a overená voči GraphQL schéme, následne sa volá príslušný *resolver* – funkcia vracajúca dáta v type a tvare špecifikovanom schémou. *Resolvery* používajú služby, ktoré navzájom komunikujú a abstrahujú CRUD operácie. Okrem toho sú zodpovedné za správu transakcií a iné „prierezové“ (*cross-cutting*) záležitosti, ako napr. bezpečnosť. Pre konverziu medzi dátovými typmi pracujú služby s mapovačmi.

Perzistentná vrstva má za úlohu interakciu s databázou a pozostáva najmä z entít (tried, ktoré sa mapujú na databázové tabuľky) a repozitárov, ktoré poskytujú CRUD operácie pre získavanie a ukladanie objektov z/do databázy.

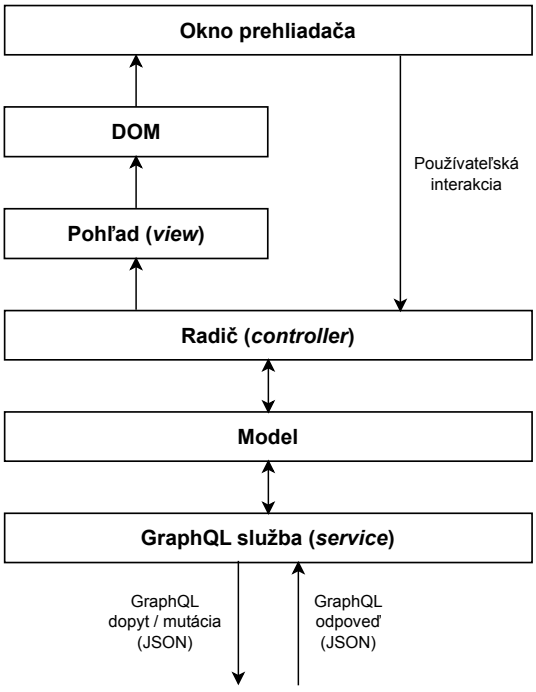
Rôzne funkcie ako bezpečnosť, *logging*, *caching* či správa výnimiek sú využiteľné naprieč rôznymi vrstvami aplikácie – z tohto dôvodu sú umiestnené v tzv. *cross-cutting* vrstve.

Pre klientskú architektúru je použitý architektonický vzor MVC (*Model-View-Controller*), ktorého hlavnými stavebnými blokmi sú: model, pohľad (*view*) a radič (*controller*). Tento vzor býva typicky aplikovaný na serverovú časť aplikácií, avšak v poslednej dobe sa používa pre správu rastúcej „bohatosti“ a zložitosti aj na klientskej strane webových aplikácií [10].

[2] https://graphql.org/learn/serving-over-http/



Obr. 4.4: Bližší pohľad na architektúru serverovej časti aplikácie



Obr. 4.5: Detail architektúry klientskej časti aplikácie [10]

Komunikácia so serverom prebieha prostredníctvom GraphQL služieb, ktoré oproti REST službám definujú operácie typicky pre jeden koncový bod (ako v prípade servera),

použitím rovnakej HTTP metódy (POST) [20]. Model by mal okrem doménových dát obsahovať aj logiku pre prácu (CRUD) s doménovými dátami [10], ktorá je však zastúpená GraphQL službami. Cieľom radiča a pohľadu je operovať nad dátami v modeli pre manipuláciu s DOM-om[3], tj. tvorba a správa HTML elementov, s ktorými môže používateľ pracovať [10]. Tieto interakcie sú vrátené späť do radiča, čo uzatvára slučku pre vytvorenie interaktívnej aplikácie. Obrázok 4.5 ukazuje takto navrhnutú architektúru klienta.

4.6 Návrh databázy

Pre aplikáciu je zvolená relačná databáza, ktorá pozostáva z tabuliek (relácií). Návrh týchto tabuliek je prezentovaný pomocou ER (*Entity-Relationship*) diagramu – nástroja modelujúceho dáta vo forme entít a vzťahov [2].

Na základe typov v GraphQL schéme a vzťahov medzi nimi (kapitola 4.4), je navrhnutý ER diagram. Typy sú reprezentované entitami a vzťahy medzi nimi nie sú vyjadrené pomocou poľa (atribútu), ale sú modelované čiarami s vyznačenou kardinalitou. Väzby typu 1:1 a 1:M sú modelované pomocou cudzích kľúčov u príslušných entít a väzobné typy (M:N) sú zastúpené väzobnými tabuľkami. Čo sa týka atribútov (polí), zostávajú rovnaké, líšia sa však dátovými typmi (návratovými typmi polí). Navrhnutý ER diagram pre prvú a druhú iteráciu je zobrazený v prílohe B.

4.7 Návrh používateľského rozhrania

Po špecifikácii požiadaviek a typov, ktoré budú v aplikácii vystupovať, možno prejsť k návrhu používateľského rozhrania (UI – *user interface*). Autor knihy *Designed for Use* ([16]) rozdeľuje návrh rozhraní na niekoľko častí (etáp): náčrty, *wireframy*, *mock-upy* (a prototypy). V nasledujúcich kapitolách sú jednotlivé časti krátko opísané spolu s ďalšími prístupmi, použitými pri návrhu. Obsahujú konkrétne ukážky niektorých návrhov, zvyšné možno nájsť v prílohe C.

Sekcie a podsekcie

Ešte pred kreslením jednotlivých náčrtov je potrebné zamyslieť sa nad hlavnými sekciami a podsekciami, pre ktoré budú vytvorené obrazovky. K tomu možno využiť diagramy prípadov použitia spolu so špecifikáciami, kde už je naznačená určitá štruktúra (napr. tab. 4.3). Tabuľka 4.5 obsahuje navrhnuté sekcie a ich podsekcie pre prvú iteráciu, ktoré budú reprezentované obrazovkami. V druhej iterácii sú rozšírené o ďalšie sekcie (tab. 4.6).

Responzívny dizajn

Keďže výsledná aplikácia má pracovať naprieč rôznymi zariadeniami a veľkosťami obrazoviek, najlepšou voľbou je použiť responzívny dizajn (*responsive web design*) [19]. Práve vďaka nemu možno navrhovať a tvoriť stránky, ktoré reagujú (*respond*) na šírku obrazovky zariadenia a zobrazujú obsah spôsobom vhodným pre aktuálnu veľkosť obrazovky. S responzívnym dizajnom sa spája aj tzv. *mobile first* prístup, teda najskôr začať s návrhom pre mobily. Tieto zariadenia majú viacero obmedzení a je náročnejšie vytvoriť dobrý návrh

[3] *Document Object Model* – základné API pre reprezentáciu a manipuláciu s obsahom HTML a XML dokumentov, kde elementy týchto dokumentov sú reprezentované stromom objektov [8]

pre limitovaný priestor, jednoduchšie však následne doplniť ďalšiu funkcionalitu so zväčšujúcim sa priestorom.

Hlavná sekcia	**Podsekcie**	**Poznámky**
Hodnotenia (zoznam)	Pridať, upraviť	• CRUD cieľov v rámci podsekcie „pridať“ • Generovanie hodnotení je súčasťou podsekcie „pridať“ • Vymazanie v rámci hlavnej sekcie
Jazyky (zoznam)	Pridať, upraviť	• Vymazanie v rámci hlavnej sekcie
Org. štruktúra	Detail	–
Plány (zoznam)	Pridať, spravovať, detail	• CRUD cieľov v rámci podsekcie „pridať“ • CRUD stretnutí v rámci podsekcie „spravovať“ • Generovanie stretnutí je súčasťou podsekcie „pridať“ • Vymazanie v rámci hlavnej sekcie
Pobočky (zoznam)	Pridať, upraviť	• Vymazanie v rámci hlavnej sekcie
Pozície (zoznam)	Pridať, upraviť	• Vymazanie v rámci hlavnej sekcie
Technológie (zoznam)	Pridať, upraviť	• Vymazanie v rámci hlavnej sekcie
Zamestnanci (zoznam)	Pridať, upraviť, detail	• Vyhľadanie v rámci hlavnej sekcie • Vymazanie v rámci hlavnej sekcie
Zručnosti (zoznam)	Pridať, upraviť	• Vymazanie v rámci hlavnej sekcie

Tabuľka 4.5: Hlavné sekcie a podsekcie aplikácie (zoradené v abecednom poradí) pre prvú iteráciu

Material Design

Material Design je koncepcia návrhu mobilných a webových aplikácií vytvorená spoločnosťou Google. Názov Material Design je odvodený od slova „materiál“, ktorý existuje v 3D

Hlavná sekcia	Podsekcie	Poznámky
Dashboard	–	–
Moje hodnotenia (zoznam)	Upraviť	–
Moje plány (zoznam)	Detail	–
Môj tím (zoznam)	Upraviť, detail	–

Tabuľka 4.6: Hlavné sekcie a podsekcie aplikácie (zoradené v abecednom poradí) pre druhú iteráciu

priestore, čo dodáva jednotlivým komponentom zmysel pre hĺbku a štruktúru, platia preň fyzikálne zákony – môže sa rozdeliť na časti a opäť spojiť, meniť farbu a tvar [17].

Na oficiálnej stránke[4] možno nájsť množstvo pravidiel pre tvorbu jednotlivých komponentov, ako aj prípadové štúdie, na ktoré sú princípy Material Design aplikované. Pravidlá boli väčšinou použité aj v rámci návrhov uvedených nižšie až na výnimky, kedy bolo potrebné prispôsobiť rozmiestnenie určitých komponentov vlastným potrebám.

Náčrty

Náčrty alebo *sketche* sú akoukoľvek reprezentáciou používateľského rozhrania formou kresby [16]. Sú určené najmä k definícii základnej štruktúry produktu a pomocou nich možno získať predstavu o jednotlivých obrazovkách, prípadne akú funkcionalitu majú poskytovať. Náčrty v tejto kapitole sú len orientačné, k mobilným verziám sú uvedené len tie desktop verzie, ktoré sa od nich odlišujú.

Na konzultáciách boli vytvorené náčrty tohto typu, pričom do procesu ich tvorby bol zapojený aj zákazník. Mal tak možnosť vyjadriť svoje názory hneď zozačiatku – oproti prototypom či implementácii nie je nutné vykonávať toľko zmien. Pôvodné náčrty boli „prekreslené“ do digitálnej podoby pomocou programu – na konzultáciách boli kreslené perom na papier.

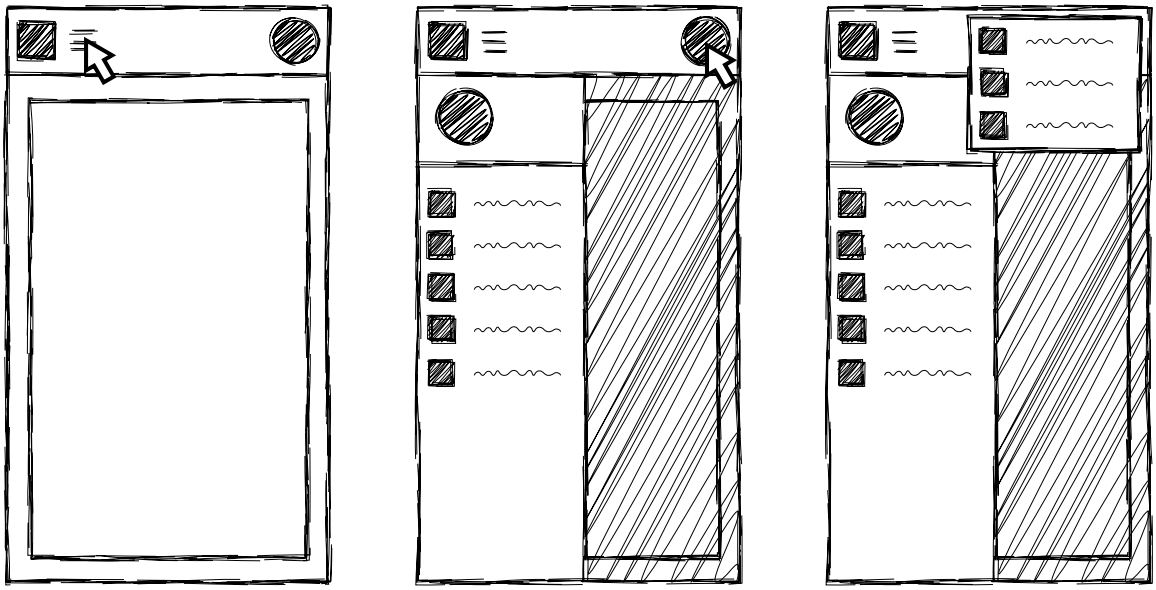
Najprv bolo potrebné vytvoriť náčrt základnej štruktúry aplikácie. Tá je pre obe iterácie rovnaká a pozostáva z hlavičky, ľavého menu a samotného obsahu. Hlavička obsahuje logo aplikácie, ikonu pre otvorenie/zatvorenie menu a obrázok používateľa. Po kliknutí naň sa otvorí menu, ktoré obsahuje položky týkajúce sa práce s účtom. Ľavé menu sa skladá z obrázku používateľa a položiek slúžiacich pre navigáciu v aplikácii. Interakcia znázornená prostredníctvom sekvencie snímok sa nazýva *storyboarding* (obr. 4.6). V náčrte pre väčšie obrazovky (tablet, desktop – obr. 4.7) sa vedľa loga nachádza aj názov aplikácie. V prípade desktopu je ľavé menu oproti mobilnej verzii implicitne otvorené.

Nasleduje návrh obrazoviek pre jednotlivé sekcie. Väčšina hlavných sekcií predstavuje zoznam (tab. 4.5 aj 4.6), ktorý môže byť ideálne reprezentovaný tabuľkou (obr. C.1). Na konci jej riadkov sú umiestnené tlačidlá slúžiace pre operácie s konkrétnymi záznamami. Nad tabuľkou vpravo je tlačidlo pre menu s hromadnými či inými akciami.

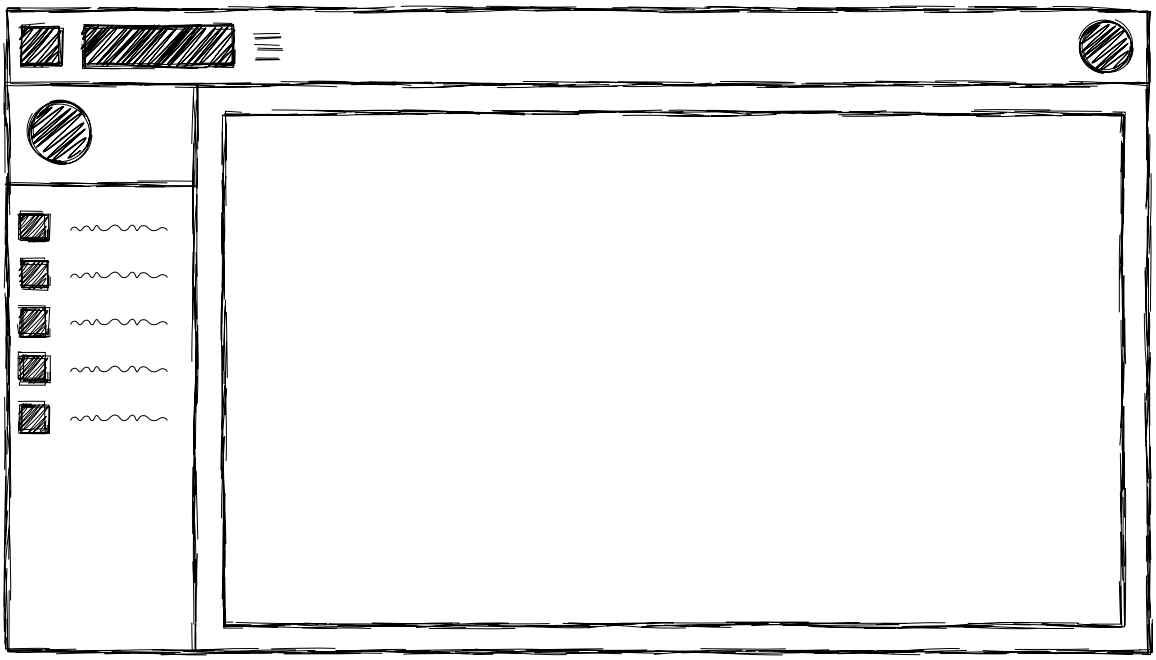
Z hlavných sekcií v prvej iterácii zostáva organizačná štruktúra, ktorá hierarchicky zobrazuje zamestnancov spoločnosti, typicky má tvar stromu. Jednotlivé uzly obsahujú stručné informácie o zamestnancovi. Po kliknutí na uzol sa zobrazí dialógové okno s obrázkom zamestnanca a bližšími informáciami (obr. C.2).

[4] https://material.io/design/

Typy pre jazyky, pozície, technológie a zručnosti (kapitola 4.4) obsahujú najmenej a zároveň rovnaké polia, takže podsekcie „pridať“ a „upraviť“ sú v tomto prípade jednoduché. Zahŕňajú formulár len s jedným poľom pre názov a tlačidlá pre potvrdenie a zrušenie (obr. C.3).



Obr. 4.6: Náčrt (*storyboard*) aplikácie pre mobilné zariadenia – základná štruktúra



Obr. 4.7: Náčrt štruktúry aplikácie pre desktop zariadenia

Pri pobočkách má zmysel priamy výber zamestnancov, ktorý v nich sídlia. Po kliknutí na tlačidlo pre výber sa otvorí dialógové okno s možnosťou vyhľadania a výberu potrebných zamestnancov (obr. C.4). Po potvrdení sa vypíšu vybraní zamestnanci pod tlačidlo. Náčrt pre desktop verziu je zobrazený na obr. C.5.

Zamestnanci sú zložitejšou štruktúrou – obsahujú hneď niekoľko atribútov, pre ktoré je nutné vytvoriť zodpovedajúce formulárové polia. V mobilnej verzii je pre každé pole vyhradený samostatný riadok, no pri väčšom priestore sú súvisiace polia umiestnené vedľa seba (obr. C.6). V podsekcii pre pridanie či úpravu zamestnanca sa teda okrem jeho osobných údajov vyskytuje aj možnosť správy osobného rozvoja, ako aj výberu členov tímu. Každá z vymenovaných častí je vizuálne oddelená pomocou osobitných panelov, kde kliknutím možno zobraziť/skryť danú časť (obr. C.7). Časť pre správu osobného rozvoja pozostáva z 3 tlačidiel (pre každú z oblastí, ktoré možno rozvíjať). Po kliknutí na príslušné tlačidlo sa otvorí dialógové okno pre výber konkrétneho druhu príslušnej oblasti a jej úrovne (obr. C.8). Po potvrdení sa výber vypíše pod tlačidlo. Výber členov tímu funguje podobne ako pri pobočkách (obr. C.4). Detail zamestnanca je ďalšou z podsekcii zamestnancov. Je možné ho zobraziť kliknutím na dané tlačidlo pri niektorom zamestnancovi (obr. C.9). Má formu dialógového okna, ktoré obsahuje jeho obrázok a ďalšie informácie o ňom.

Plány môžu zahŕňať správu aj iných typov, ktoré sú s nimi spojené (tab. 4.5). Podsekcia „pridať“ umožňuje okrem samotného plánu aj vytvoriť ciele či vygenerovať stretnutia. Obsahuje formulárové polia reprezentujúce základné informácie o pláne, zaškrtávacie pole (*checkbox*), po ktorého zaškrtnutí sa odkryjú možnosti pre generovanie stretnutí a nakoniec pridávanie či vymazávanie cieľov pre konkrétny plán (obr. C.10). Po rozšírení obrazovky na desktop verziu sú súvisiace polia umiestnené vedľa seba a tlačidlo pre pridanie cieľa je zarovnané na ľavú stranu (p. obr. C.11). Podsekcia pre správu plánu umožňuje plánovanie a úpravu stretnutí, ako aj zmenu percent jednotlivých cieľov v rámci stretnutia. Každému stretnutiu je pridelená karta (*tab*), na konci ktorej je interaktívny textový editor pre písanie poznámok k stretnutiu (obr. C.12). Pridanie a vymazanie stretnutia je uskutočniteľné prostredníctvom akcii v menu, ktoré sa zobrazí po kliknutí na tlačidlo v pravom hornom rohu. Detail plánu je analogický ku správe s tým rozdielom, že údaje nemožno meniť, namiesto textového editoru je zobrazený formátovaný text a menu v pravom hornom rohu nie je k dispozícii.

Hlavné sekcie typu zoznam v druhej iterácii sú (ako v prípade prvej iterácie) reprezentované tabuľkami (obr. C.1). Podsekcie sú zase podobné už spomínaným, napr. „upraviť“ pri hlavnej sekcii „môj tím“ je podobné sekcii „upraviť“ pri „zamestnanci“ a pod.

Na obrázku C.14 vľavo je zobrazený náčrt pre mobilnú verziu *dashboardu*[5], ktorý pozostáva z 5 častí (podľa poradia):

- základné informácie o aktuálne prihlásenom používateľovi
- graf zobrazujúci celkový progres na cieľoch v rámci zvoleného plánu
- prehľad stretnutí
- členovia tímu
- oblasti osobného rozvoja

Pri desktop verzii sa vyššie uvedené časti preskupia podľa obrázka C.14 vpravo. Druhá iterácia pridáva správu používateľov a spolu s ňou aj správu rolí, ktorá rozširuje podsekciu

[5]podľa [6]: „Dashboard je zobrazenie najdôležitejších informácií potrebných pre dosiahnutie jedného alebo viacerých cieľov; usporiadaný na jednej obrazovke tak, že informácie možno sledovať na prvý pohľad.“

„pridať“/„upraviť“ zamestnanca vo forme panelu s jednotlivými rolami (obr. C.13). Nakoniec je nutné pridať náčrt obrazovky pre prihlásenie, ktorý sa skladá z loga aplikácie a polí pre zadanie prihlasovacích údajov (obr. C.15).

Wireframy

Oproti náčrtom, ktoré predstavovali základný návrh obrazoviek, reprezentujú tzv. „drôtové modely“ (ďalej len *wireframy*) presnejšiu štruktúru obrazovky. Nemali by sa tu však zobrazovať farby, tiene či obrázky, ide najmä o obsah (text), kde ho zahrnúť a umiestniť. Z náčrtov, ktoré postupne vznikli na konzultáciách boli vytvorené *wireframy*, vylepšené o menšie nedostatky vyskytujúce sa v náčrtoch. Tie boli opäť prebrané so zákazníkom a ich výsledné podoby sa nachádzajú v tejto kapitole (a prílohe C.2).

V základnej štruktúre aplikácie je potrebné bližšie špecifikovať položky, ktoré sa majú nachádzať v ľavom menu. Tie je možné určiť na základe hlavných sekcií v tabuľke 4.5, no dôležité je aj ich poradie. Ideálnejšie je zoradiť ich podľa pravdepodobnej frekvencie použitia, ako podľa abecedy (obr. 4.8).

Obr. 4.8: *Wireframe* aplikácie pre mobilné zariadenia – základná štruktúra

V predošlej kapitole bola obecne načrtnutá tabuľka pre zoznamy, kde však nie sú konkretizované názvy jednotlivých stĺpcov. *Wireframy* pre štruktúry, pri ktorých sa ukazuje len názov, sú zobrazené na obr. C.16 a C.17. Ide o jazyky, pozície, technológie, zručnosti a plány, pri pobočkách sa okrem názvu zobrazuje aj adresa. Plány naviac obsahujú menu pre výber zamestnanca (obr. C.17). Zamestnanci majú viacero osobných údajov, podľa ktorých ich možno aj vyhľadávať (obr. C.18). Stĺpce v tabuľkách sa dajú rolovať, najmä ak sa nezmestia na obrazovku.

Každý z uzlov organizačnej štruktúry pozostáva z mena, pozície a pobočky zamestnanca (obr. C.19 vľavo). Dialógové okno s detailom zamestnanca je rovnaké aj pre zoznam zamestnancov (obr. C.19 vpravo).

Podsekcie pre jazyky, pozície, technológie a zručnosti obsahujú jedno formulárové pole pre názov (obr. C.20), pobočky navyše obsahujú adresu a tlačidlo pre výber zamestnancov (obr. C.21). Formulárové polia sú od tlačidiel odlíšené dvojitou čiarou.

Wireframy jednotlivých častí podsekcie pre pridanie alebo upravenie zamestnanca sú na obrázkoch C.22, C.23 a C.24, pridanie a spravovanie plánu zase na obrázkoch C.25 a C.26.

Ako bolo spomenuté v predošlej kapitole, *dashboard* pozostáva z 5 častí. Základné informácie o prihlásenom používateľovi obsahujú obrázok používateľa, meno a priezvisko, pozíciu, nadriadeného a skóre (obr. C.28). Úroveň pozície je reprezentovaná počtom hviezdičiek. Pre graf znázorňujúci celkový progres v rámci zvoleného plánu je vybraný tzv. *burndown* graf[6], pomocou ktorého možno ľahko a jasne zobraziť dve najdôležitejšie časti: koľko percent z cieľov zostáva a koľko času je treba na dokončenie cieľov [3]. V časti pre stretnutia sa nachádza dátum, miestnosť a opis nadchádzajúceho a posledného stretnutia. „Môj tím“ obsahuje zoznam zamestnancov, ktorí sú v rovnakom tíme ako prihlásený zamestnanec. Každú položku tvorí obrázok, meno, pozícia a pobočka zamestnanca. Osobný rozvoj sa skladá z troch častí, pri každej je výpis konkrétnych druhov oblastí zamestnanca.

Druhá iterácia pridáva okrem *dashboardu* správu rolí (obr. C.27) a prihlásenie (obr. C.29). Tiež pridáva sekcie „moje plány“, „môj tím“ a „moje hodnotenia“, pre ktoré návrhy vytvorené neboli (sú podobné už spomínaným sekciám, akurát sa zobrazujú len vlastné objekty). Zoradenie položiek v rámci menu pre druhú iteráciu je zobrazené na obr. C.30 – prednosť majú „vlastné“ sekcie, nakoľko sa u nich predpokladá častejšia návštevnosť.

Mock-upy

Oproti *wireframom* pridávajú *mock-upy* dekorácie alebo vizuálne detaily: tiene, textúry, obrázky, priehľadnosť [16]. Všetky tieto prvky neslúžia len pre „skrášlenie“ vzhľadu aplikácie – napr. tiene môžu byť použité pre zvýraznenie hierarchie a farby pre upútanie pozornosti používateľa.

Ukážkové *mock-upy* sa nachádzajú v prílohe C. Z kapacitných dôvodov tejto práce boli vybrané len niektoré, konkrétne obrazovky pre úpravu zamestnanca (obr. C.31) a pridanie hodnotenia (obr. C.32).

[6] https://www.scrumdesk.com/is-it-your-burn-down-chart/

Kapitola 5

Implementácia serverovej časti aplikácie

Táto kapitola sa venuje implementácii aplikácie z pohľadu serveru. Vychádza z rôznych návrhov definovaných v predchádzajúcej kapitole (4). Sú tu opísané technológie, použité pri vývoji a štruktúra aplikácie, rozdelená do viacerých častí (modulov). Nasleduje nastavenie techológie GraphQL, s ktorým súvisia aj niektoré hlavné prvky aplikácie, ako napr. schéma či *resolvery*. Na informácie o hlavných prvkoch nadväzuje kapitola o implementácii vybraných funkcií aplikácie, kde sú definované a opísane ďalšie typy týkajúce sa schémy GraphQL.

5.1 Použité technológie

Pre implementáciu serverovej časti aplikácie bol zvolený jazyk Java v spolupráci s *frameworkom* Spring Boot, pre databázovú časť systém MySQL a pre samotný vývoj prostredie IntelliJ IDEA. Nižšie sú stručne opísané každé z týchto technológií, ostatné technológie sú v rámci použitia opísané v ďalších kapitolách.

Spring Boot

Framework Spring sa za posledné desaťročie stal skutočne štandardom pre vývoj Java aplikácií [25]. Začal ako jednoduchá alternatíva k *Java Enterprise Edition* (JEE). Namiesto vývoja komponentov ako „ťažkopádnych“ *Enterprise Java Beans* (EJB), ponúka Spring jednoduchší prístup k vývoju *enterprise* Java aplikácií pomocou vkladania závislostí (*dependency injection*) a aspektovo orientovaného programovania pre dosiahnutie možností EJB za použitia obyčajných Java objektov (POJO[1]).

Ako názov napovedá, *Dependency Injection* umožňuje vložiť závislosti do komponentov podľa potreby [24]. Týmto spôsobom komponenty nie sú nútené vytvárať a lokalizovať závislosti, vďaka čomu môžu byť voľne spojené. Klasickým prístupom je inicializácia potrebných závislostí napr. v konštruktore, čo vedie k ťažko udržovateľnému, ťažko testovateľnému a úzko spojenému kódu. Pomocou vkladania závislostí však Spring dokáže automaticky vytvoriť, spravovať a vložiť závislosti do objektov, ktoré ich potrebujú.

[1] *Plain Old Java Object* – bežný Java objekt bez zvláštnych požiadaviek, ako napr. dedičnosť špecifickej triedy alebo implementácia špecifického rozhrania [14]

Aspektovo orientované programovanie je programovací model, ktorý implementuje *cross-cutting* logiku, kde patrí napr. *logging*, transakcie, metriky, bezpečnosť či iné záležitosti. Poskytuje štandardizovaný mechanizmus nazývaný aspekt pre zapuzdrenie týchto záležitostí do jedného miesta. Aspekty sú následne poskladané do iných objektov, takže *cross-cutting* logika je automaticky aplikovaná naprieč celou aplikáciou.

Spring síce zjednodušuje kód komponentov, ale konfiguráciu rôznych prvkov (napr. správa transakcií) je nutné explicitne špecifikovať, a to buď použitím XML alebo Javy [25]. Detekcia komponentov redukovala konfiguráciu, ale Spring stále vyžadoval množstvo konfigurácie. Ďalším problémom je špecifikácia správnych verzií závislostí, keďže niektoré verzie konkrétnch knižníc nemusia byť kompatibilné s inými.

Spring Boot vylepšuje vývoj Spring aplikácií vo viacerých smeroch. Prináša štyri zásadné vylepšenia:

- **automatická konfigurácia** – odstraňuje nutnosť explicitnej konfigurácie často používaných prvkov či funkcií, ktoré sú spoločné pre väčšinu aplikácií
- **štartovacie (*starter*) závislosti** – špeciálne Maven (alebo Gradle) závislosti, ktoré využívajú tranzitívne závislosti pre zoskupenie bežne používaných knižníc do jedného celku (ich verzie boli spolu otestované)
- **rozhranie príkazového riadka** – voliteľný prvok umožňujúci písanie celej aplikácie pomocou aplikačného kódu, bez potreby tradičného zostavenia projektu
- **aktivátor (*actuator*)** – ponúka pohľad do vnútorných častí aplikácie počas jej behu

MySQL

MySQL je jedným z najpopulárnejších relačných systémov riadenia bázy dát (RSRBD) využívajúci jazyk SQL (*Structured Query Language*) pre vkladanie, získavanie a úpravu dát [23]. Vývojový proces MySQL sa zameriava na poskytnutie veľmi efektívnej implementácie často vyžadovaných prvkov. Výhodou MySQL je veľkosť a rýchlosť, jednoduchosť inštalácie, dodržiavanie štandardov, podpora komunity a jednoduché rozhranie pre iný *software*.

IntelliJ IDEA

IntelliJ IDEA je populárne vývojové prostredie (IDE[2]), vyvinuté spoločnosťou JetBrains[3], určené najmä pre vývoj Java aplikácií. Hlavnými prvkami sú: podrobná kódová asistencia, rýchla navigácia, inteligentná analýza chýb či refaktorizácia. Okrem toho obsahuje vstavané nástroje pre verzovanie, výstavbu projektu (Maven, Gradle a pod.), terminál, správu databázy a ďalšie[4].

5.2 Štruktúra aplikácie

Základná kostra aplikácie bola vytvorená v IntelliJ IDEA pomocou sprievodcu, kde bolo nutné vyplniť obecné údaje o projekte a vybrať potrebné závislosti. Ide o spomínané štartovacie závislosti, z ktorých sa tu však vyskytujú len najčastejšie používané. Ostatné závislosti bolo treba manuálne doplniť, k inicializácií projektu však tento sprievodca postačoval.

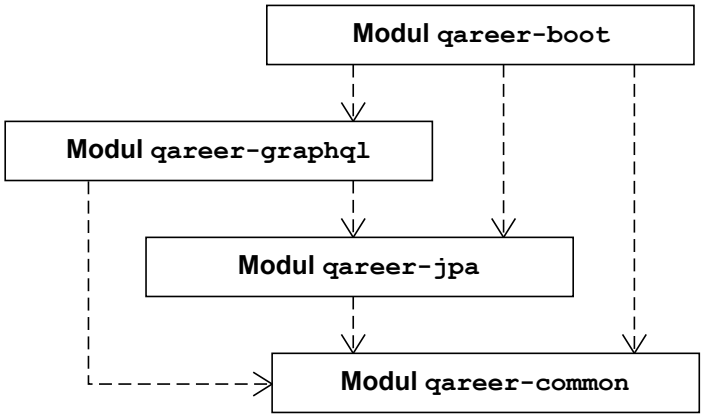
[2] *Integrated Development Environment*
[3] https://www.jetbrains.com/idea/
[4] https://www.jetbrains.com/idea/features/

Aplikácia je rozdelená do viacerých modulov pre zjednodušenie vývoja a udržovateľnosti. Pre výstavbu, správu a automatizáciu aplikácie je použitý nástroj Maven[5]. Vygenerovaný súbor `pom.xml` je určený ako rodičovský POM[6], ktorý spája a odkazuje jednotlivé submoduly [22], uvedené v elemente `modules`. Ďalej obsahuje hlavne závislosti (a ich verzie) spoločné pre všetky moduly. Ako rodič hlavného modulu je nastavený `spring-boot-starter-parent`, vďaka čomu možno využiť správu závislostí Mavenu pre dedičnosť verzií závislostí niekoľko bežne používaných knižníc, tj. nie je nutné explicitne špecifikovať tieto verzie [25].

V aplikácií sú teda štyri submoduly, ktoré ju tvoria a plnia jej funkcie:

- `qareer-boot` – hlavná aplikácia a konfigurácie
- `qareer-graphql` – služby, schéma, mapovanie, reprezentácia typov a funkcií
- `qareer-jpa` – perzistencia, entitné triedy a repozitáre
- `qareer-common` – spoločné konštanty, výnimky a pod.

Každému z týchto modulov je nastavený hlavný POM (`qareer`) ako rodič, od ktorého dedia závislosti (a ich verzie), ďalej sú tu závislosti špecifické pre daný submodul. Keďže niektoré moduly môžu používať iné pre splnenie potrebnej funkcie, vznikajú medzi nimi (interné) závislosti, definované v `pom.xml` podobne ako bežné (externé) závislosti. Priame závislosti medzi modulmi sú definované nasledovne: `qareer-boot` --→ `qareer-graphql` --→ `qareer-jpa` --→ `qareer-common`. Maven však automaticky dokáže doplniť tranzitívne závislosti (obr. 5.1), takže nie je nutné ich explicitne špecifikovať [22]. Moduly pozostávajú z balíčkov, ktorých obsah a účel bude rozobraný v ďalších kapitolách. V prílohe D sú uvedené balíčky modulov a závislosti, ktoré medzi nimi vznikajú.



Obr. 5.1: Submoduly rodičovského modulu a závislosti (aj tranzitívne) medzi nimi

5.3 Nastavenie technológie GraphQL a hlavné prvky aplikácie

Pre sprevádzkovanie technológie GraphQL na strane servera je použitá knižnica GraphQL Java[7], konkrétne štartovací balíček Spring Boot GraphQL Starter[8]. Po jeho pridaní do hlav-

[5] https://maven.apache.org/
[6] *Project Object Model* – XML reprezentácia projektu v súbore `pom.xml` [22]
[7] https://github.com/graphql-java/graphql-java
[8] https://github.com/graphql-java-kickstart/graphql-spring-boot

ného POM vo forme závislosti (`graphql-spring-boot-starter`), je sprístupnený koncový bod (`/graphql`), na ktorý je možné posielať GraphQL požiadavky typu POST[9]. V kombinácii s knižnicou GraphQL Java Tools[10] nie je nutné vytvárať GraphQL schému manuálne. Pridaním závislosti `graphql-java-tools` sú automaticky detekované súbory s príponou `.graphqls`, v ktorých je schéma definovaná.

Schéma

Pre vytvorenie GraphQL schémy sú použité typy, ktoré boli špecifikované v rámci návrhu (kapitola 4.4). Jednotlivé typy reprezentujú dáta, s ktorými aplikácia pracuje. Ďalej je však nutné rozšíriť schému o operácie, konkrétne dopyty a mutácie, ktoré sú uvedené v špeciálnych typoch `Query` a `Mutation`. Schéma je kvôli prehľadnosti rozdelená do viacerých `.graphqls` súborov. Keďže nie je možné uviesť v rámci schémy typy s rovnakým názvom[11], je potrebné definovať koreňové typy len jedenkrát, tj. zvyšné ich musia rozšíriť použitím kľúčového slova `extend`. Nezáleží, v ktorom zo súborov sú umiestnené koreňové typy – knižnica GraphQL Java Tools prechádza všetky nájdené `.graphqls` súbory a automaticky „pozbiera“ definované typy, z ktorých následne vytvorí schému[12].

Koreňové typy sú umiestnené v súbore `employees.graphqls` (prvý, ktorý bol vytvorený), v ostatných súboroch je použitý `extend` a konkrétne operácie (kód 5.1). Niektoré operácie vyžadujú viacero vstupných parametrov – namiesto všetkých potrebných parametrov sú zastúpené jedným vstupným (`input`) typom. Týmto spôsobom je skrátený zápis, predávanie parametrov je znovupoužiteľné, menej náchylné k chybám a parametre možno jednoduchšie organizovať [20].

```
type Employee {
    ...
}

type Query {
    ...
}

type Mutation {
    ...
}
```

```
type Branch {
    ...
}

extend type Query {
    ...
}

extend type Mutation {
    ...
}
```

Kód 5.1: Koreňové typy `Query` a `Mutation` uvedené v súbore `employee.graphqls` (vľavo), rozšírené o operácie špecifické pre pobočky v súbore `branches.graphqls` (vpravo)

Súbory `.graphqls` sú umiestnené v module `qareer-graphql`, konkrétne v priečinku `src/main/resources/graphql/`. V nasledujúcich kapitolách budú ešte spomenuté bližšie ukážky týkajúce sa ich obsahu.

[9] https://www.baeldung.com/spring-graphql
[10] https://github.com/graphql-java-kickstart/graphql-java-tools
[11] https://graphql.github.io/graphql-spec/June2018/#sec-Schema
[12] https://github.com/graphql-java-kickstart/graphql-java-tools/blob/master/src/main/kotlin/com/coxautodev/graphql/tools/SchemaParserBuilder.kt

Resolvery

Schéma definuje operácie, ktoré môžu klienti vykonávať a tiež vzťahy medzi jednotlivými typmi [20]. Čo však schéma nedefinuje, je spôsob, akým budú dáta získané. Na tento účel slúžia *resolvery*.

Koreňové typy `Query` a `Mutation` potrebujú špeciálne *beany*, definované v kontexte Springu pre správu rôznych polí koreňových typov[13]. Požiadavkou je, že musia implementovať rozhranie `GraphQLQueryResolver`[14] v prípade `Query` a `GraphQLMutationResolver`[15] v prípade `Mutation`, pričom každému poľu v koreňovom type zo schémy musí patriť metóda s rovnakým názvom v jednej z týchto tried. Metóda musí taktiež vracať zodpovedajúci typ – jednoduché typy (`String`, `Int`, `Long` atď.) sú systémom automaticky namapované na ekvivalentné Java typy. Každý komplexný typ je reprezentovaný Java *beanom*, rovnaká trieda však musí zastupovať rovnaký GraphQL typ, pričom na jej názve nezáleží.

Okrem vyššie uvedených *resolverov* existujú v rámci knižnice GraphQL Java Tools ešte rozhrania `GraphQLSubscriptionResolver`[16] a `GraphQLResolver`[17]. Prvý z nich sa používa pre *subscriptiony*, ktoré v aplikácii nie sú využité a druhý pre polia typov. Môže ísť o jednoduché aj zložité polia, ktoré vyžadujú vlastný spôsob získania dát. Pravidlá sú rovnaké ako pri spomínaných *resolveroch*, akurát metódy musia mať jeden parameter – *bean*, pre ktorý je *resolver* vytvorený.

Resolvery sú umiestnené v balíčku `resolver` (modul `qareer-graphql`). Pre väčšinu typov schéma poskytuje dopyty a mutácie, pre ktoré sú vytvorené *beany* s príponou `Resolver` (napr. `BranchQueryResolver`, `LanguageMutationResolver` a pod.), implementujúce potrebné rozhrania. Komplexné typy sú zastúpené POJO triedami s príponou `QLO`[18], lokalizované v balíčku `model`. Obsahujú len potrebné atribúty zodpovedajúce schéme, pre vygenerovanie *getterov* a *setterov* je použitá knižnica Lombok[19], ktorá automatizuje tvorbu často používaného kódu. Triedy sú anotované pomocou `@Data`[20], vďaka čomu sú automaticky doplnené aj metódy ako konštruktor či `toString` pre prevod objektu do reťazca.

Služby

Resolvery delegujú logiku operácií na služby. Balíček `service` obsahuje rozhrania, ktoré majú príponu `Service` a deklarujú možné operácie pre daný typ. Triedy s príponou `Impl` implementujú tieto rozhrania a nachádzajú sa v `service/impl/`. Sú anotované `@Service`[21] (špecializácia anotácie `@Component`[22]), čo indikuje, že trieda je automaticky detekovaná Springom a zároveň poskytuje podnikovú službu pre ostatné vrstvy aplikácie [4].

[13] https://www.baeldung.com/spring-graphql

[14] https://github.com/graphql-java-kickstart/graphql-java-tools/blob/master/src/main/kotlin/com/coxautodev/graphql/tools/GraphQLQueryResolver.java

[15] https://github.com/graphql-java-kickstart/graphql-java-tools/blob/master/src/main/kotlin/com/coxautodev/graphql/tools/GraphQLMutationResolver.java

[16] https://github.com/graphql-java-kickstart/graphql-java-tools/blob/master/src/main/kotlin/com/coxautodev/graphql/tools/GraphQLSubscriptionResolver.java

[17] https://github.com/graphql-java-kickstart/graphql-java-tools/blob/master/src/main/kotlin/com/coxautodev/graphql/tools/GraphQLResolver.java

[18] odvodené od *GraphQL Object*

[19] https://projectlombok.org/

[20] https://projectlombok.org/features/Data

[21] https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/stereotype/Service.html

[22] https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/stereotype/Component.html

Služby výhradne pracujú s triedami v balíčku `model`, keďže vstupnými parametrami a návratovými hodnotami dostupných metód sú práve inštancie týchto tried. Metódy sú jednoduché alebo zložité, záleží od konkrétneho prípadu použitia. Zložité metódy pozostávajúce z viacerých databázových operácií sú anotované `@Transactional`[23]. Týmto spôsobom nadobúdajú vlastnosti transakcie, z ktorých najdôležitejšia je atomicita – určuje, či sa operácie vykonajú ako celok, alebo sa nevykonajú vôbec [12]. Operácie môžu byť súčasťou iných služieb alebo nízkoúrovňových repozitárov.

Perzistencia

Perzistencia dát je v aplikácii zabezpečená prostredníctvom *Java Persistence API* (JPA), ktoré umožňuje prístup, skladovanie a spravovanie dát medzi Java objektmi a relačnou databázou [24]. Rovnako ako tabuľky sú dôležitou súčasťou relačných databáz, tak aj entity, ktoré sa na ne mapujú, sú dôležitou súčasťou JPA [12]. Entita je Java reprezentáciou databázovej tabuľky a mapovanie medzi objektovým a relačným modelom sa nazýva ORM (*Object-Relational Mapping*). Bežné Java triedy môžu byť jednoducho transformované do entít pomocou anotácií – takmer akákoľvek trieda s bezparametrickým konštruktorom sa môže stať entitou.

Do aplikácie je JPA *framework* pridaný pomocou štartovacej závislosti `spring-boot-starter-data-jpa`. Triedy reprezentujúce entity sú uložené v balíčku `entity` (modul `qareer-jpa`). Každá trieda je anotovaná `@Entity`[24], čím je označená ako entita. Špecifikácia názvu tabuľky, na ktorú sa má namapovať, je uvedená v elemente `name` anotácie `@Table`. Atribút triedy zastupujúci primárny kľúč (identitu entity) je anotovaný `@Id`. Jednoduché atribúty (typu `String`, `Long` a pod.) sú anotované `@Basic`, niektoré sú doplnené o `@Column`, kde možno bližšie špecifikovať vlastnosti stĺpca (názov stĺpca, nulovateľnosť, názov tabuľky a pod.)

Medzi entitami existuje viacero typov vzťahov. Z pohľadu konkrétnej entity sú to: 1:1, 1:M, M:1 a M:N. Prvé tri sú jednoducho namapované prostredníctvom príslušných anotácií (podľa poradia): `@OneToOne`, `@OneToMany` a `@ManyToOne`. Zložitejším prípadom je vzťah M:N (`@ManyToMany`), kde je nutné vytvoriť tretiu tabuľku (tzv. väzobnú), keďže jedna entita nemôže ukladať množinu hodnôt cudzích kľúčov v rámci jedného riadku. Nie je nutné ju explicitne reprezentovať entitnou triedou, stačí použiť anotáciu `@JoinTable`. Všetky väzobné tabuľky v aplikácii však obsahujú parametre navyše, takže ich treba explicitne reprezentovať triedou, ktorá sa odkazuje na zvyšné dve tabuľky pomocou `@ManyToOne` a obsahuje potrebné parametre. Primárny kľúč väzobnej tabuľky je tiež zastúpený triedou s anotáciou `@Embeddable`, pretože ide o zložený kľúč pozostávajúci z dvoch cudzích kľúčov. Tieto triedy sa nachádzajú v `entity/id/` a majú príponu `Id`. Triedy, ktoré ich používajú, majú príslušný atribút označený `@EmbeddedId` namiesto anotácie `@Id`.

Repozitáre poskytujú abstrakciu pre interakciu s dátovými úložiskami [24]. Tradične obsahujú rozhranie, ktoré poskytuje niekoľko vyhľadávacích metód ako `findById`, `findAll` pre získanie dát a metódy pre perzistenciu a vymazávanie dát. Repozitáre tiež zahŕňajú triedu, ktorá implementuje toto rozhranie použitím technológií závislých na dátových úložiskách (napr. pre databázu je to JDBC alebo JPA, pre LDAP sa používa JNDI). Tento prístup bol populárny, avšak je nutné písať veľké množstvo kódu, ktorý sa opakuje v každej

[23] https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/transaction/annotation/Transactional.html

[24] bližšie informácie ohľadom anotácií tohto typu možno nájsť na adrese https://docs.oracle.com/javaee/7/api/index.html

implementácií repozitára. Spring Data sa zameriava na odstránení tohto problému tým, že automaticky generuje implementáciu v čase behu. Jedinou požiadavkou je, že repozitárové rozhrania musia dediť od jedného z možných rozhraní, ktoré ponúka Spring Data – je ich viacero, záleží od požadovaných metód.

Rozhrania reprezentujúce repozitáre sú uložené v balíčku `repository`. Každé z nich má príponu `Repository` a dedí rozhranie `JpaRepository`, ktoré je parametrizované dvoma parametrami: doménovým typom, s ktorým má repozitár pracovať (entitná trieda) a typom jeho ID [25]. Týmto spôsobom získavajú repozitáre 18 metód pre vykonávanie bežných perzistentných operácií. Pre prispôsobenie operácií konkrétnym požiadavkám je použitý mechanizmus, ktorý dokáže na základe názvu metódy a jej parametrov vygenerovať príslušný SQL dopyt[25]. Názov, resp. výraz môže obsahovať napr. logické spojky (`And`, `Or`) či SQL výrazy ako `OrderBy` alebo `Distinct`. Repozitáre sú anotované `@Repository`[26] pre ich detekciu v rámci Springu.

Ako vyplýva z kapitoly 5.1, pre databázovú časť je použitý systém MySQL. Aplikácia pracuje s lokálne spustenou MySQL databázou, ku ktorej je nutné nastaviť potrebné prihlasovacie údaje: používateľské meno, heslo a URL databázy. Tie sa nachádzajú v konfiguračnom súbore `src/main/resources/application.yml` (modul `qareer-boot`). Zadaný používateľ musí mať v rámci systému MySQL pridelené oprávnenia pre manipuláciu s databázou (tvorba tabuliek, obmedzení, vkladanie údajov a pod.). Databáza na zadanej URL adrese musí existovať, byť prázdna a pre podporu slovenských znakov mať nastavené zotriedenie `utf8_slovak_ci`. Spring Boot dokáže automaticky vytvoriť databázu a naplniť ju potrebnými údajmi. Pre tento účel sú použité 2 SQL skripty: `schema.sql` a `data.sql`, lokalizované v rovnakom priečinku ako súbor `application.yml`. Súbor `schema.sql` obsahuje DDL[27] príkazy pre vytvorenie tabuliek a obmedzení, súbor `data.sql` zase obsahuje DML[28] príkazy pre vloženie ukážkových dát. Inicializáciu pri spustení aplikácie možno nastaviť parametrom `spring.datasource.initialization-mode` (hodnota `always` alebo `never`)[29].

Mapovanie

Služby definujú oproti repozitárom obecnejšie metódy, v ktorých však používajú operácie repozitárov. Keďže služby pracujú výhradne s triedami v balíčku `model` a repozitáre s entitnými triedami v balíčku `entity`, je potrebné vytvoriť konverziu medzi ich atribútmi. Manuálny spôsob prevodu pomocou *getterov* a *setterov* je zdĺhavý a vzniká pri ňom opakujúci sa kód, ktorý je ťažké udržiavať. Z tohto dôvodu vznikli mapovacie *frameworky* uľahčujúce prácu automatickým generovaním týchto metód. Jedným z nich je MapStruct, ktorý je použitý v aplikácii a umožňuje generovanie mapovacích tried na základe anotácií[30].

Mapovače sú rozhrania definované v balíčku `mapper`, anotované `@Mapper`. Tentokrát však nejde o Spring anotáciu, ale o anotáciu *frameworku* MapStruct, vďaka ktorej je vygenerovaná implementácia daného rozhrania v čase zostavenia programu (*build-time*)[31]. Pre detekciu mapovačov v rámci Springu ponúka MapStruct element `componentModel`,

[25] https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#repositories.query-methods.query-creation

[26] https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/stereotype/Repository.html

[27] *Data Definition Language*

[28] *Data Manipulation Language*

[29] https://docs.spring.io/spring-boot/docs/current/reference/html/howto-database-initialization.html#howto-initialize-a-database-using-spring-jdbc

[30] http://mapstruct.org/documentation/stable/reference/html/

[31] http://mapstruct.org/documentation/stable/reference/html/#defining-mapper

ktorý je nastavený na hodnotu `spring`, čím môžu byť mapovače v iných triedach použité pomocou `@Autowired`[32].

Mapovače deklarujú metódy a mapovanie zdrojového a cieľového typu je automaticky rozpoznané na základe návratového typu a typov parametrov danej metódy. V niektorých prípadoch je však potrebné explicitne špecifikovať zdrojové a cieľové typy pomocou anotácie `@Mapping`[33]. Mapované typy často odkazujú ďalšie typy, ktoré je tiež nutné namapovať. MapStruct automaticky vygeneruje potrebné metódy pre vnorené typy, čím môže v mapovačoch vznikať redundancia. Napr. pri mapovaní entitnej triedy `Branch` na triedu `BranchQLO` je pre mapovač `BranchMapper` vygenerovaná metóda mapovania odkazovanej triedy `Employee` na triedu `EmployeeQLO`. Tá sa už však nachádza v mapovači `EmployeeMapper`, čím vzniká spomínaná redundancia. Pre elimináciu tohto problému je v anotácii `@Mapper` definovaný element `uses` s hodnotou odkazovaného mapovača (`.class`)[34]. MapStruct tak automaticky nájde potrebnú metódu v odkazovanom mapovači pri generovaní implementácie danej metódy.

Validácia

Údaje, prichádzajúce od klienta by mali byť overené nielen v rámci klienskej časti webovej aplikácie (v prehliadači), ale aj na serveri. Aplikácia bežiaca vo webovom prehliadači je len jednou z možností, ako posielať dáta na server. Jednou z jej hlavných úloh je odľahčiť serveru časť overovania dát zadávaných používateľmi, nemožno sa však spoliehať, že používateľ využije práve klientskú aplikáciu pre posielanie požiadaviek.

Validácia posielaných dát prebieha na viacerých miestach serverovej časti aplikácie. Požiadavky sú najskôr overené voči definovanej GraphQL schéme, v ktorej sa overuje najmä nulovosť špecifikých polí, čo zabezpečuje knižnica GraphQL Java[35]. Po úspešnom overení nasleduje invokácia *resolverov*, na ktoré sú polia namapované [20]. Už v tejto časti (nie v službách) sú kontrolované ďalšie podmienky, ktoré musia zadávané hodnoty spĺňať, aby došlo k detekcii nesprávnych údajov čo najskôr. Kontrolujú sa najmä rozsahy hodnôt, veľkosti zoznamov, ale aj nulovosť.

Resolvery sú validované pomocou API pre validáciu *beanov* (JSR-349), ktoré definuje niekoľko obmedzení vo forme anotácií, ako napr. `@Min`, `@Max` a pod. (balíček `javax.validation.constraints`) [4]. Takto anotované sú hlavne `QLO` triedy (a ich atribúty) reprezentujúce vstupné typy, ktoré sú použité ako parametre jednotlivých *resolverov*. Okrem ponúkaných anotácií pre validáciu sú vytvorené aj vlastné, ktoré napr. overujú, či zadaný dátum nie je víkend. Vlastné anotačné typy a triedy, ktoré implementujú ich validačnú logiku sa nachádzajú v balíčku `validation` (modul `qareer-common`).

Dáta sú ďalej overované na úrovni entít, a to pomocou elementov anotácie `@Column`, ktoré kontrolujú nulovosť či dĺžku (v prípade reťazcov). Poslednou etapou validácie sú databázové obmedzenia, ktoré kontrolujú hodnoty ešte pred uložením do databázy.

Autentizácia a autorizácia

Druhá iterácia pridáva správu používateľov a ich rolí, s ktorými sú spojené pojmy autentizácia a autorizácia. Autentizácia je proces overenia totožnosti používateľa, typicky po-

[32] http://mapstruct.org/documentation/stable/reference/html/#configuration-options
[33] http://mapstruct.org/documentation/stable/reference/html/#basic-mappings
[34] http://mapstruct.org/documentation/stable/reference/html/#invoking-other-mappers
[35] https://github.com/graphql-java/graphql-java/tree/master/src/main/java/graphql/validation

skytnutím osobných údajov (napr. prihlasovacie meno a heslo) [21]. Autorizácia je proces overenia, či má používateľ právo vykonávať určitú aktivitu.

Pre zabezpečenie serverovej časti aplikácie na rôznych vrstvách je použitý flexibilný a prispôsobiteľný *framework* Spring Security. Do aplikácie je pridaný vo forme závislosti Spring Security Starter. Modul `qareer-graphql` je rozšírený o balíček `security`, v ktorom sa nachádzajú najmä triedy použité pri konfigurácií zabezpečenia.

Spring Security používa rozhranie `UserDetailsService`, ktoré obsahuje metódu `loadUserByUsername` pre vyhľadanie detailov používateľa (rozhranie `UserDetails`) podľa prihlasovacieho mena. `UserDetails` reprezentuje autentizovaného používateľa a Spring Security preň poskytuje vlastnú implementáciu (`org.springframework.security.core.user.details.User`). V službe, ktorá implementuje rozhranie `UserDetailsService` je prepísaná metóda `loadUserByUsername` tak, aby bol používateľ načítaný z databázy vo forme entitnej triedy `User`. Tá je už spolu s rolami namapovaná na požadovanú triedu.

Pre autentizáciu je použitý JWT (*JSON Web Token*), čo je metóda podľa štandardu RFC 7519[36] pre prenos bezpečnostných parametrov (*claims*) medzi dvoma stranami. Tvorbu *tokenov* vo forme reťazcov zabezpečuje trieda `TokenProvider` (balíček `security`) s použitím knižnice JJWT[37]. Tie sú vytvorené na základe rozhrania `Authentication` vytvoreného pri prihlasovaní používateľa.

Schéma je rozšírená o vstupné typy pre uskutočnenie prihlásenia. Údaje používateľa sú reprezentované typom `Credentials` (kód 5.2), ktorý obsahuje login a heslo. Samotné prihlásenie realizuje mutácia `authorize`, ktorej parameter je typu `Credentials`. Návratovým typom je `LoginPayload`, pozostávajúci z vygenerovaného *tokenu* a identifikovaného používateľa. Na základe zadaných údajov je najskôr vytvorený objekt triedy `UsernamePasswordAuthenticationToken`, ktorý je použitý ako parameter pre autentizáciu pomocou `AuthenticationManager`. Vytvorený objekt sa nastaví do bezpečnostného kontextu aplikácie [27] a tiež sa použije pre vytvorenie JWT *tokenu*. Podľa zadaného loginu sú v databáze vyhľadané informácie o používateľovi, ktoré sú spolu s vygenerovaným *tokenom* zaslané žiadateľovi.

```
input Credentials {
    login: String!
    password: String!
}
```

```
type LoginPayload {
    token: String
    user: User
}
```

Kód 5.2: Vstupný typ pre autentizačné údaje (vľavo) a typ pre *token* a identifikovaného používateľa (vpravo)

Autorizácia je zabezpečená triedou `JWTFilter`, konkrétne jej metódou `doFilter`, kde sú analyzované hlavičky požiadaviek, v ktorých je hľadaný *token* vo forme reťazca. Ten je najskôr validovaný metódou `validateToken` triedy `TokenProvider`. Ak kontrola prebehla v poriadku, je zavolaná metóda `getAuthentication`, ktorá na základe platného *tokenu* zistí práva používateľa a nastaví ho do bezpečnostného kontextu aplikácie. S autorizáciou súvisia aj práva pre volanie určitých metód. Spring Security poskytuje bezpečnosť na úrovni metód použitím rôznych anotácií [21]. Anotované sú hlavne metódy na úrovni služieb, napr. pomocou `@PreAuthorize`, ktorá zabezpečuje, že sú splnené určité podmienky ešte pred spustením anotovanej metódy [27]. V rámci tejto anotácie možno zadať bezpečnostný výraz pomocou jazyka SpEL[38] [21], kde sa dajú kombinovať metódy ako `hasRole` spolu s logickým

[36] https://tools.ietf.org/html/rfc7519
[37] https://github.com/jwtk/jjwt
[38] *Spring Expression Language*

výrazmi `AND`, `OR` a pod. V aplikácii sú často volané aj vlastné metódy v rámci týchto výrazov, napr. `isPlanSupervisor(#id)` s využitím `id` – parametru anotovanej metódy.

Vyššie uvedené triedy sú registrované v rámci konfiguračnej triedy `SecurityConfig`, ktorá prispôsobuje nastavenie bezpečnosti úpravou metód `configure` [25]. Tu sú hlavne nastavené koncové body, na ktorých server „počúva“, prípadne je možné použiť filtrovanie pridaním špecifických rolí. Aplikácia pracuje len s jedným koncovým bodom, a to `/graphql`, kde sú prijímané požiadavky bez obmedzenia (`permitAll`), keďže napr. metóda pre prihlásenie by mala byť prístupná nehľadiac na rolu používateľa.

5.4 Vybrané funkcie aplikácie

V predchádzajúcej kapitole bolo spomenuté, ako jednotlivé prvky aplikácie pracujú a komunikujú medzi sebou. Táto kapitola sa zameriava skôr na logiku vybraných funkcií, ktoré aplikácia poskytuje, sú tu uvedené najzaujímavejšie funkcie z pohľadu zložitosti, tj. nie obyčajné CRUD operácie, ktoré sú vykonávané napr. pri pozíciách.

Správa hodnotení

Správa hodnotení zahŕňa generovanie, úpravu a vymazanie (aj hromadné). Pre generovanie hodnotení boli určené 4 možnosti:

- **jednoduché** – ohodnotenie je vytvorené pre zadaného hodnotiteľa a hodnoteného
- **viacnásobné** – každému hodnotiteľovi sú priradení všetci zadaní hodnotení
- **členovia tímu** – ohodnotenie je vytvorené pre všetkých vybraných vedúcich a ich členov
- **vedúci tímu** – členovia tímu vybraných vedúcich budú hodnotiteľmi a ich vedúci budú hodnotenými

Pre všetky vyššie uvedené prípady je v GraphQL schéme definovaný spoločný vstupný typ obsahujúci polia `assessors` a `assessees` (kód 5.3). Oproti typu `Assessment` sa tu namiesto poľa `assessmentGoals` nachádza väzba na vstupné typy cieľov (`goals`) – pri generovaní sú zadávané ciele, väzby na ne sú vytvorené systémom, aby nedošlo k zadaniu hodnotenia, ktoré sa netýka vytváraného. Vstupný typ pre cieľ potom obsahuje len základné polia bez väzieb.

```
input AssessmentInput {
    name: String!
    description: String!
    endDate: Date
    assessors: [ID!]!
    assessees: [ID!]!
    goals: [GoalInput!]!
}
```

```
input GoalInput {
    name: String!
    description: String
    percentage: Int!
}
```

Kód 5.3: Vstupné typy pre hodnotenie (vľavo) a cieľ (vpravo) a väzba medzi nimi

Generovanie hodnotení má za úlohu mutácia s názvom `generateAssessments`, ktorej je predávaný parameter typu `AssessmentInput`. Logika je implementovaná v metóde príslušnej služby. Na základe počtu položiek polí `assessors` a `assessees` sa overuje 5 prípadov, ktoré môžu nastať:

- ak obidve obsahujú práve 1 položku, ide o prípad „jednoduché“
- ak obidve obsahujú 1 a viac položiek, ide o prípad „viacnásobné“
- ak `assessors` má 1 a viac položiek, `assessees` má 0, ide o prípad „členovia tímu“
- ak `assessees` má 1 a viac položiek, `assessors` má 0, ide o prípad „vedúci tímu“
- ak obidve majú 0 položiek, nastane chyba

V prípadoch „členovia tímu“ a „vedúci tímu“ sa ešte pred vygenerovaním overuje, či zadaní hodnotitelia, resp. hodnotení sú vedúcimi tímu (existuje aspoň jeden zamestnanec, ktorého vedúci je overovaný zamestnanec). Pre tieto prípady sú vyhľadaní podriadení zadaných vedúcich a sú v hodnotení nastavení ako hodnotení, resp. hodnotitelia. Po vytvorení potrebných hodnotení sú pre každý cieľ a hodnotenie vytvorené záznamy vo väzobnej tabuľke spájajúcej hodnotenia a ciele.

Pre úpravu hodnotenia definuje schéma vstupný typ `AssessmentUpdateInput` (kód 5.4), obsahujúci opis hodnotenia a pole `assessmentGoals` pre vstupné väzobné typy, ktoré oproti klasickému typu neobsahujú pole pre hodnotenie (doplní systém).

```
input AssessmentUpdateInput {
    description: String!
    assessmentGoals: [
        AssessmentGoalInput!]!
}
```

```
input AssessmentGoalInput {
    percentage: Int!
    description: String
    goal: ID!
}
```

Kód 5.4: Vstupné typy pre úpravu hodnotenia (vľavo) a pre väzbu medzi hodnotením a cieľom (vpravo)

Keďže je možné zadať ID cieľov, ktoré majú byť upravené, je potrebné overiť, či skutočne patria upravovanému hodnoteniu. Najskôr sú z databázy načítané ciele podľa ID hodnotenia, následne je každé zadané ID cieľa porovnané s načítanými. V prípade, že vyhľadané ID neobsahujú niektoré zo zadaných ID, nastane chyba. V opačnom prípade sú uložené všetky zadané hodnoty, spolu so samotným hodnotením. V rámci vstupu je tiež možné zadať menší počet `assessmentGoals`, ako je v databáze uložených. V tomto prípade nenastane chyba, ale uložia sa len zadané hodnoty.

Samostatné a hromadné vymazanie cieľov funguje na podobnom princípe. V aplikácii by ciele nemali vystupovať samostatne, pretože ich (napr. oproti jazykom) nie je možné priamo spravovať. Je teda nutné zabezpečiť, aby po vymazaní hodnotení boli odstránené nielen väzby medzi hodnoteniami a cieľmi, ale aj ciele samotné – to však len vtedy, ak už ciele nie sú viac odkazované (vo väzobnej tabuľke).

CRUD plánov a generovanie stretnutí

Generovanie stretnutí prebieha v rámci tvorby plánu, kedy je možné zadať potrebné údaje ohľadom rozsahu generovania a cieľov, ktoré majú byť súčasťou každého vytvoreného stretnutia. Pre tvorbu plánu je teda v schéme definovaný vstupný typ `PlanCreateInput` (kód

5.5), ktorý oproti typu pre plán neobsahuje ID a väzbu na stretnutia, pre ciele využíva rovnaký vstupný typ ako vyššie uvedené hodnotenia. Polia týkajúce sa generovania stretnutí nie sú v schéme kontrolované na nulovosť, pretože záleží na poli `recurring`, podľa ktorého návratovej hodnoty budú dané polia buď ignorované alebo využité pre generovanie. Validácia však prebieha v *resolveri* pre plán, a to pomocou vlastnej anotácie, ktorá podľa nastavenej hodnoty `recurring` overuje, resp. neoveruje polia na nulovosť.

```
input PlanCreateInput {
    name: String!
    description: String!
    recurring: Boolean!
    recurrence: Recurrence
    repeatEvery: Int
    startDate: Date
    endDate: Date
    startsAt: String
    duration: String
    meetingRoom: String
    employee: ID!
    supervisor: ID!
    goals: [GoalInput!]!
}
```

Kód 5.5: Vstupný typ pre tvorbu plánu

Pri tvorbe plánu sa overuje, či zadaný vedúci je naozaj vedúcim, následne je plán uložený do databázy, kvôli získaniu vygenerovaného ID novovytvoreného plánu. Stretnutia sa generujú na základe troch možných hodnôt, ktoré môže nadobúdať pole `recurrence`: denne, týždenne alebo mesačne. Najjednoduchším z hľadiska kontroly dátumu je týždenné generovanie, keďže ako počiatočný dátum (`startDate`) nemožno zvoliť víkend. K posuvu dní v týždni dochádza pri dennom a mesačnom opakovaní. V prípade, že v iterácii generovania stretnutí vyjde plánovaný deň na víkend, nie je preň vytvorené stretnutie. Pri mesačnom opakovaní je v takomto prípade nastavený najbližší deň po víkende, tj. pondelok. Údaje stretnutia sú spolu s vygenerovaným ID plánu uložené do databázy. Generovanie končí, akonáhle je dosiahnutý konečný dátum (`endDate`). Nakoniec sú každé zadané ciele s ID plánu uložené do databázy a pre ne a vygenerované stretnutia sú vytvorené záznamy do väzobných tabuliek medzi nimi, ktoré reprezentujú históriu cieľov v rámci plánu (`GoalHistory`).

Pre úpravu plánov špecifikuje schéma vstupný typ `PlanUpdateInput` (kód 5.6), ktorý obsahuje len pole pre väzbu na vstupné typy stretnutia (ID je zadávané v rámci mutácie pre úpravu plánu). Vstupný typ `MeetingInput` pre stretnutie obsahuje všetky polia ako klasický typ `Meeting`, okrem väzby na plán, ktorá je napevno zadaná pri úprave plánu. Podobne vstupný typ pre históriu cieľov neobsahuje pole pre stretnutie.

```
input PlanUpdateInput {
    meetings: [MeetingInput
        !]!
}

input MeetingInput {
    id: ID
    description: String
    date: Date!
    startsAt: String
    duration: String
    meetingRoom: String
    goalHistories: [
        GoalHistoryInput!]!
}

input GoalHistoryInput {
    percentage: Int!
    goal: ID!
}
```

Kód 5.6: Vstupné typy potrebné pre úpravu plánu

V úprave plánu sú iterované zadané stretnutia, ktoré môžu obsahovať ID v prípade existujúcich (úprava), alebo hodnotu `null` (vytvorenie) v prípade neexistujúcich stretnutí. Existujúce ID v rámci upravovaného plánu sú zistené z databázy a overované so zadávanými, takže pri nesprávnom ID stretnutia nastane chyba. Rovnako sú validované aj ciele v rámci typov pre histórie cieľov – ciele nie je možné pri úprave plánu meniť, musia byť použité tie isté, aké boli naviazané na plán pri jeho tvorbe. Ak validácia prebehla úspešne, sú všetky stretnutia vytvorené, resp. upravené a takisto aj záznamy vo väzobných tabuľkách pre histórie cieľov.

Vymazávanie plánov je realizované podobne ako pri hodnoteniach – najskôr sú vymazané záznamy pre histórie cieľov podľa ID stretnutí daného plánu, následne sú vymazané stretnutia samotné spolu so súvisiacimi cieľmi, no a nakoniec je vymazaný požadovaný plán.

Výpočet skóre zamestnancov

Skóre zamestnanca bolo definované v rámci návrhu GraphQL schémy (kapitola 4.4) ako pole vracajúce celočíslenú hodnotu. Skóre je vyhodnocované na základe konkrétnych oblastí osobného rozvoja a ich úrovne pre daného zamestnanca. Z tohto dôvodu sú pôvodné typy pre oblasti rozšírené o pole `score`, čím možno nastaviť celočíselnú hodnotu skóre pre konkrétnu zručnosť, technológiu či jazyk. Pre CRUD operácie s oblasťami sú v schéme definované vstupné typy, napr. `LanguageInput` pre jazyk, ktorý obsahuje názov jazyka a skóre (kód 5.7).

```
input LanguageInput {
    name: String!
    score: Int!
}
```

Kód 5.7: Vstupný typ pre jazyk

Výpočet, resp. prepočet skóre prebieha v rámci rôznych mutácií. Pri tvorbe a úprave zamestnanca sú prechádzané zadané jazyky, zručnosti a technológie a podľa ich úrovne je vypočítané výsledné skóre. Zadaná hodnota skóre vo vstupnom type je teda len základ, od ktorého sa odvíja konečná hodnota. Počet úrovní oblastí reprezentovaných vymenovaným typom môže byť rôzny. Jednotlivým úrovniam je pridelený index 0 až $n - 1$, kde n je počet úrovní vo vymenovanom type. K základu je pripočítaný základ vynásobený indexom a výsledná hodnota je pripočítaná k celkovému skóre zamestnanca. Výpočet v rámci iterácie

teda možno zhrnúť do nasledujúceho vzorca:

$$S_Z = S_Z + S_O + S_O * I_O \tag{5.1}$$
$$S_Z = S_Z + S_O * (1 + I_O), \tag{5.2}$$

kde S_Z predstavuje aktuálne celkové skóre zamestnanca Z v rámci iterácie, S_O je skóre konkrétnej oblasti O a I_O je index úrovne oblasti O. Prepočet skóre ďalej prebieha v mutáciách pre úpravu a vymazanie danej oblasti. Tu je opäť potrebné prepočítať skóre pre zamestnancov, ktorí majú nastavenú konkrétnu oblasť. Najskôr je zistené pôvodné skóre zamestnanca, následne je na základe novopridelenej hodnoty skóre a úrovne, ktorú má zamestnanec aktuálne nastavenú, vypočítaná hodnota výsledného skóre podľa nasledujúceho vzorca:

$$S'_Z = S_Z - S_O * (1 + I_O) + S'_O * (1 + I_O) \tag{5.3}$$
$$S'_Z = S_Z + (1 + I_O) * (S'_O - S_O), \tag{5.4}$$

kde S'_Z je nové skóre zamestnanca Z, S_Z je pôvodné skóre zamestnanca Z, S'_O predstavuje nové skóre oblasti O, S_O je pôvodné skóre oblasti O a I_O reprezentuje index úrovne oblasti O. Pri vymazávaní oblasti je nastavená hodnota $S'_O = 0$, čím dôjde k odpočítaniu pôvodného skóre oblasti od skóre zamestnanca.

Kapitola 6

Implementácia klientskej časti aplikácie

Táto kapitola sa podobne ako kapitola 5 venuje implementácii, avšak so zameraním na klientskú časť aplikácie. Vychádza z analýzy a návrhu (4) aplikácie, ale aj z predchádzajúcej kapitoly, keďže niektoré časti súvisia s funkčnosťou na serveri. Kapitola 6.1 sa zaoberá technológiami použitými pri vývoji aplikácie, nasleduje 6.2, kde sú opísané dôležité priečinky a súbory, z ktorých aplikácia pozostáva. Nastavenie technológie GraphQL spolu s hlavnými prvkami, ktoré aplikácia poskytuje, sú špecifikované v kapitole 6.3. Posledná kapitola (6.4) sa venuje implementácii vybraných funkcií aplikácie. Ukážky z výslednej aplikácie možno nájsť v prílohe E.

6.1 Použité technológie

Pre implementáciu klientskej časti aplikácie bol zvolený jazyk TypeScript v spolupráci s *frameworkom* Angular verzie 6, knižnica Angular Material a pre samotný vývoj prostredie WebStorm. Nižšie sú stručne opísané každé z týchto technológií, ostatné technológie sú v rámci použitia opísané v ďalších kapitolách.

Angular

Framework Angular prináša nástroje a možnosti, ktoré boli dostupné len pre vývoj na strane servera, webovým klientom, čím uľahčuje vývoj, testovanie a udržovateľnosť bohatých a komplexných webových aplikácií [10]. Angular aplikácie vyjadrujú funkčnosť prostredníctvom vlastných elementov a komplexné aplikácie môžu produkovať HTML dokument obsahujúci zmes štandardných a vlastných značiek.

Webové aplikácie možno obecne rozdeliť na dva typy: *round-trip* a *single-page*. Dlhú dobu boli webové aplikácie vyvíjané podľa *round-trip* modelu. Prehliadač vyžiadal počiatočný HTML dokument zo servera a ďalšie používateľské interakcie, ako napr. kliknutie na odkaz alebo potvrdenie formulára viedli k vyžiadaniu celého nového HTML dokumentu. Väčšina rovnakého obsahu však bola zahrnutá v každej odpovedi zo servera.

Single-page aplikácie pracujú na princípe odoslania počiatočného HTML dokumentu do prehliadača, avšak interakcie sú riešené prostredníctvom AJAX[1] požiadaviek. Tie umožňujú vyžiadanie len malých častí HTML alebo dát, ktoré sú následne vložené do existujúcich

[1] *Asynchronous JavaScript and XML*

elementov a zobrazené používateľovi. Týmto spôsobom sa HTML dokument nikdy znovu nenačíta alebo nenahradí a používateľ tak môže pokračovať v interakcií bez prerušenia.

Angular vyniká v tvorbe *single-page* aplikácií nielen vďaka inicializačnému procesu, ale aj kvôli výhodám použitia MVC vzoru. V kapitole 4.5 boli časti tohto vzoru opísané, Angular však pracuje s trochu odlišnou terminológiou – pre radič používa komponent (*component*) a pre pohľad zase šablónu (*template*).

Existuje viacero verzií Angularu. Originálny AngularJS bol populárny, ale obsahoval zvláštne implementované prvky, čo spôsobovalo zložitejší vývoj webových aplikácií. Angular (verzie 2 a viac) je kompletným prepisom, ktorý je jednotnejší, jednoduchší na učenie a dá sa s ním ľahšie pracovať. Po vydaní verzie 2 bola väčšina Angular aplikácií ešte implementovaná v jazyku JavaScript, no s každou ďalšou verziou klesala podpora pre prácu s týmto jazykom. Pre novšie verzie sa začal používať jazyk TypeScript, ktorý je nadmnožinou jazyka JavaScript. Stavia teda na špecifikácii JavaScriptu, no obohacuje ho o prvky ako typové anotácie či modifikátory prístupu. Správne použitie TypeScriptu má za výsledok stručnejší, čitateľnejší a ľahšie udržovateľný kód.

Angular Material

Knižnica Angular Material[2] poskytuje veľké množstvo rôznych komponentov[3], ktoré implementujú typické interakčné vzory podľa špecifikácie Material Design. Ku každému komponentu je uvedený prehľad, API, ktoré poskytuje a ukážkové príklady jeho použitia. Komponenty sú prispôsobené pre prácu na rôznych typoch zariadení. Knižnica je plne testovaná naprieč rôznymi modernými prehliadačmi a optimalizovaná pre použitie s Angularom.

WebStorm

WebStorm[4] je vývojové prostredie od spoločnosti JetBrains, ktoré podobne ako IntelliJ IDEA (kapitola 5.1) poskytuje bohaté prvky. Zameriava sa však hlavne na podporu jazyka JavaScript (a jemu podobných jazykov), Node.js, HTML a CSS. Podporuje moderné webové *frameworky*, medzi ktoré patrí aj Angular.

6.2 Štruktúra aplikácie

Pre Angular existuje veľké množstvo rôznych šablón, ktoré možno použiť ako základ aplikácie. Jednou z nich je voľne dostupná AdminPro Lite[5], ktorá bola vybraná kvôli podobnosti s návrhnutým používateľským rozhraním (kapitola 4.7). Ponúka predpripravené časti ako hlavičku či menu a pre demonštráciu obsahuje ukážkové komponenty knižnice Angular Material. Ďalej je tu niekoľko obrázkov, ktoré možno použiť ako profilové fotografie používateľov. Základná responzivita je zabezpečená prostredníctvom CSS súborov, ktoré definujú štýl komponentov pri rôznych rozmeroch obrazoviek. Pre prispôsobenie šablóny štýlu navrhnutého používateľského rozhrania tak stačí vykonať minimum úprav.

Vývoj aplikácií v jazyku JavaScript je závislý na systéme balíčkov (závislostí). Väčšina z nich obsahuje len niekoľko riadkov kódu, ale komplexná hierarchia, ktorá medzi nimi vzniká, je náročná na manuálnu správu, preto sa používa správca balíčkov. V aplikácií je

[2] https://material.angular.io/
[3] https://material.angular.io/components/
[4] https://www.jetbrains.com/webstorm/
[5] https://www.wrappixel.com/templates/materialpro-angular-lite/

použitý NPM[6], ktorý pracuje s balíčkami definovanými v súbore `package.json`. Dôležitým balíčkom je `angular-cli` – vďaka nemu možno vytvoriť základnú kostru Angular aplikácie prostredníctvom príkazového riadku (príkaz `ng new`). Takto vytvorenú štruktúru používa aj šablóna AdminPro Lite.

Inštalované balíčky sú sťahované a ukladané do priečinka `node_modules/`. Samotná implementácia prebieha najmä v priečinku `src/`, ktorý obsahuje aplikačné súbory vrátane zdrojových súborov (`src/app/`) či statických súborov (`src/assets/`), ako napr. obrázkov.

Väčšina hlavných sekcií definovaných v kapitole 4.7 sa týka doménových typov špecifikovaných v rámci GraphQL schémy. Pre každý takýto typ je v `sections/` vytvorený priečinok s jeho názvom (`branches/`, `employees/` a pod.), ktorý obsahuje podpriečinky pre jednotlivé sekcie. Okrem toho zahŕňa súbory pre modul (`*.module.ts`) či smerovanie (`*.routing.ts`). Pre súbory je použitá konvencia názvov podľa odporúčaného vzoru `prvok.typ.ts`[7]. Podpriečinky pozostávajú hlavne zo súborov pre komponenty (`*.component.ts`) a im zodpovedajúce šablóny (`*.component.html`). V prípade ostatných hlavných sekcií (napr. *dashboard*) sú komponenty umiestnené priamo v ich priečinku. Zdieľané prvky, ktoré sú používané na rôznych miestach aplikácie, sa nachádzajú v priečinku `shared/`.

Angular aplikácie pracujú s modulmi, pričom existujú dva druhy: JavaScript modul, ktorý obsahuje funkcie používané pomocou kľúčového slova `import` a Angular modul, ktorý sa používa pre opis aplikácie alebo skupiny súvisiacich prvkov. Dôležitou časťou každej Angular aplikácie je koreňový modul definovaný v súbore `app.module.ts` (`src/app/`). V aplikácií je viacero modulov, ale koreňový modul je ako jediný použitý v hlavnom „spúšťacom“ súbore `main.ts` (`src/`), kde je identifikovaný ako počiatočný. Bola uvedená základná štruktúra aplikácie, ďalšie priečinky, resp. súbory budú spomenuté v kapitolách nižšie.

6.3 Nastavenie technológie GraphQL a hlavné prvky aplikácie

Pre sprevádzkovanie GraphQL v rámci klientskej časti aplikácie je použitý nástroj Apollo Client (súčasť platformy Apollo GraphQL), spomínaný na konci kapitoly 3.3. Jeho integrácia s Angularom je zabezpečená prostredníctvom inštalácie potrebných balíčkov[8].

Do koreňového modulu sú vložené 2 dôležité moduly: `ApolloModule` a `HttpLinkModule`, spolu s nimi 2 služby: `Apollo` a `HttpLink`. Aby boli tieto služby prístupné v aplikácii, sú vložené pomocou techniky vkladania závislostí do konštruktoru triedy `AppModule`. V konštruktore sú teda dostupné inštancie týchto služieb, pomocou ktorých možno vytvoriť pripojenie na server. `Apollo` poskytuje funkciu `create` pre vytvorenie sieťového spojenia so serverom, ktorej sú predané 2 parametre. Prvým je `link` – adresa (koncový bod), na ktorej „počúva“ server. Pre jeho vytvorenie je použitá funkcia `create` služby `HttpLink`, ktorej je predaná už konkrétna URL adresa. Ďalším parametrom je `cache`, kde je použitá inštancia triedy `InMemoryCache`, ako normalizované dátové úložisko podporujúce všetky Apollo Client prvky[9]. Týmto spôsobom je nastavené spojenie so serverom, kde je možné posielať potrebné požiadavky a prijímať odpovede. Ďalšie prvky, týkajúce sa technológie GraphQL na strane klienta, budú spomenuté v rámci nasledujúcich kapitol.

[6] *Node.js Package Manager* – https://www.npmjs.com/
[7] https://angular.io/guide/styleguide#general-naming-guidelines
[8] https://www.apollographql.com/docs/angular/basics/setup#without-schematics
[9] https://www.apollographql.com/docs/angular/basics/caching

Modely

Existujú dva typy modelov: *view model*, ktorý reprezentuje dáta predávané z komponentu do šablóny a doménový model, ktorý obsahuje dáta týkajúce sa konkrétnej domény. Každý doménový typ (zo schémy) je reprezentovaný súborom s príponou `.model.ts`, ktorý pozostáva z rozhraní. Jedným z hlavných princípov jazyka TypeScript je typová kontrola, ktorá sa zameriava na tvar hodnôt a rozhrania sú určené k pomenovaniu týchto typov[10]. Súbory obsahujú viacero rozhraní pre rôzne tvary, ktoré typ môže nadobúdať, napr. pre vstupné typy (`input`) uvedené v GraphQL schéme na serveri. Súčasťou môžu byť tiež rozhrania pre dopyty a mutácie, určené pre kontrolu ich návratových typov. Keďže modely sú používané v rôznych častiach aplikácie, sú uložené v priečinku `shared/models/`.

Komponenty

Viacero moderných aplikácií prijalo prístup založený na komponentoch pri vývoji aplikácií [26]. V Angulare je komponent zodpovedný za správu šablóny a poskytuje jej dáta a logiku, ktorú potrebuje [10]. Namiesto uchovávania logiky v jednom komponente, je aplikácia rozdelená do viacerých komponentov, čím sú jednotlivé prvky rozdelené do stavebných blokov, ktoré môžu byť použité na viacerých miestach aplikácie a samostatne testované.

Komponent je trieda, ktorej anotácia `@Component` príjima objekt obsahujúci viacero vlastností [26]. Vlastnosť `selector` deklaruje HTML názov elementu, čím možno pridať komponent použitím špecifikovanej značky. Vlastnosť `templateUrl` udáva odkaz na HTML šablónu a `styleUrls` zase obsahuje pole odkazov k (S)CSS súborom, špecifickým pre komponent.

Podobne ako každá aplikácia vyžaduje koreňový modul, je taktiež potrebný koreňový komponent, ktorý má názov `AppComponent` a je umiestnený v `src/app/app.component.ts` [26]. Ako bolo spomenuté, sekcie aplikácie sú reprezentované komponentmi. Tie sú väčšinou tvorené znovupoužiteľnými komponentmi knižnice Angular Material či vlastnými komponentmi, ktoré sa nachádzajú v priečinku `shared/components/`.

Šablóny

Šablóny sú definované použitím HTML elementov, ktoré sú obohatené o dátové väzby (*data bindings*) [10]. Komponenty definujú dáta pre pohľady (*view data*), spomínané pri modeloch ako *view model*, pre zjednodušenie šablón a ich interakcií s komponentom. Vďaka dátovým väzbám možno tieto dáta používať v rámci šablón.

Šablóny sú v aplikácií definované dvojakým spôsobom – buď ako súčasť anotácie `@Component` (vlastnosť `template`), kde je HTML kód priamo uvedený, alebo vo zvlášť HTML súbore (s uvedením odkazu vo vlastnosti `templateUrl`). Prvým spôsobom sú definované kratšie kódy, napr. dialógové okná v `shared/components/`. Druhým spôsobom je definovaná väčšina šablón, keďže pozostávajú z väčšej časti kódu.

Služby

Modely, okrem reprezentácie doménových dát, obsahujú typicky aj logiku (CRUD operácie), ktorú však v aplikácií zabezpečujú služby. Služby sú obecne objekty, ktoré abstrahujú spoločnú logiku, používanú na viacerých miestach [26]. Môžu obsahovať funkcie alebo aj statické hodnoty ako reťazce či čísla, a pri ich použití v rôznych častiach aplikácie tak možno

[10] https://www.typescriptlang.org/docs/handbook/interfaces.html

tieto dáta zdieľať. Angular označuje triedy reprezentujúce služby pomocou `@Injectable`, čím sú registrované v rámci jeho systému vkladania závislostí.

Aplikácia používa najmä Apollo služby pre prácu s GraphQL[11]. Ide o bežné Angular služby, ktoré obsahujú funkcie pre tvorbu dopytov či mutácií. Dopyty sú tvorené pomocou metódy `Apollo.watchQuery`, ktorá vracia špeciálny typ `Observable` obohatený o metódy (napr. `refetch`) pre manipuláciu s dopytom[12]. `Observable` je kľúčovým stavebným blokom knižnice Reactive Extensions (RxJS), ktorý reprezentuje „pozorovateľnú" sekvenciu udalostí [10]. Objekt sa môže prihlásiť k „pozorovaniu" `Observable` prostredníctvom funkcie `subscribe` a získavať tak upozornenia zakaždým, čo nastane udalosť. Funkcia `Apollo.query` vracia `Observable`, ktorá vyšle výsledok len raz. Oproti tomu `Apollo.watchQuery` dokáže vyslať niekoľko výsledkov, čo možno využiť napr. pri opätovnom zasielaní požiadaviek na server (pre obnovu *cache*). Väčšina funkcií definovaných v rámci služieb používa práve tieto funkcie. Je im predaný objekt ako parameter s vlastnosťami pre špecifikáciu konkrétneho GraphQL dopytu (`query`), jeho premenných (`variables`) a pod. GraphQL dopyty sú uvedené v reťazcoch, ich spracovanie do formy pre použitie s `Apollo.query` má za úlohu funkcia `gql` knižnice `graphql-tag`[13]. Takto spracované dopyty sú uložené v konštantách (súbory `*.constants.ts`) v priečinku `shared/constants/`.

Funkcie pre mutácie sú analogické k funkciám pre dopyty, používa sa však metóda `Apollo.mutate`. Služby používajú modely ako typové argumenty[14] funkcie `Apollo.mutate` či `Apollo.watchQuery`, kvôli kontrole typov. Taktiež samotné funkcie služieb, ktoré vracajú `Observable` s rozhraním modelu ako typovým argumentom. Všetky spomínané služby sa nachádzajú v priečinku `shared/services/`.

Smerovanie

Väčšina webových aplikácií obsahuje viacero sekcií, resp. stránok, medzi ktorými je potrebné sa určitým spôsobom navigovať. Smerovanie (*routing*) je výraz používaný pre opis schopnosti aplikácie meniť obsah stránky, ako sa po nej používateľ naviguje [26]. Klasickým prístupom pri *round-trip* aplikáciách je vyžiadanie stránky na určitej URL adrese od servera, ktorý následne pošle vygenerovanú stránku. Pri *single-page* aplikáciách je však smerovanie plne závislé na prehliadači, kde JavaScript manipuluje s aktuálnou URL pre zabezpečenie aktuálnej pozície v rámci aplikácie.

Konfigurácia pre navigáciu je konvenčne riešená v súbore `app.routing.ts`, definovanom v priečinku `src/app/` [10]. Pre smerovanie je do aplikácie importovaná knižnica `@angular/router`, ktorej trieda `Routes` je použitá pre definíciu kolekcií ciest. Ku každej ceste možno uviesť komponent, ktorý sa má zobraziť používateľovi. Keďže je aplikácia rozdelená do modulov, je použitý tzv. *lazy loading* a namiesto komponentu je ku každej ceste uvedený modul. Týmto spôsobom je načítaný modul, len ak je to potrebné, čo môže ušetriť na veľkosti súboru hlavných balíčkov (*core bundles*) [26]. Pri väčšine sekcií aplikácie je potrebné načítať tiež menu a hlavičku, ktoré sú definované ako zvlášť komponenty spojené do komponentu `FullComponent` (`app/layouts/full/`). Aby ich nebolo nutné načítať spolu s každým komponentom, s ktorým sa majú tiež zobraziť, je `FullComponent` definovaný ako rodičovský.

[11] https://www.apollographql.com/docs/angular/basics/services
[12] https://www.apollographql.com/docs/angular/basics/queries#queryref
[13] https://github.com/apollographql/graphql-tag
[14] https://www.typescriptlang.org/docs/handbook/generics.html

Ku každému modulu pre sekcie je vytvorený súbor s konfiguráciou smerovania, kde je nastavená implicitná cesta k hlavnej sekcii. Pre každú sekciu je potom tiež vytvorené smerovanie, kde už sú odkazované konkrétne komponenty.

V rámci smerovania je nastavená ochrana (*guard*) pre niektoré cesty, ktorá overuje určité podmienky a môže vykonať potrebné funkcie pri zmene URL adresy. Jedným z niekoľkých druhov ochrán je rozhranie `CanDeactivate`. Služba, ktorá toto rozhranie implementuje, zabezpečuje aby bol používateľ upozornený pri opustení aktuálnej stránky. Je definovaná hlavne pri formulároch, aby nedošlo k strate nepotvrdených informácií.

Podpora viacjazyčnosti

Pre podporu viacerých jazykov v aplikácií je použitá knižnica `ngx-translate`[15], ktorej modul `TranslateModule` je importovaný do koreňového modulu `AppModule`. V module je nakonfigurovaná funkcia pre lokalizáciu a načítanie (`loader`) súborov, kde sú uložené preklady. Tieto súbory sa nachádzajú v `src/assets/i18n/` a sú typu JSON. Každý je pomenovaný podľa skratky daného jazyka (`en.json` a `sk.json`), pričom ich štruktúra pozostáva z vnorených JSON objektov, ktorých kľúče sú rovnaké pre všetky súbory.

V konštruktore koreňového komponentu `AppComponent` je použitá služba `TranslateService`, kde je nastavené, aký jazyk sa má implicitne použiť. Tiež je použitá v rámci komponentu hlavičky (`app/layouts/full/header/`), v ktorom prebieha zmena jazyka prostredníctvom funkcie `changeLanguage`. Vybraný jazyk je najskôr uložený do lokálneho úložiska prehliadača (`localStorage`[16]) pomocou služby `LocalStorageService` knižnice `ngx-webstorage`. Následne je nastavený funkciou `use` služby `TranslateService`. V koreňovom komponente sa pri spustení aplikácie vyhľadá v lokálnom úložisku nastavený jazyk, v prípade jeho neexistencie sa použije implicitný.

Knižnica `ngx-translate` ponúka viacero spôsobov, ako nastaviť konkrétne hodnoty prekladov. V aplikácii sú najčastejšie použité tzv. „rúry“ (*pipes*) a direktívy (*directives*). V Angulare existuje niekoľko druhov direktív, pri prekladoch sa používajú atribútové direktívy – triedy schopné modifikovať správanie a vzhľad elementu, na ktorý sú aplikované [10]. Pri aplikácii na element sa teda zmení jeho textový obsah podľa hodnoty kľúča zadaného na vstup direktívy. „Rúry“ sa obecne používajú pre formátovanie dát [26]. Sú to triedy, ktoré transformujú dáta ešte predtým, ako ich obdrží direktíva alebo komponent [10].

Autentizácia a autorizácia

Pre autentizáciu je v aplikácii vytvorených niekoľko služieb. Služba `AccountService` sa používa pre správu účtu používateľa a zdieľanie informácií o ňom medzi viacerými komponentmi. Prihlasovanie a odhlasovanie zabezpečuje služba `LoginService` a pre prácu s JWT *tokenmi* je použitá služba `AuthServerProvider`.

Prihlásenie je realizované v komponente `LoginComponent`, ktorý pozostáva z 2 polí pre zadanie autentizačných údajov: prihlasovacieho mena a hesla. Po vyplnení a potvrdení formulára je zavolaná mutácia `authorize` (kód 6.1). V prípade úspešnej autentizácie používateľa je najskôr uložený získaný *token* do úložiska `sessionStorage`[17], čím je zapamätaný v rámci aktuálneho sedenia používateľa. Následne je objekt `User` uložený do premennej služby `AccountService`, ktorá umožňuje k nemu pristupovať z iných komponentov. Obsahuje informácie o pridelených roliach a zamestnancovi, na ktorého sa prihlásený používateľ

[15] https://github.com/ngx-translate/core
[16] https://www.w3.org/TR/webstorage/#dom-localstorage
[17] https://www.w3.org/TR/webstorage/#the-sessionstorage-attribute

viaže. Po prihlásení je presmerovaný na úvodnú stránku, tj. *dashboard*, v prípade, že bola zadaná iná adresa, je na ňu presmerovaný.

Autorizácia, v prípade klientskej časti aplikácie, sa týka najmä prístupu k jednotlivým sekciám. V predchádzajúcich kapitolách boli v rámci smerovania spomenuté tzv. ochrany. Pre zamedzenie, resp. povolenie prístupu k určitej ceste je použitá trieda s názvom `UserRouteAccessService` implementujúca rozhranie `CanActivate`. Súbory s príponou `*.routing.ts` definujúce cestu k danému komponentu sú v prípade obmedzenia prístupu rozšírené o vlastnosť `canActivate`, kde je trieda uvedená. Zároveň je tu pridaná vlastnosť `authorities` s poľom vymenovaných rolí, ktoré majú mať prístup k danej ceste. Funkcia `canActivate` triedy `UserRouteAccessService`, ktorá má prístup k poľu s uvedenými rolami, kontroluje, či prihlásený používateľ disponuje aspoň s jednou z vymenovaných rolí. V prípade, že áno, vracia hodnotu `true`, čím je používateľovi povolený prístup k danej adrese. Ak používateľ nemá právo prístupu, je presmerovaný na stránku so správou „prístup zamietnutý“.

```
mutation authorize($credentials: Credentials!) {
  authorize(credentials: $credentials) {
    token
    user {
      login
      roles
      employee {
        id
        firstName
        lastName
        photo
      }
    }
  }
}
```

Kód 6.1: Mutácia pre prihlásenie používateľa a získanie potrebných údajov

Aby mohol mať používateľ právo k vykonávaniu určitých dopytov alebo mutácií, je potrebné posielané požiadavky doplniť o *token* získaný pri autentizácii, ktorý by sa mal nachádzať v hlavičke posielanej žiadosti. Apollo Link poskytuje funkčnosť (*middleware*) pre modifikáciu každej požiadavky posielanej cez `link`[18], ktorá získava *token* uložený v rámci `sessionStorage` a ten nastavuje do HTTP hlavičky žiadosti. Funkcia definovaná v koreňovom module je spojená s `HttpLink`, čím je zabezpečená požadovaná funkčnosť.

Odhlásenie používateľa je možné vyvolať stlačením príslušného tlačidla v hlavičke aplikácie. V takom prípade je najskôr vyprázdnená *cache*, kde sú ukladané všetky získané dáta z dopytov a mutácií, pomocou funkcie `reset`. Následne je vyčistené úložisko pre sedenia používateľa, spolu s uloženými údajmi v rámci služby `AccountService`. Väčšina dopytov a mutácií vracia typ `Observable`, ku ktorému môžu byť „prihlásené“ určité komponenty. Po odhlásení používateľa by mali byť „odhlásené“, pretože ak dôjde k prihláseniu ďalšieho používateľa, ktorý nebude mať požadované práva k „prihláseným“ operáciám, nastane chyba

[18] https://www.apollographql.com/docs/angular/basics/network-layer#linkMiddleware

kvôli nedostatku oprávnení. K dopytom a mutáciám je preto pridaný operátor `takeUntil`[19] knižnice RxJS, ktorému je pridaný parameter typu `Subject`[20]. Pri zmene hodnoty tohto parametru dôjde k „odhláseniu“ danej `Observable`. Parameter je vo forme premennej nastavený v službe `LoginService`, pri odhlásení používateľa dôjde k zmene jeho hodnoty a tým sú „odhlásené“ všetky potrebné `Observable`.

6.4 Vybrané funkcie aplikácie

Táto kapitola sa podobne ako kapitola 5.4 venuje skôr logike vybraných funkcií aplikácie, riešeniam a problémom, ktoré nastali počas ich implementácie, no hlavne sú tu opísané dopyty či mutácie použité pre získanie alebo úpravu požadovaných dát.

Správa plánov

V kapitole 5.4 bol opísaný priebeh tvorby, úpravy a vymazania plánu z pohľadu serverovej časti aplikácie. Názov kapitoly „Správa plánu“ však neznamená súhrn všetkých týchto troch operácií, ale konkrétne ide o plánovanie stretnutí v rámci daného plánu. Rovnomenná podsekcia bola definovaná v rámci návrhu používateľského rozhrania (kapitola 4.7).

V podsekcii je najskôr na základe URL zistené ID plánu, ktoré je použité ako parameter dopytu `getPlan` (kód 6.2). Výberová množina dopytu sa skladá z viacerých častí, ktoré majú byť načítané. Medzi ne patria hlavne základné údaje o stretnutiach uvedené pri tvorbe plánu, meno zamestnanca pre zobrazenie v rámci nadpisu, či informácie o cieľoch, ktoré sú rovnaké pri každom stretnutí (hlavne názov a opis).

Šablóna pre správu plánu obsahuje stretnutia reprezentované kartami (komponent `mat-tab`[21]), kde každá má v názve dátum stretnutia, podľa ktorého sú usporiadané. V hlavičke každej karty sú okrem dátumov uchovávané aj ID stretnutí. Po kliknutí na danú kartu sú načítané ďalšie informácie ohľadom stretnutia pomocou dopytu `getMeeting` (kód 6.2), ktoré sú použité pre vyplnenie obsahu zvolenej karty. Týmto spôsobom sú stretnutia načítané na požiadanie, teda nie ako celok, čo zvyšuje rýchlosť načítania komponentu a používateľ vidí podrobnosti len o stretnutiach, o ktoré má záujem.

Správa plánu umožňuje pridať, upraviť aj vymazať stretnutia. Stretnutie sa pridáva tlačidlom v pravom hornom menu, po jeho kliknutí sa otvorí dialógové okno, čo je samostatný komponent obsahujúci jedno formulárové pole pre zadanie dátumu. Dátum možno zvoliť okrem víkendu, po potvrdení je podľa zadaného dátumu vyhľadaný index pre umiestnenie novej karty. Polia stretnutia sú po umiestnení karty naplnené hodnotami z predošlej karty, v prípade, že karta má byť umiestnená ako prvá, sú použité hodnoty z plánu. Novovytvoreným stretnutiam nie sú pridelené ID (aby mohli byť vytvorené), upravované majú svoje ID uvedené v rámci formuláru. Formuláru pridaného stretnutia je nastavený parameter `dirty`[22] na hodnotu `true`, ktorý je automaticky zmenený aj pri úprave formuláru už existujúceho stretnutia.

[19] https://www.learnrxjs.io/operators/filtering/takeuntil.html
[20] https://www.learnrxjs.io/subjects/
[21] https://material.angular.io/components/tabs/overview
[22] https://angular.io/guide/form-validation#why-check-dirty-and-touched

```
query getPlan($id: Long!) {
  getPlan(id: $id) {
    name
    startsAt
    duration
    meetingRoom
    employee {
      firstName
      lastName
    }
    goals {
      id
      name
      description
      percentage
    }
    meetings {
      id
      date
    }
  }
}
```

```
query getMeeting($id: Long!) {
  getMeeting(id: $id) {
    startsAt
    duration
    description
    meetingRoom
    goalHistories {
      percentage
      goal {
        id
      }
    }
  }
}
```

Kód 6.2: Dopyty pre získanie základných informácií použitých pri podsekcii „Správa plánu“

Vymazanie aktuálne vybranej karty (stretnutia) možno vykonať pomocou tlačidla v menu, kde je aj tlačidlo pre pridanie nového stretnutia. Stretnutie však nie je hneď fyzicky vymazané, akurát je odstránená karta a vyberie sa predchádzajúca. Vymazané stretnutia obsahujúce ID sa zaznamenávajú, kvôli neskoršiemu vymazaniu.

Potvrdením celého formulára pre správu plánu sú najskôr filtrované stretnutia reprezentované formulármi s parametrom `dirty` nastaveným na `true`, čím sú získané len upravené stretnutia. Tie sú pridané do plánu a spolu s jeho ID je zavolaná mutácia `updatePlan` pre úpravu plánu. ID zaznamenaných stretnutí, ktoré boli vymazané, sú tiež spolu s ID plánu použité v mutácii `deleteMeetings` pre fyzické odstránenie stretnutí.

Organizačná štruktúra spoločnosti

Organizačná štruktúra spoločnosti tvorí strom zamestnancov, zoradených hierarchicky podľa vedúcich. Pôvodne bolo zamýšľané vytvoriť dopyt pre načítanie „najvyššieho“ vedúceho a pomocou poľa `employees` pre podriadených vo výberovej množine pristúpiť k informáciám o podriadených zamestnancoch. Ako však kód 6.3 naznačuje, problémom je neznámy počet vnorení, keďže nie je známa výška stromu, ktorý vytvára hierarchickú štruktúru zamestnancov. Jedným z riešení by bolo vypočítať maximálnu hĺbku vnorenia na serveri a podľa zaslaného čísla dynamicky vytvoriť reťazec pre dopyt, ktorý by bol následne poslaný na server, čím by boli zaslaní zamestnanci už vo vhodnej štruktúre. Tento spôsob však nie je najideálnejším riešením problému.

```
query ($id: Long!) {
  getDirector(id: $id) {
    firstName
    lastName
    employees {
      firstName
      lastName
      employees {
        ???
      }
    }
  }
}
```

Kód 6.3: Hypotetický dopyt pre získanie celej štruktúry zamestnancov, kde však nie je známa hĺbka vnorenia

Pre organizačnú štruktúru je vybraná knižnica OrgChart[23], kvôli podpore pokročilých funkcií, ako napr. priblíženie, posúvanie či dynamické načítavanie uzlov pomocou AJAX[24]. Ďalším riešením vyššie uvedeného problému bolo načítať dáta postupne, tj. najskôr zistiť „najvyššieho“ vedúceho a následne, pomocou funkcie knižnice, dynamicky načítať ďalšie uzly (zamestnancov) podľa kliknutej šípky. Funkcia pre načítanie dát pomocou AJAX je však prispôsobená REST architektúram – ako dátové zdroje je nutné nastaviť konkrétne URL adresy, čo nevyhovuje aktuálnej aplikácii. Z tohto dôvodu bolo nutné prepísať funkcie knižnice, ktorá je písaná v jQuery, čo celkom skomplikovalo situáciu. Nakoniec bola požadovaná funkčnosť implementovaná a dáta bolo možné po kliknutí dynamicky načítať. Štruktúra sa však zobrazovala horizontálne, čo pôsobilo neprehľadne a zákazník požadoval vertikálne zobrazenie. Knižnica OrgChart má však problém práve s kombináciou dynamické načítanie a vertikálne zobrazenie[25] a pri kliknutí na šípku sa prestanú zobrazovať ďalšie uzly.

Konečným riešením, ktoré je aj aktuálne použité v aplikácii, je načítanie zamestnancov ako celku pomocou dopytu `getEmployees` (kód 6.4). Pole zamestnancov je preusporiadané do stromovej štruktúry podľa ID vedúceho. Zamestnanec bez vedúceho je považovaný za koreňového zamestnanca, ktorí sa v štruktúre zobrazí ako prvý. Môže však nastať prípad, že takýchto zamestnancov bude viac, čo je riešené výberovým menu, kde je možné zvoliť koreňového zamestnanca, podľa ktorého sa má štruktúra vykresliť. Táto ponuka nie je zobrazená v prípade, ak neexistuje žiadny zamestnanec, alebo ak existuje len jeden koreňový zamestnanec (vtedy je štruktúra priamo zobrazená). Štruktúra sa vykresľuje do tretej úrovne vnorenia, pri dvojkliku na uzol je otvorené dialógové okno, ktoré podľa predaného ID zamestnanca o ňom načíta bližšie informácie.

[23] https://github.com/dabeng/OrgChart
[24] https://rawgit.com/dabeng/OrgChart/master/demo/index.html
[25] https://github.com/dabeng/OrgChart/issues/458

```
query {
  getEmployees {
    id
    firstName
    lastName
    photo
    branch {
      id
      name
    }
    position {
      name
    }
    supervisor {
      id
    }
  }
}
```

Kód 6.4: Dopyt pre získanie všetkých zamestnancov

Dashboard

Dashboard, ktorý je súčasťou druhej iterácie, zobrazuje množstvo informácií o aktuálne prihlásenom používateľovi. Takmer všetky sú získané pomocou jedného dopytu – `getEmployee` (kód 6.5). Samostatným dopytom `getUpcomingAndLatestMeeting` (kód 6.6) sú získané informácie o nadchádzajúcom a poslednom stretnutí – tento dopyt vracia pole s dvomi hodnotami (v prípade nenájdeného stretnutia môžu nadobúdať nulové hodnoty). Parametrom oboch dopytov je ID zamestnanca získané podľa prihláseného používateľa.

Dashboard pozostáva z viacerých častí, ktoré sú tvorené kontajnermi (komponent `mat-card`[26]). Kontajner s názvom „Môj tím“ obsahuje zoznam zamestnancov, o ktorých je možné získať bližšie informácie kliknutím na konkrétneho zamestnanca. V takom prípade sa otvorí rovnaké dialógové okno ako pri organizačnej štruktúre spoločnosti. Kontajner pre osobný rozvoj pozostáva z jazykov, zručností a technológií, ktorými disponuje prihlásený používateľ. Tie sú reprezentované komponentmi `mat-chip`[27], ktoré sú farebne odlíšené podľa dosiahnutej úrovne v konkrétnej oblasti. Pri prejdení myšou na konkrétny `mat-chip` sa zobrazí úroveň vo forme komponentu `mat-tooltip`[28]. Ďalší kontajner zobrazuje súhrnné informácie o prihlásenom používateľovi, pre ktoré je použitých prvých osem polí vo výberovej množine dopytu `getEmployee`.

Kontajner s názvom „Štatistiky“ je určený pre prehľadné zobrazenie progresu v rámci plánov zamestnanca. Z informácií o plánoch sú pri načítaní `dashboardu` zistené len ich ID a názov. Na prepínanie medzi plánmi je použité menu pre výber, po zvolení konkrétneho plánu sú načítané zvyšné informácie potrebné pre vykreslenie grafu pomocou dopytu `getPlan` (kód 6.6).

[26] https://material.angular.io/components/card/overview
[27] https://material.angular.io/components/chips/overview
[28] https://material.angular.io/components/tooltip/overview

```
query getEmployee($id: Long!) {
  getEmployee(id: $id) {
    title
    firstName
    lastName
    photo
    score
    positionSeniority
    position {
      name
    }
    supervisor {
      firstName
      lastName
    }
    employees {
      id
      firstName
      lastName
      position {
        name
      }
    }
    skillEmployees {
      level
      skill {
        name
      }
    }
    languageEmployees {
      proficiency
      language {
        name
      }
    }
    technologyEmployees {
      seniority
      technology {
        name
      }
    }
    plans {
      id
      name
    }
  }
}
```

Kód 6.5: Dopyt pre získanie väčšiny informácií potrebných pre *Dashboard*

```
query getUpcomingAndLatestMeeting
    ($employeeId: Long!) {
  getUpcomingAndLatestMeeting(
      employeeId: $employeeId) {
    date
    startsAt
    description
    meetingRoom
  }
}
```

```
query ($id: Long!) {
  getPlan(id: $id) {
    meetings {
      date
      goalHistories {
        percentage
      }
    }
  }
}
```

Kód 6.6: Dopyty potrebné pre kontajnery „Stretnutia“ (vľavo) a „Štatistiky“ (vpravo)

Pre *burndown* graf je použitá knižnica `ng2-charts`[29], ktorá poskytuje množstvo rôznych grafov a funkcií pre ich prispôsobenie vlastným potrebám. Najskôr je zistený celkový počet dní medzi načítanými stretnutiami plánu, ktoré server posiela zoradené vzostupne podľa dátumu. Pre každé stretnutie je spočítaná priemerná hodnota percent všetkých cieľov, ktorá je odčítaná od hodnoty 100. Tým je získaný priemerný zostávajúci počet percent do dokončenia cieľov pre dané stretnutie. Každé stretnutie je reprezentované bodom v grafe

[29] https://valor-software.com/ng2-charts/

a body sú spojené čiarou, je teda použitý čiarový graf. Druhá čiara zobrazuje ideálny progres v rámci daného plánu. Jej body sú vypočítané tak, aby smerovala rovnomerne od 100 % k 0 %. Pri prejdení myšou na konkrétny bod je zobrazený detail vo forme bubliny, ktorá obsahuje dátum stretnutia, priemerný a ideálny počet percent pre daný dátum a rozdiel týchto percent.

Posledný kontajner ukazuje informácie o nadchádzajúcom a poslednom stretnutí, hlavne opis, dátum, čas a miesto stretnutia.

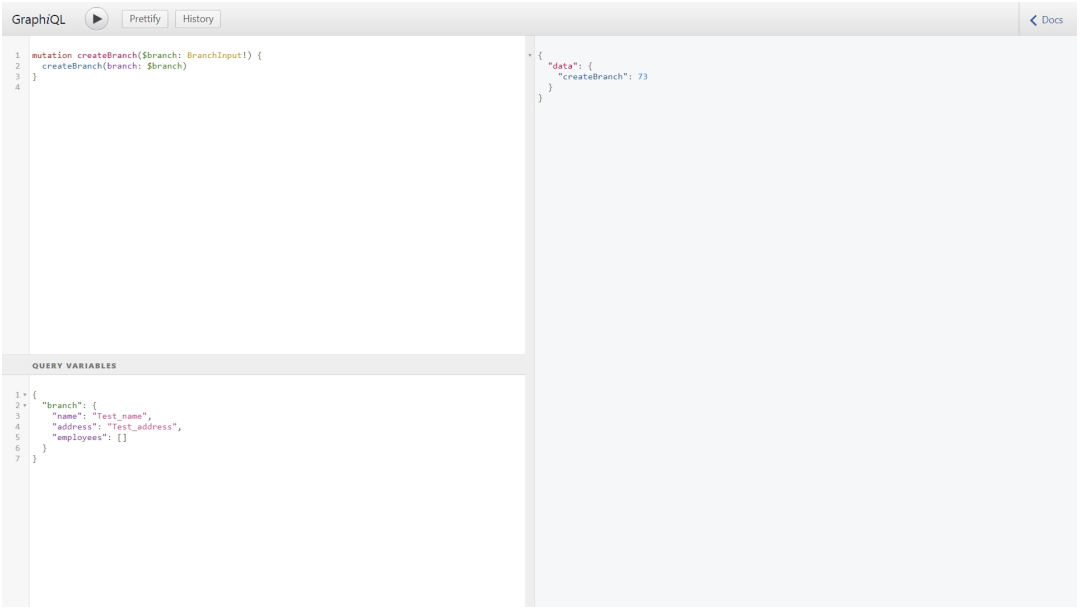
Kapitola 7

Testovanie aplikácie

Kapitola sa venuje rôznym nástrojom a spôsobom, ktoré boli použité pri testovaní aplikácie. Najskôr sú opísané priebežné testy, vykonané počas vývoja aplikácie (kapitola 7.1), následne záverečné testy uskutočnené zamestnancami firmy, pre ktorú je aplikácia vyvinutá (kapitola 7.2). Záverečná kapitola (7.3) obsahuje námety na zlepšenie aplikácie.

7.1 Priebežné testovanie

Pre priebežné testovanie prvej a druhej iterácie vývoja aplikácie bol použitý nástroj GraphiQL[1]. Ide o vývojové prostredie integrované v rámci webového prehliadača, ktoré vyvinul Facebook, pre možnosť vytvárania dopytov a skúmania GraphQL API [20]. Podporuje zvýrazňovanie syntaxe, automatické doplňanie kódu či upozornenia na chyby pri písaní dopytov alebo mutácií, ktoré možno poslať priamo na server prostredníctvom webového prehliadača.

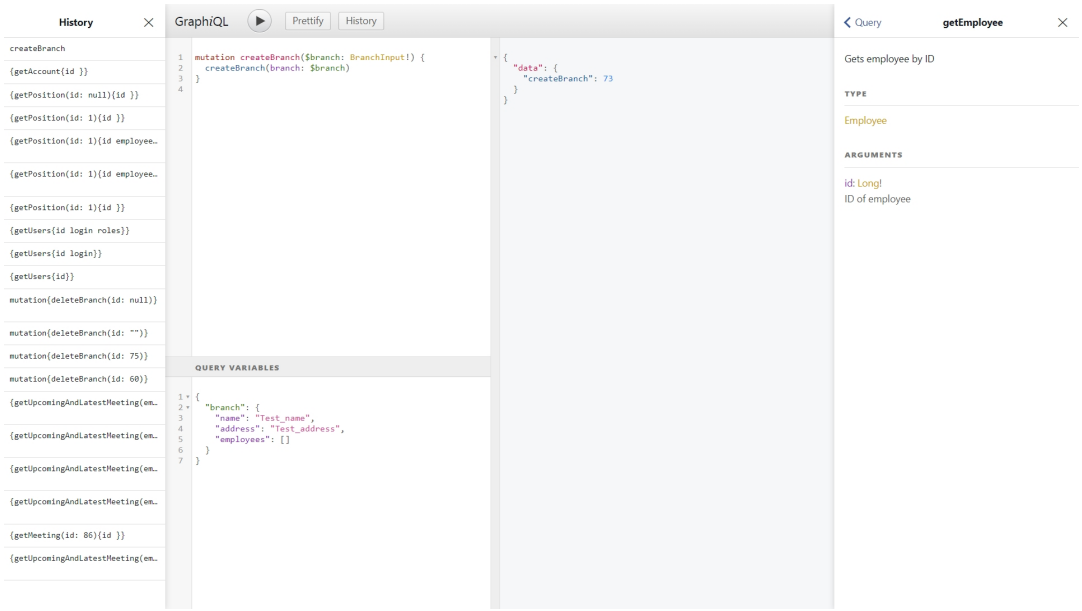


Obr. 7.1: Nástroj GraphiQL – základné rozloženie

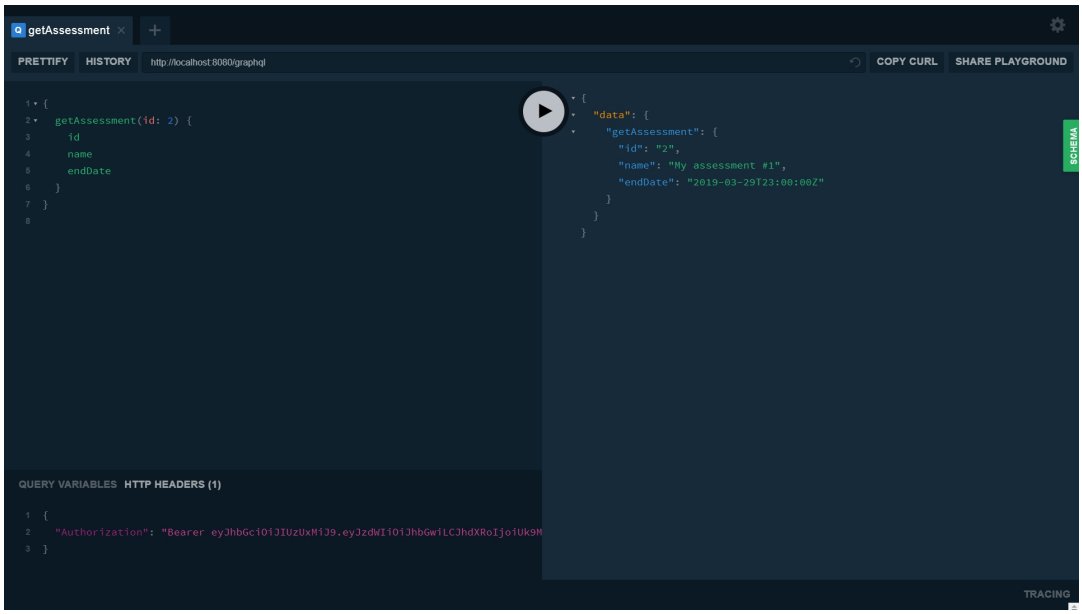
[1] https://github.com/graphql/graphiql

Prostredie pozostáva z viacerých častí (obr. 7.1). Hlavný panel je určený pre písanie dopytov či mutácií, po odoslaní pomocou tlačidla pre spustenie sa v pravom paneli zobrazí odpoveď od servera. Ak operácia vyžaduje premenné, je možné ich napísať do spodného panela.

GraphiQL ďalej zobrazuje históriu zadávaných operácií pomocou tlačidla „History“ (obr. 7.2). Po kliknutí na tlačidlo „Docs“ v pravom hornom rohu, sa zobrazí panel pre prehliadanie dokumentácie schémy, ktorá je čítaná z definovaných schém na serveri.



Obr. 7.2: Nástroj GraphiQL – história operácií a dokumentácia schémy



Obr. 7.3: Nástroj GraphQL Playground – podpora špecifikácie HTTP hlavičiek

Pre otestovanie dopytov a mutácií definovaných v druhej iterácii, ktoré vyžadovali autorizovaný prístup, bol použitý nástroj GraphQL Playground[2] (obr. 7.3). Ten, na rozdiel od GraphiQL, pridáva ďalšie možnosti, hlavne zadávanie HTTP hlavičiek, kde boli vkladané vygenerované *tokeny*.

Počas vývoja aplikácie bola testovaná ako prvá, tak druhá iterácia aplikácie. Najskôr bol implementovaný základ serverovej časti aplikácie, ktorý bol testovaný pomocou vyššie uvedených nástrojov. Následne sa pridala klientská časť, ktorá tiež bola samostatne otestovaná. Po prepojení oboch častí prebiehali systémové testy, tj. či obe časti spolu fungujú podľa očakávaní.

Pri posielaní dopytov či mutácií na server pomocou GraphiQL, alebo vypĺňaní formulárov na klientovi, boli použité funkčné a nefunčné testy. Funkčné testy sa zameriavajú na kontrolu, či aplikácia spĺňa definované požiadavky [7]. Nefunkčné testy kontrolujú, či sa aplikácia chová správne pri zadaní nesprávnych hodnôt.

Ďalším z testov bolo overenie responzivity aplikácie. Klientská časť aplikácie pracuje vo webovom prehliadači, preto je nutné overiť, či funguje správne nielen v rôznych prehliadačoch, ale aj iných zariadeniach. Testovanie tohto typu prebiehalo v prehliadačoch Mozilla Firefox (v. 66.0) a Google Chrome (v. 74.0), kde sa aplikácia správala podľa očakávaní. Aplikácia bola tiež odskúšaná na mobile a tablete so systémom Android.

7.2 Záverečné testovanie

Po dokončení a otestovaní všetkých častí aplikácie bolo realizované záverečné testovanie, ktoré prebiehalo u zákazníka. Testovania sa zúčastnil tím 5 zamestnancov (vedúci tímu a 4 členovia), ktorým boli v aplikácii vytvorené účty a k nim poskytnuté prihlasovacie údaje. Každému zo zamestnancov boli pridelené rôzne používateľské roly okrem vedúceho, ktorému boli pridelené všetky.

Zamestnancom bola predložená séria rôznych testov vo forme predvyplnenej tabuľky – každá obsahovala číslo a názov daného testu, kroky tvoriace test, očakávaný a skutočný výstup testu, úspešnosť jednotlivých krokov a nakoniec prípadné poznámky. Testy sa zameriavali na funkčnosť operácií špecifických pre konkrétne roly. Test pre vedúceho pozostával z niekoľkých vybraných operácií. Zamestnanci mali za úlohu najskôr konkrétne testy realizovať, následne zadať skutočný výstup testu, posúdiť jeho úspešnosť a spísať poznámky v prípade neúspechu, alebo ak pri teste nastali nejaké problémy. Až na drobné nedostatky, ako napr. nesprávne zobrazenie niektorých elementov, dopadli testy úspešne. Ukážka testov pre zamestnanca disponujúceho všetkými rolami je zobrazená v prílohe F.

7.3 Námety na zlepšenie

Po otestovaní aplikácie sa zamestnanci zhodli na viacerých vylepšeniach, ktoré by aplikácia v budúcnosti mohla obsahovať:

- export plánovaných stretnutí, napr. do aplikácie Google Calendar alebo Microsoft Outlook
- automatické posielanie upozornení o zmenách na e-mail
- evidencia absolvovaných školení
- skóre zamestnancov by mohlo byť počítané aj na základe rýchlosti plnenia cieľov

[2] https://github.com/prisma/graphql-playground

Kapitola 8

Záver

V tejto práci bol čitateľ oboznámený s tromi hlavnými typmi klient-server aplikácií: tenký klient, bohatý klient a bohatý webový klient. Následne boli opísané dnes najpopulárnejšie technológie pre tvorbu aplikačných programových rozhraní, a to najmä REST a GraphQL, ktorých kľúčové vlastnosti boli porovnané. Práca sa však viac zameriava na GraphQL, keďže túto technológiu využíva aj výsledná aplikácia.

Analýza a návrh webovej aplikácie, ktorá sa zaoberá osobným rozvojom zamestnancov, boli ďalšími časťami tejto práce. Špecifikované požiadavky boli do detailov analyzované a podľa nich vytvorené diagramy prípadov použitia spolu s opisom jednotlivých prípadov. Následne bola navrhnutá GraphQL schéma definujúca typy a vzťahy medzi nimi, ktoré v aplikácii vystupujú. Návrh architektúry bol rozdelený na časti, ktoré boli jednotlivo opísané, pričom databáza bola navrhnutá v samostatnej kapitole, a to pomocou ER diagramu. Kapitola pre analýzu a návrh aplikácie bola zakončená návrhom používateľského rozhrania, kde boli aplikované rôzne prístupy doplnené o konkrétne ukážky.

Implementáciu aplikácie bolo z hľadiska komplexnosti nutné rozdeliť na 2 časti: serverovú a klientskú. Pri oboch bola aplikovaná technológia GraphQL spôsobom špecifickým pre použité nástroje. Boli taktiež opísané vybrané funkcie aplikácie s ohľadom na GraphQL.

Čo sa týka samotnej technológie GraphQL, tá je oproti používaným alternatívam zložitejšia na implementáciu. Na serveri bolo nutné definovať navyše schému, ktorá však prináša výhody jednoznačného určenia typov a ich vzťahov, ako aj dopytov a mutácií dostupných na jednom koncovom bode.

Jednoduchosť prichádza z pohľadu klienta, keďže oproti REST-u je definovaný len jeden koncový bod, na ktorý sú požiadavky posielané. Navyše dotazovací jazyk (QL) umožňuje klientovi žiadať dáta v tvare a štruktúre, akej potrebuje. Dôkazom bol napr. dopyt pre získanie dát potrebných pre *dashboard* (kód 6.5). V prípade REST-u by bolo potrebné špecifikovať viacero koncových bodov, navyše namiesto jedného vnoreného dopytu by museli byť dáta načítané pomocou viacerých HTTP volaní.

Čo sa však naopak stalo problémom GraphQL pri implementácií aplikácie, bolo vytvorenie dopytu pre organizačnú štruktúru spoločnosti. Tá má stromovú štruktúru s neznámou hĺbkou, pre ktorú podľa definície jazyka GraphQL nie je jednoduché vytvoriť dopyt. Klient by potreboval poznať hĺbku dopredu, aby následne mohol vytvoriť požadovaný dopyt, čo je nepraktické.

Pre zhrnutie, GraphQL je silným nástrojom, ktorý prináša množstvo výhod, je však stále ešte v začiatkoch. Dôležitým faktom je, že bol vytvorený spoločnosťou Facebook, kvôli množstve a tvarom dopytov, s ktorými sa potrebovali efektívne vysporiadať. Z tohto dôvodu sa GraphQL hodí skôr pre komplexnejšie aplikácie, ktoré vyžadujú väčšiu flexibilitu

z hľadiska získavania dát. Pre jednoduchšie API vyhovuje použitie REST-u. Ako však tvrdia niektorí autori ([5]), takmer všetko čo sa dá spraviť pomocou GraphQL, je možné dosiahnuť aj pomocou REST-u, avšak za cenu efektivity.

Výsledná aplikácia bola otestovaná postupmi uvedenými v záverečnej kapitole. Súčasťou bolo aj testovanie zamestnancami firmy, pre ktorých boli definované špecifické testy. Až na menšie nedostatky, dopadli testy úspešne. Zamestnanci podali námety na zlepšenie aplikácie, akú funkčnosť by ešte mohla poskytovať.

Literatúra

[1] Ahuja, A.; Ahuja, N.: Thin Clients: Secure and Cost Effective Client Access Devices for Government Organizations. Január 2007.

[2] Bagui, S.; Earp, R.: *Database Design Using Entity-Relationship Diagrams, Second Edition*. Boston, MA, USA: Auerbach Publications, druhé vydanie, 2011, ISBN 9781439861769.

[3] Cohn, M.: *Agile Estimating and Planning*. Robert C. Martin Series, Pearson Education, 2005, ISBN 9780132703109.

[4] Cosmina, I.; Harrop, R.; Schaefer, C.; aj.: *Pro Spring 5: An In-Depth Guide to the Spring Framework and Its Tools*. Berkeley, CA, USA: Apress, piate vydanie, 2017, ISBN 9781484228074.

[5] Doerrfeld, B.; Sandoval, K.; Wood, C.: *GraphQL or Bust: To Use It Or Not: That Is The Question*. Leanpub, 2018.

[6] Few, S.: *Information Dashboard Design: The Effective Visual Communication of Data*. O'Reilly Media, Inc., 2006, ISBN 0596100167.

[7] Filipova, O.; Vilão, R.: *Software Development From A to Z: A Deep Dive into all the Roles Involved in the Creation of Software*. Apress, 2018, ISBN 9781484239445.

[8] Flanagan, D.: *JavaScript: The Definitive Guide: Activate Your Web Pages (Definitive Guides)*. O'Reilly Media, 2011, ISBN 0596805527.

[9] Fraternali, P.; Comai, S.; Bozzon, A.; aj.: Engineering rich internet applications with a model-driven approach. *ACM Transactions on the Web (TWEB)*, ročník 4, č. 2, 2010: str. 7.

[10] Freeman, A.: *Pro Angular 6*. Apress, 2018, ISBN 9781484236499.

[11] Jin, B.; Sahni, S.; Shevat, A.: *Designing Web APIs: Building APIs That Developers Love*. O'Reilly Media, 2018, ISBN 9781492026921.

[12] Keith, M.; Schincariol, M.; Nardone, M.: *Pro JPA 2 in Java EE 8: An In-Depth Guide to Java Persistence APIs*. Apress, 2018, ISBN 9781484234204.

[13] Lange, F.: *Eclipse rich ajax platform: Bringing rich client to the web*. Apress, 2008, ISBN 1430218835.

[14] Layka, V.: *Learn Java for Web Development: Modern Java Web Development*. Berkeley, CA, USA: Apress, prvé vydanie, 2014, ISBN 9781430259831.

[15] Mabey, M.; Doupé, A.; Zhao, Z.; aj.: dbling: Identifying extensions installed on encrypted web thin clients. *Digital Investigation*, ročník 18, 2016: s. S55–S65.

[16] Mathis, L.: *Designed for Use: Create Usable Interfaces for Applications and the Web*. Pragmatic Bookshelf, prvé vydanie, 2011, ISBN 9781934356753.

[17] Mew, K.: *Learning Material Design*. Packt Publishing, 2015, ISBN 9781785289811.

[18] Microsoft: *Smart Client Architecture and Design Guide (Patterns & Practices)*. Microsoft Press, 2004, ISBN 0735618534.

[19] Peterson, C.: *Learning Responsive Web Design: A Beginner's Guide*. O'Reilly Media, 2014, ISBN 9781449362942.

[20] Porcello, E.; Banks, A.: *Learning GraphQL: Declarative Data Fetching for Modern Web Apps*. O'Reilly Media, 2018, ISBN 1492030716.

[21] Reddy, K. S. P.: *Beginning Spring Boot 2: Applications and Microservices with the Spring Framework*. Apress, 2017, ISBN 97814842-29309.

[22] Shah, S.: *Maven for Eclipse*. Packt Publishing, 2014, ISBN 9781783987122.

[23] Tahaghoghi, S. M.; Williams, H. E.: *Learning MySQL: Get a Handle on Your Data*. O'Reilly Media, 2006, ISBN 9780596008642.

[24] Varanasi, B.; Belida, S.: *Spring REST*. Apress, 2015, ISBN 1484208242.

[25] Walls, C.: *Spring Boot in Action*. Greenwich, CT, USA: Manning Publications Company, prvé vydanie, 2016, ISBN 9781617292545.

[26] Wilken, J.; Aden, D.; Aden, J.: *Angular in Action*. Manning Publications Company, 2018, ISBN 9781617293313.

[27] Winch, R.; Mularien, P.: *Spring Security 3.1*. Packt Publishing, 2012, ISBN 9781849518260.

Príloha A

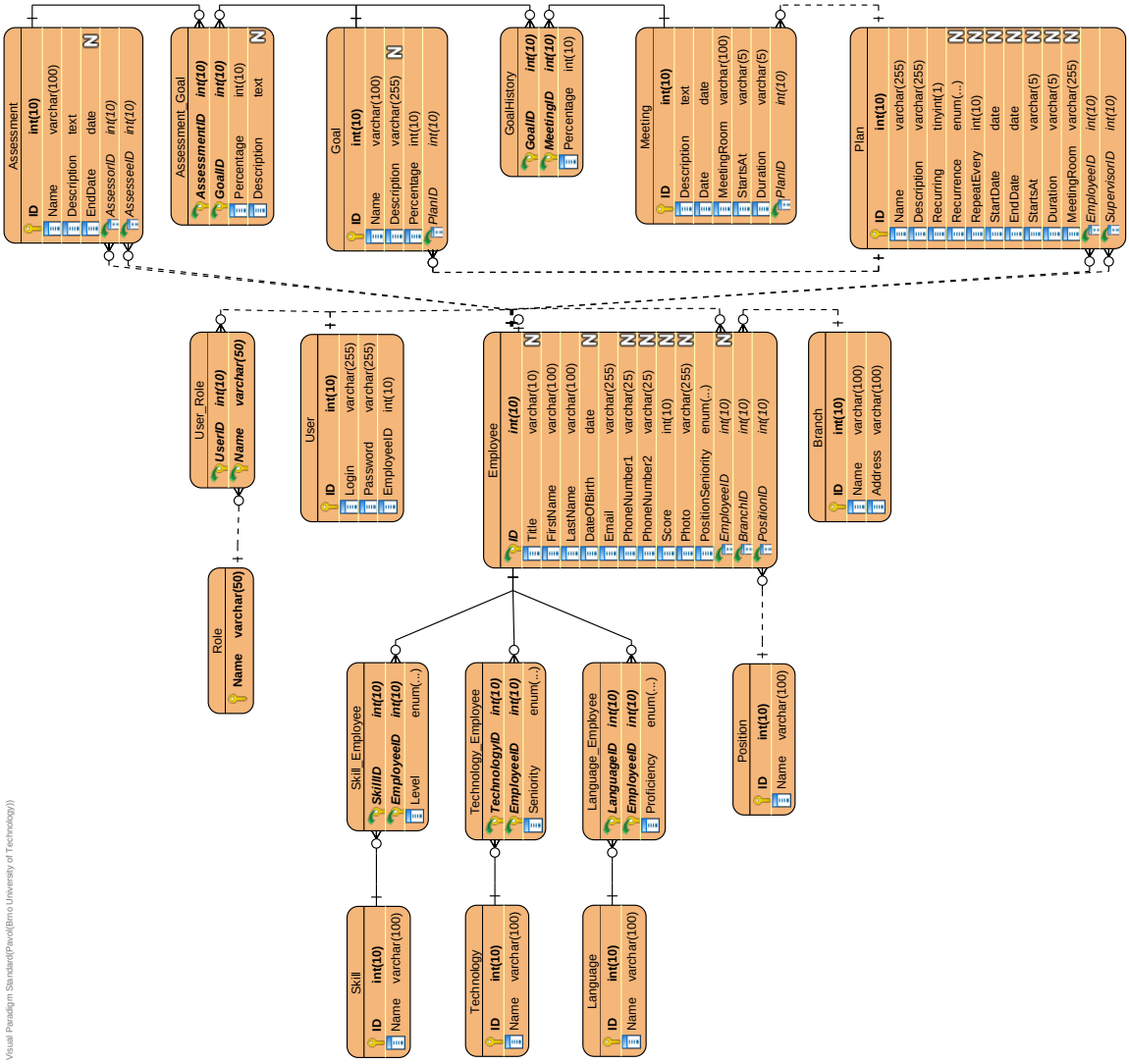
Obsah priloženého CD

Priložené CD pozostáva hlavne z týchto priečinkov a súborov:

- `client/` – priečinok so zdrojovými súbormi a programovou dokumentáciou pre klientskú časť aplikácie
- `server/` – priečinok so zdrojovými súbormi a programovou dokumentáciou pre serverovú časť aplikácie
- `latex/` – priečinok so zdrojovými súbormi pre vytvorenie textu práce (PDF)
- `xmalik14.pdf` – text diplomovej práce
- `README.txt` – návod pre spustenie aplikácie

Príloha B

Návrh databázy

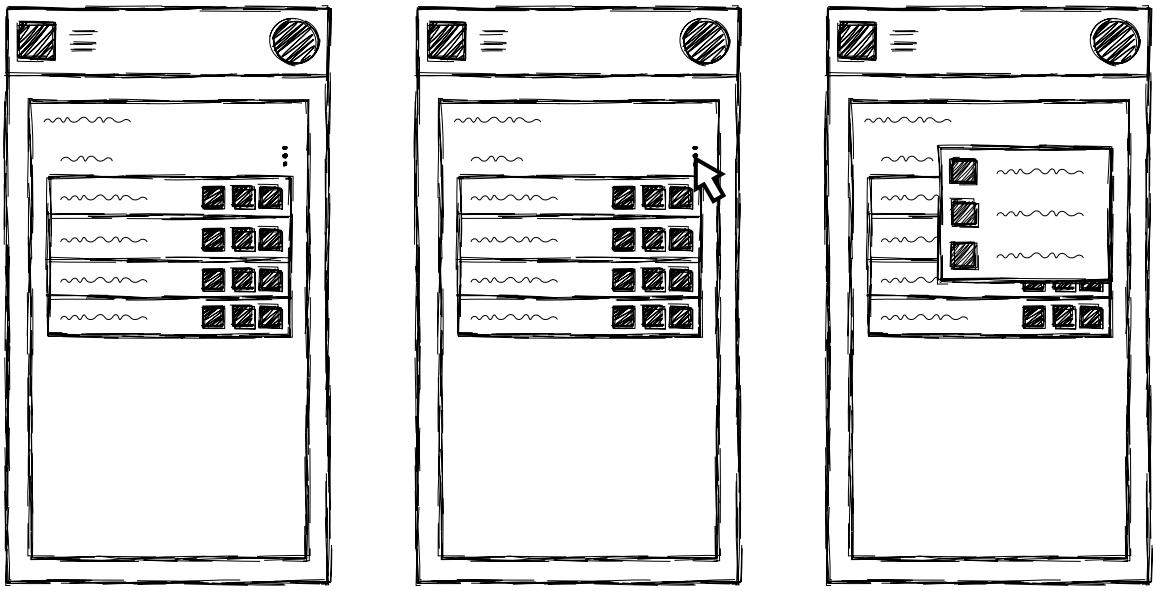


Obr. B.1: Navrhnutý ER diagram aplikácie

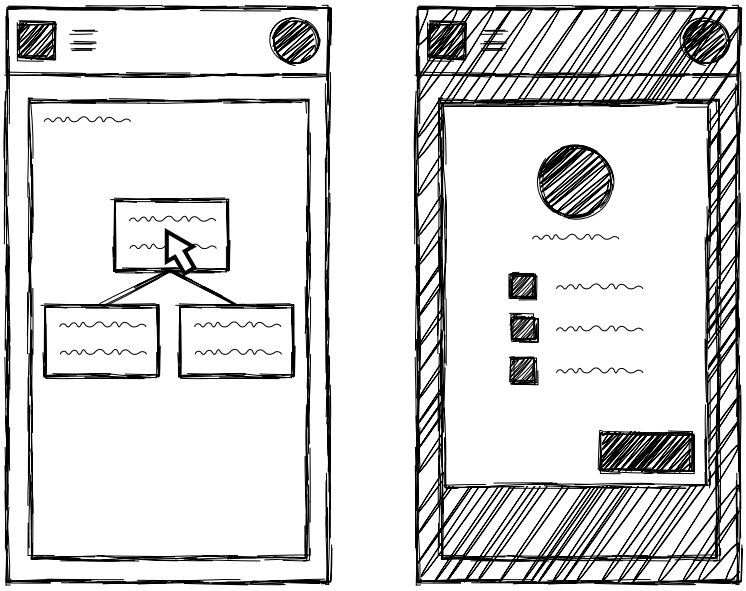
Príloha C

Návrhy používateľského rozhrania

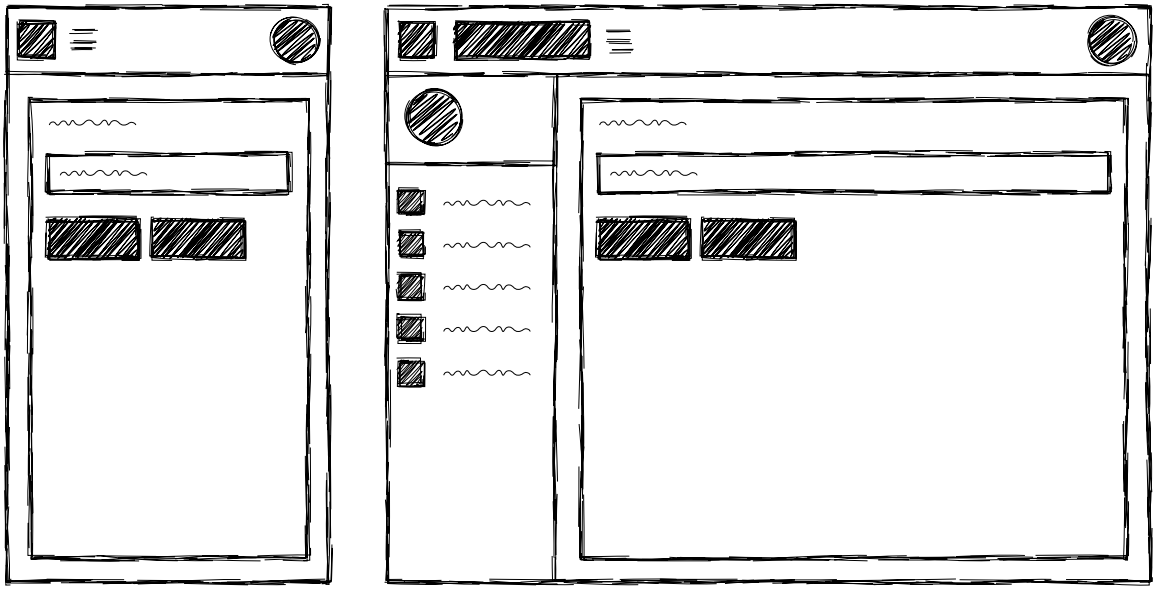
C.1 Náčrty



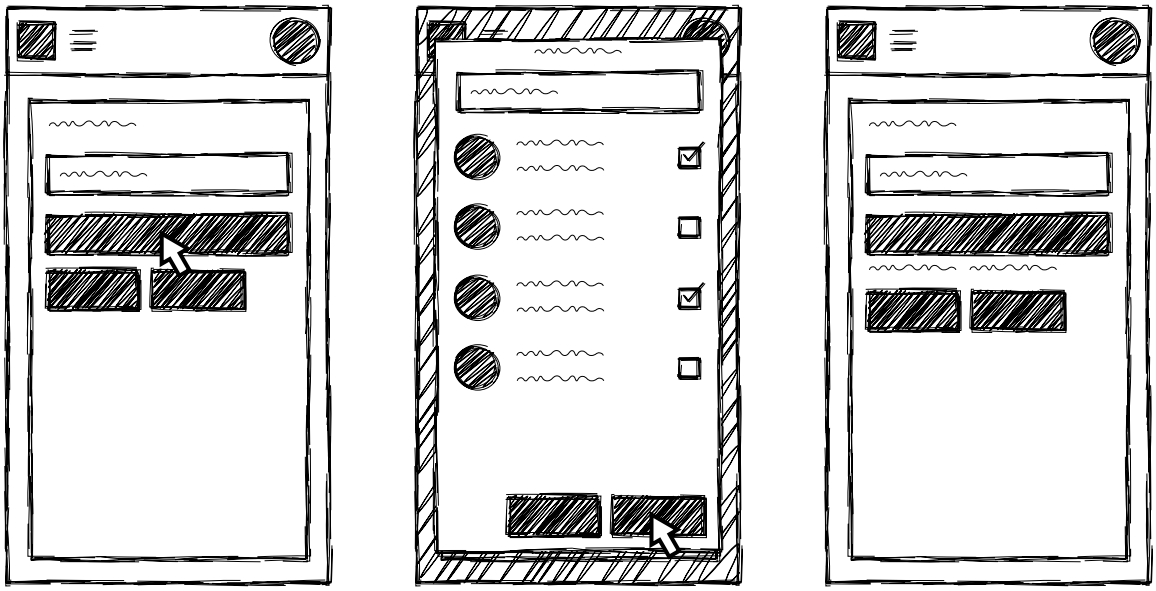
Obr. C.1: *Storyboard* aplikácie pre mobilné zariadenia – zoznamy



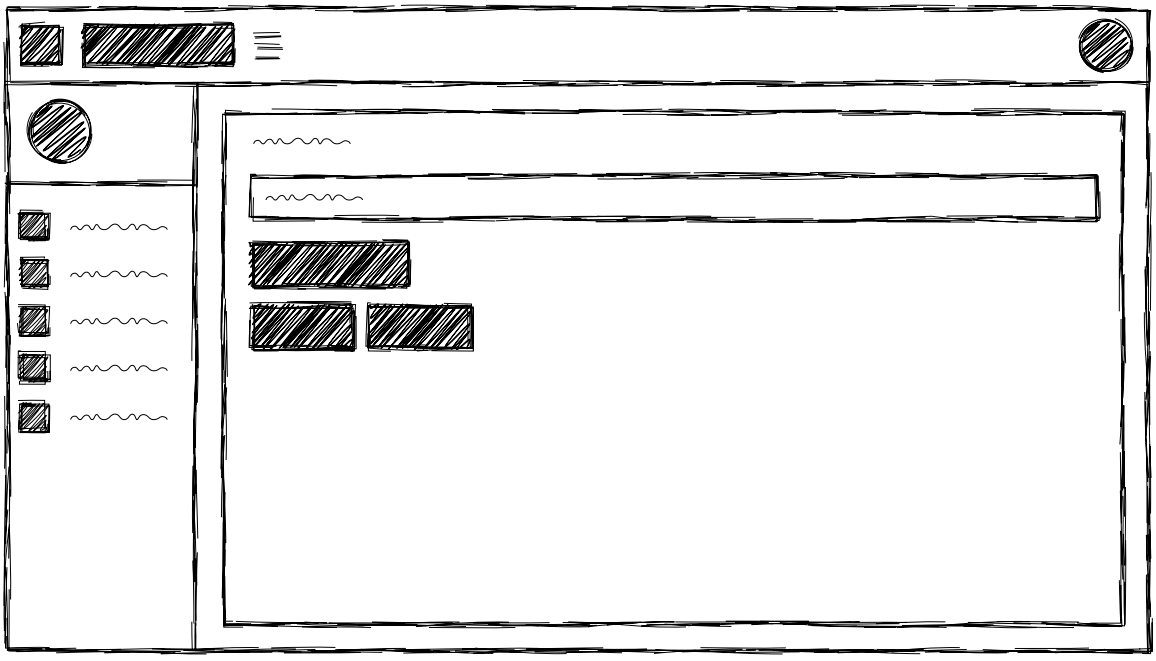
Obr. C.2: *Storyboard* aplikácie pre mobilné zariadenia – organizačná štruktúra



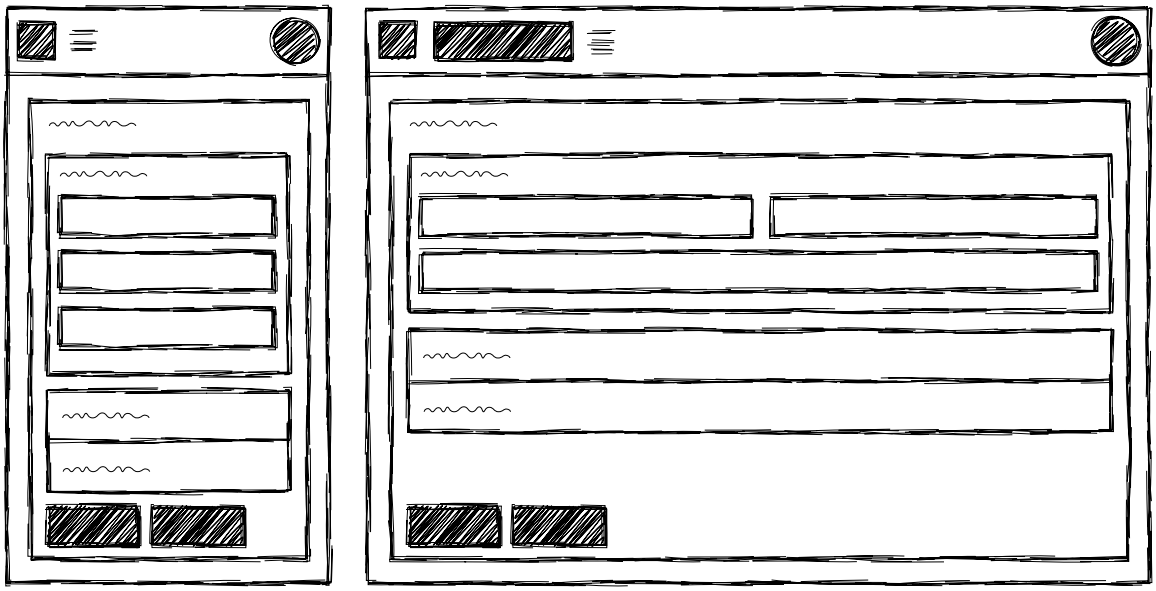
Obr. C.3: Štruktúra aplikácie pre mobilné (vľavo) a desktop (vpravo) zariadenia – jednoduché podsekcie



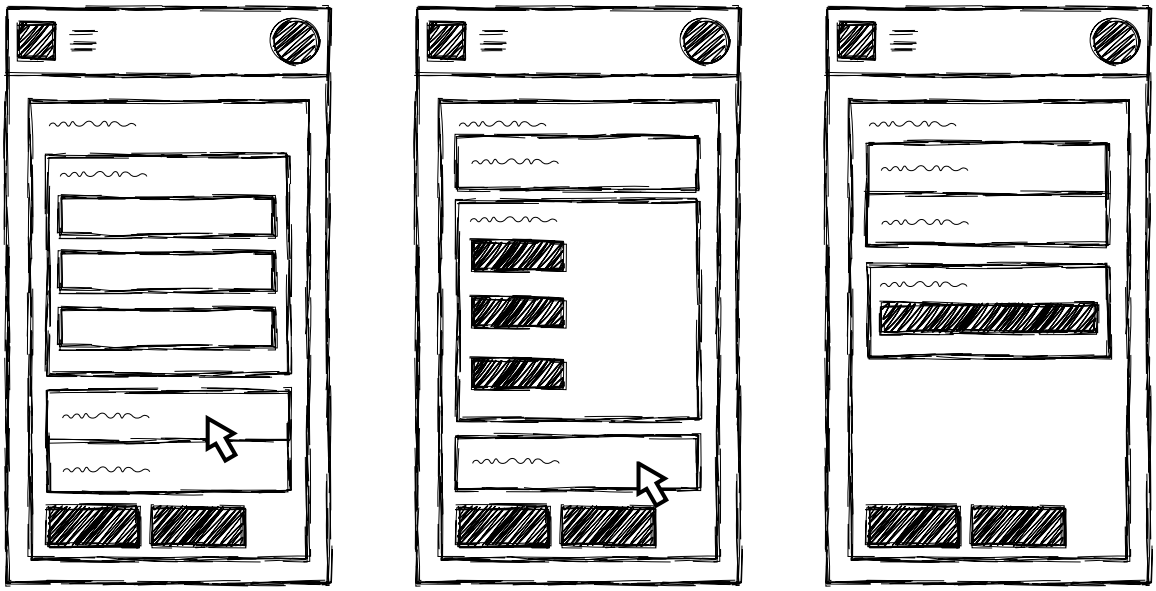
Obr. C.4: *Storyboard* aplikácie pre mobilné zariadenia – vyhľadanie a výber zamestnancov



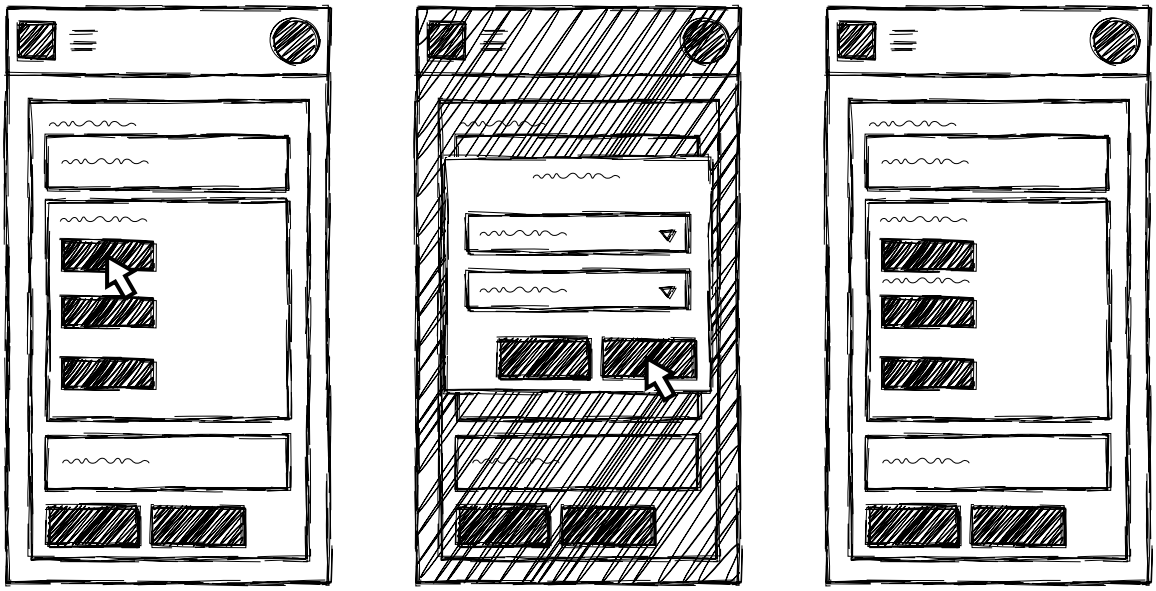
Obr. C.5: Štruktúra aplikácie pre desktop zariadenia – podsekcia s tlačidlom pre výber zamestnancov



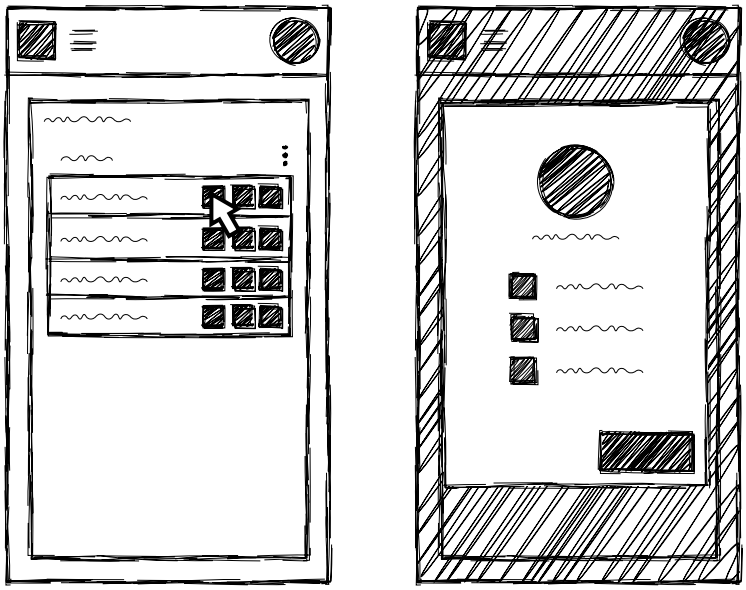
Obr. C.6: Štruktúra aplikácie pre mobilné (vľavo) a desktop (vpravo) zariadenia – pridanie alebo upravenie zamestnanca



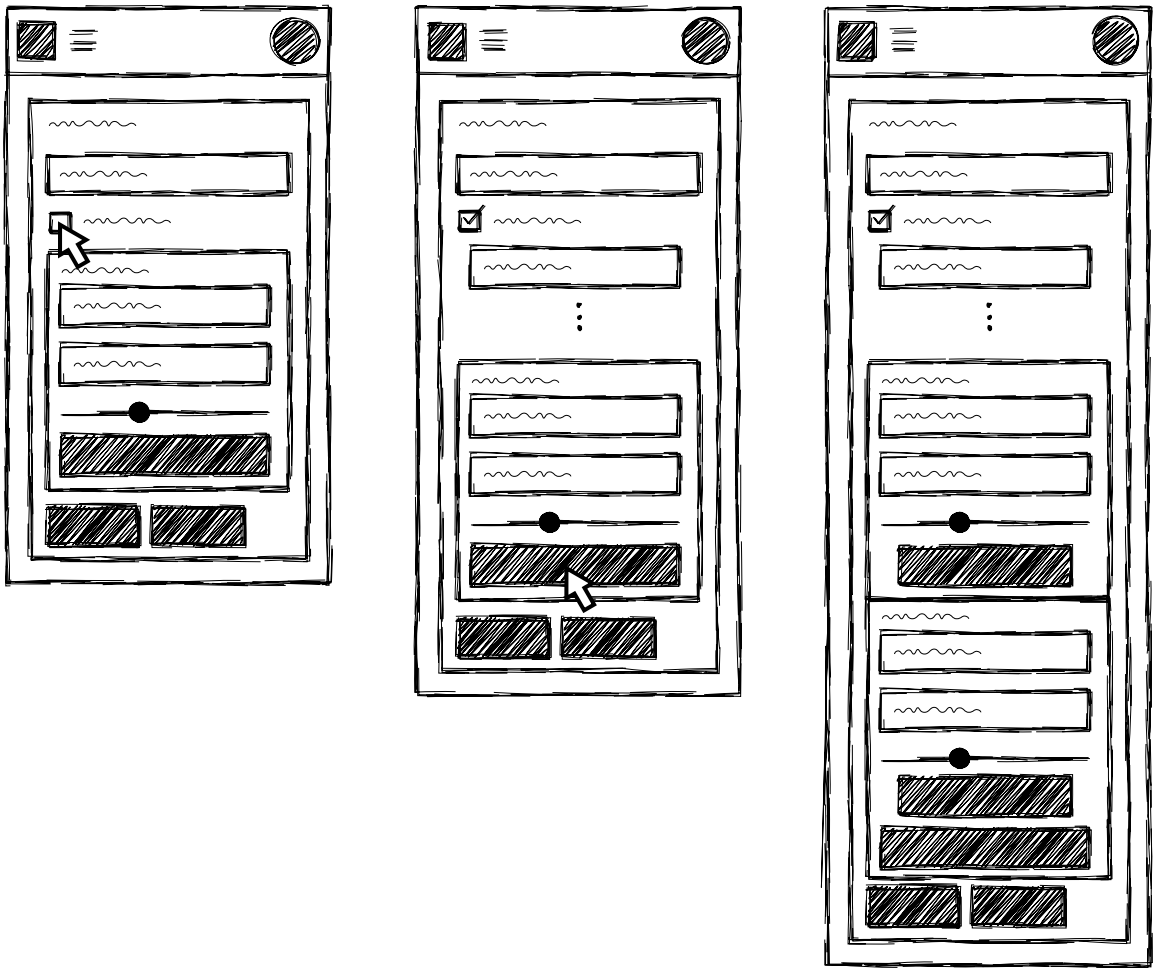
Obr. C.7: *Storyboard* aplikácie pre mobilné zariadenia – osobné údaje, osobný rozvoj a výber členov tímu zamestnanca



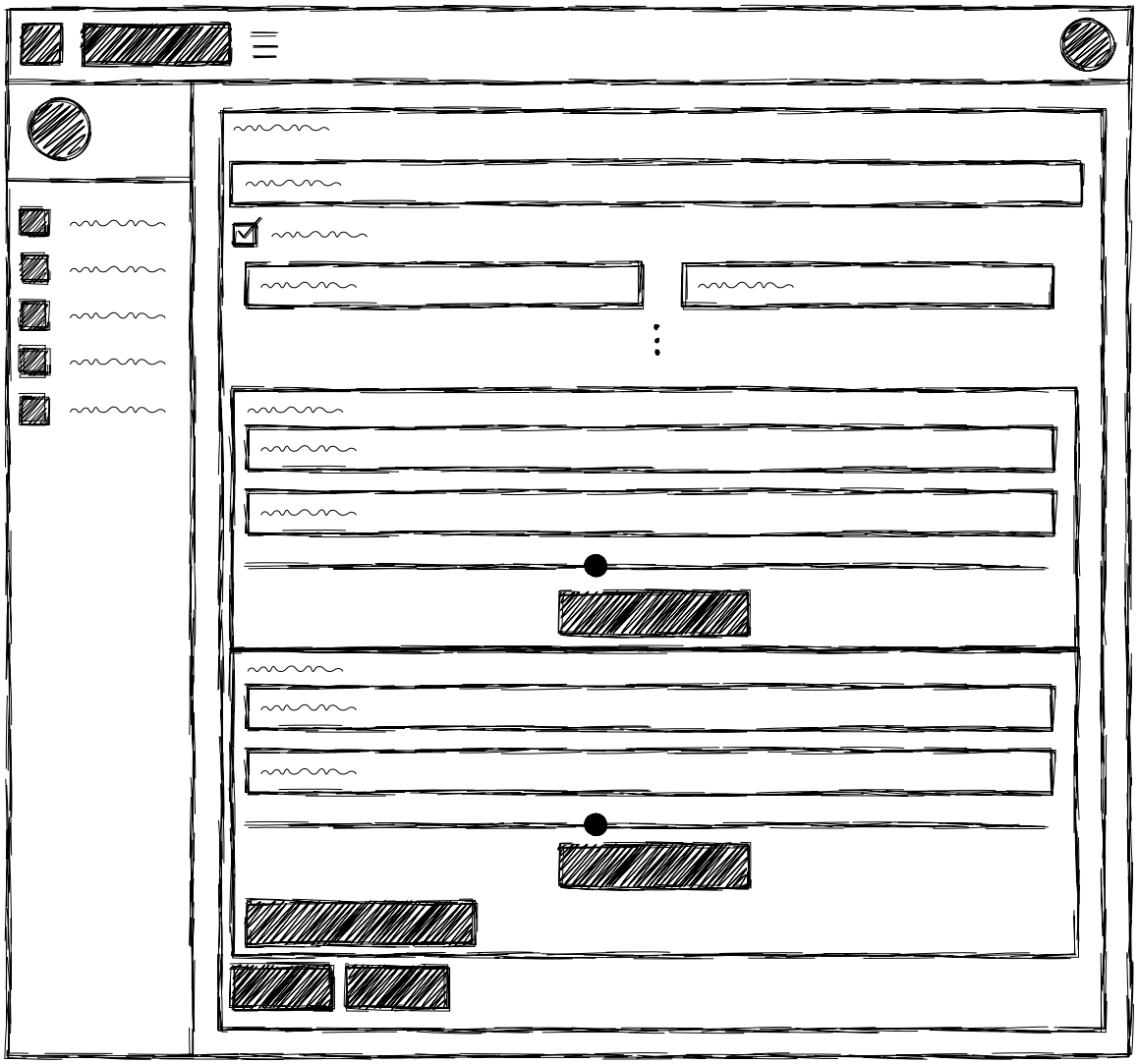
Obr. C.8: *Storyboard* aplikácie pre mobilné zariadenia – správa osobného rozvoja zamestnanca



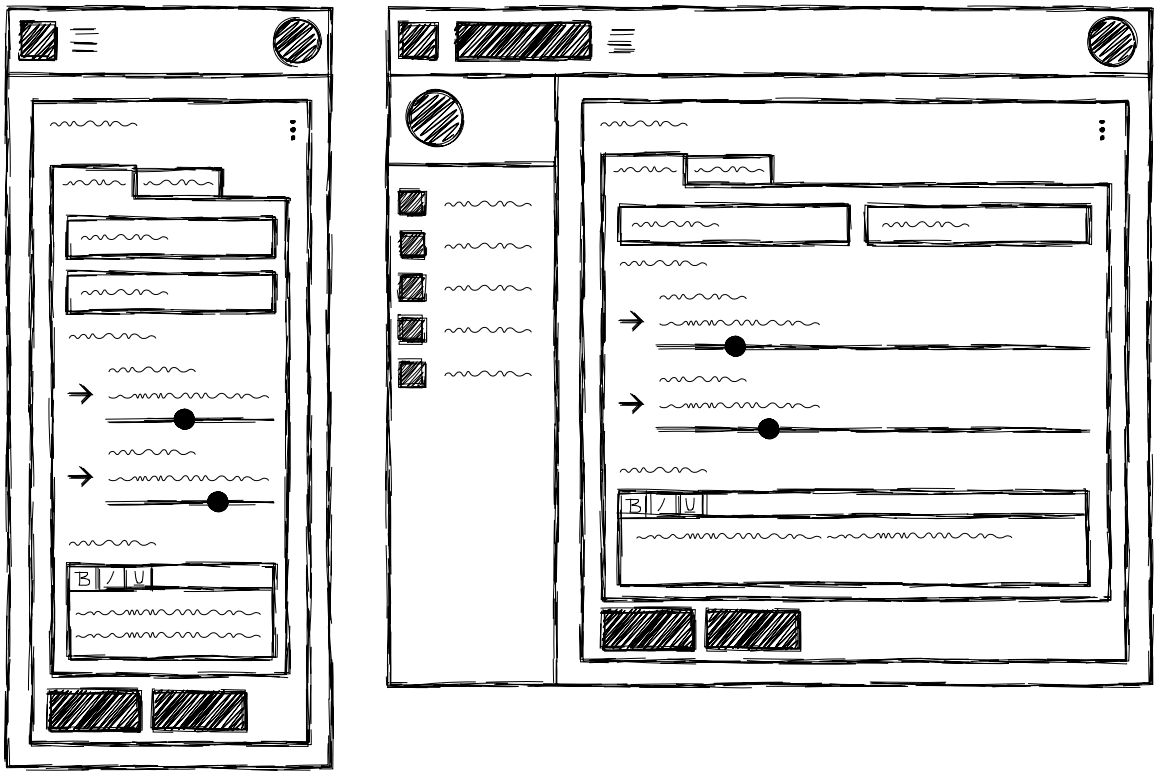
Obr. C.9: *Storyboard* aplikácie pre mobilné zariadenia – detail zamestnanca



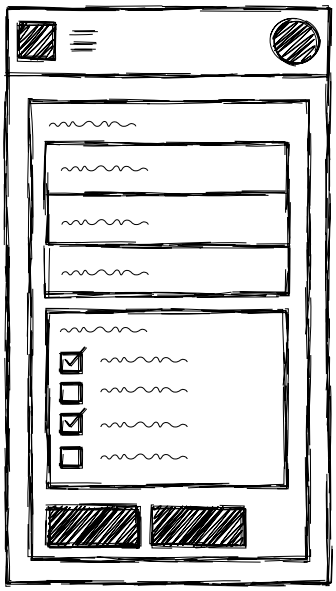
Obr. C.10: *Storyboard* aplikácie pre mobilné zariadenia – pridanie plánu



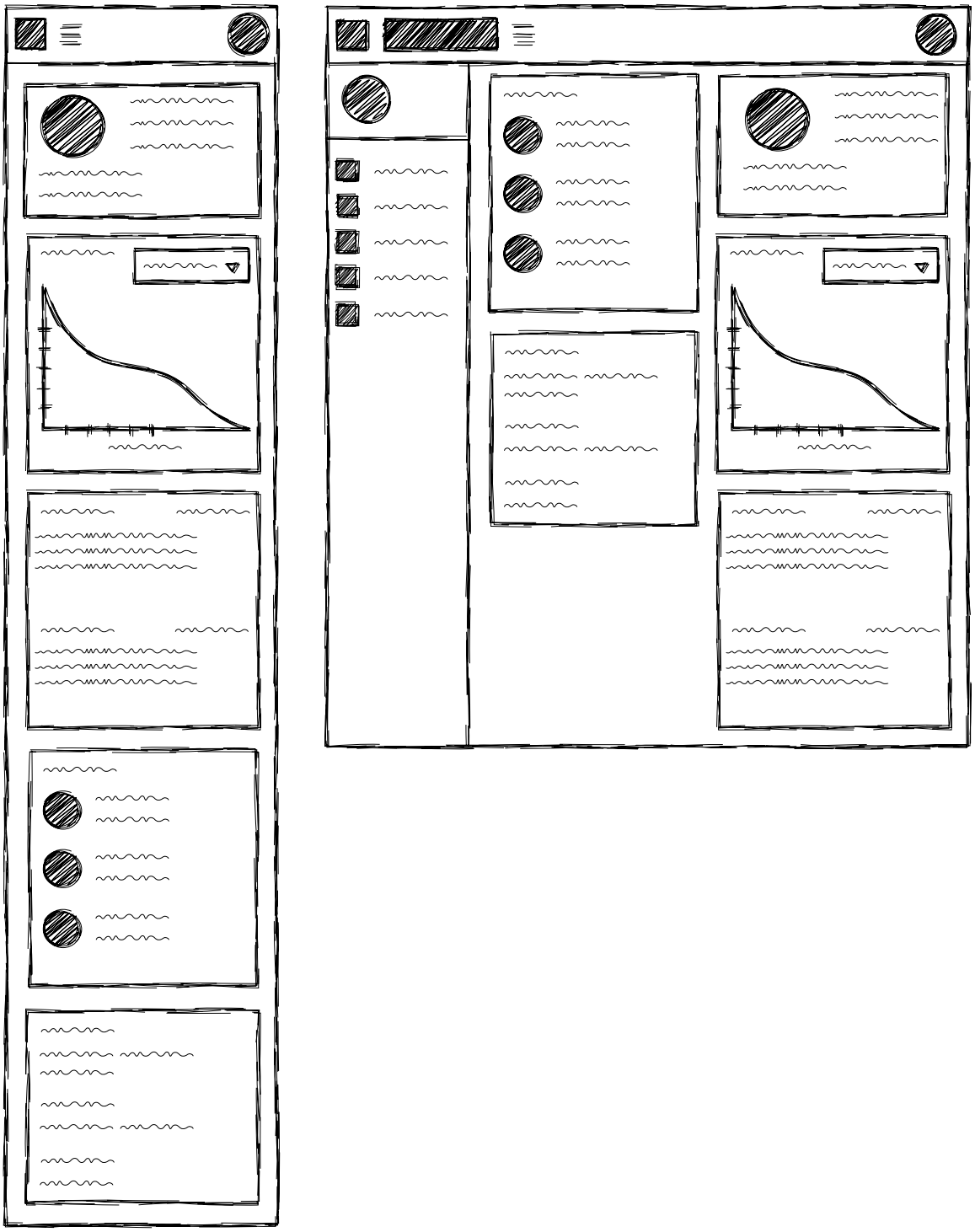
Obr. C.11: Štruktúra aplikácie pre desktop zariadenia – pridanie plánu



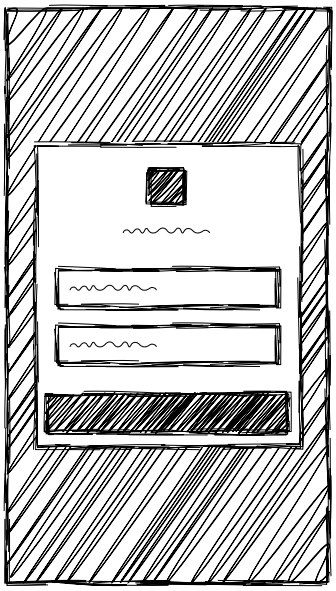
Obr. C.12: Štruktúra aplikácie pre mobilné (vľavo) a desktop (vpravo) zariadenia – správa plánu



Obr. C.13: Štruktúra aplikácie pre mobilné zariadenia – správa rolí zamestnanca

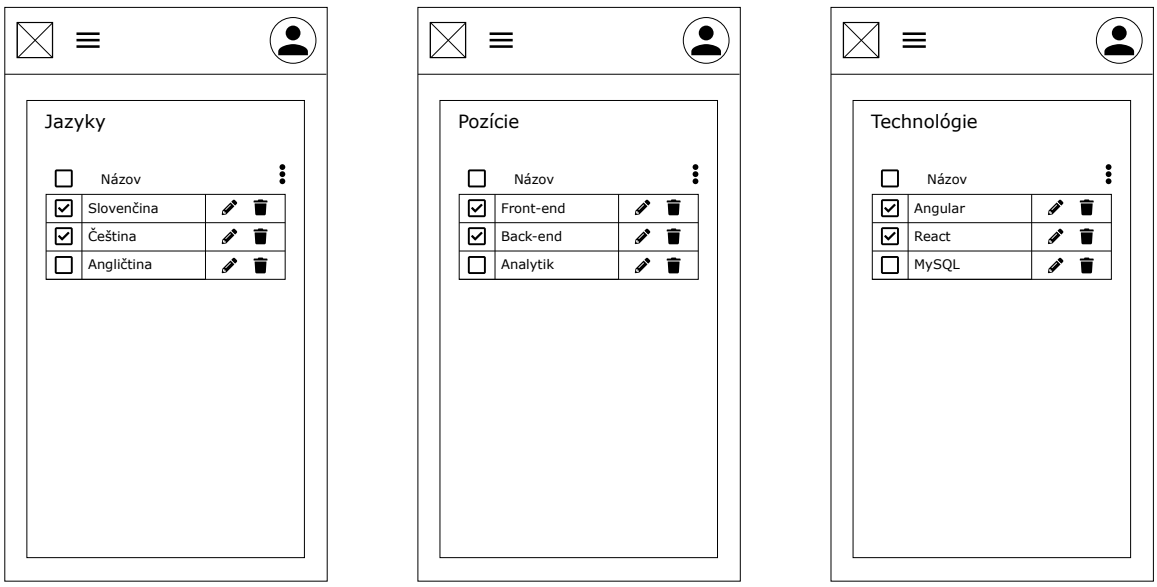


Obr. C.14: Štruktúra aplikácie pre mobilné (vľavo) a desktop (vpravo) zariadenia – *dashboard*

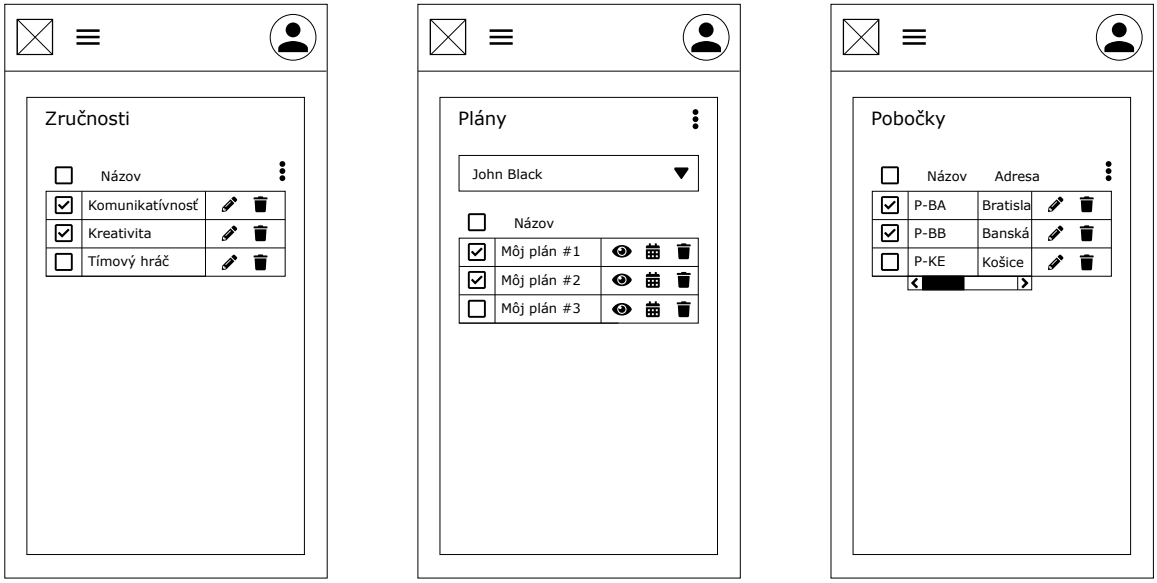


Obr. C.15: Štruktúra aplikácie pre mobilné zariadenia – prihlásenie

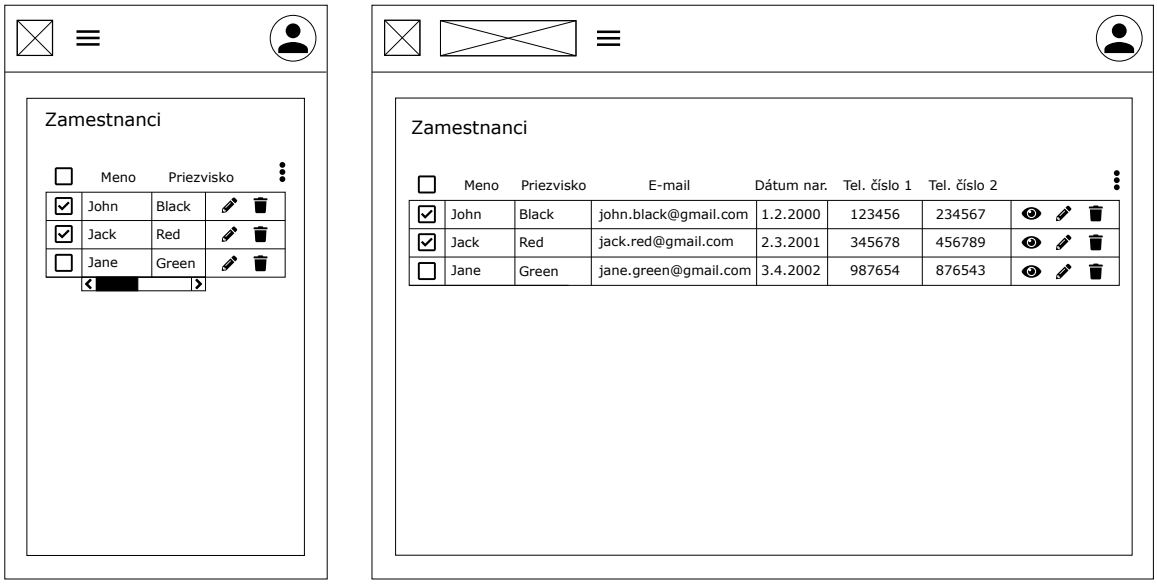
C.2 *Wireframy*



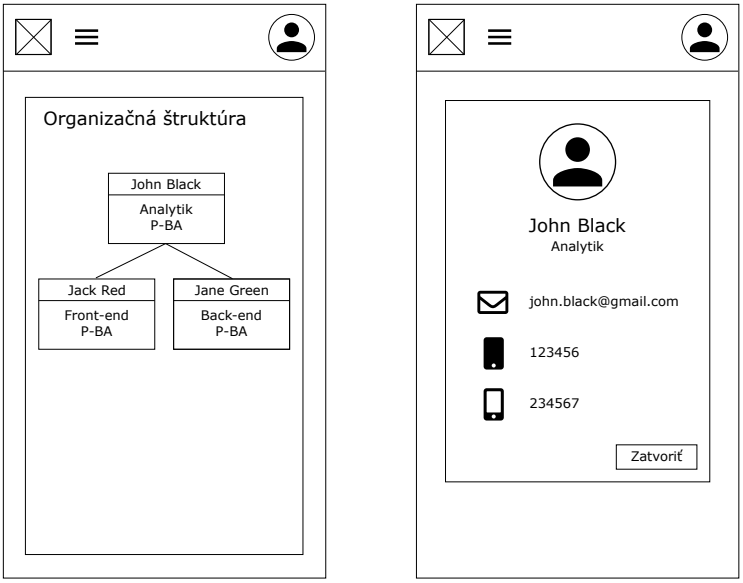
Obr. C.16: Tabuľky pre jazyky, pozície a technológie – mobilné zariadenia



Obr. C.17: Tabuľky pre zručnosti, plány a pobočky – mobilné zariadenia



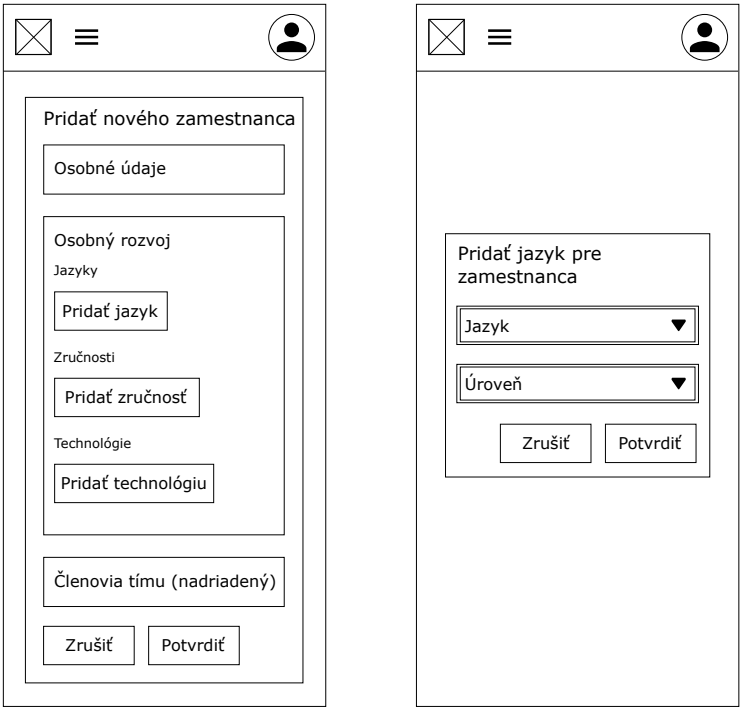
Obr. C.18: Tabuľka pre zamestnancov – mobilné (vľavo) a desktop (vpravo) zariadenia



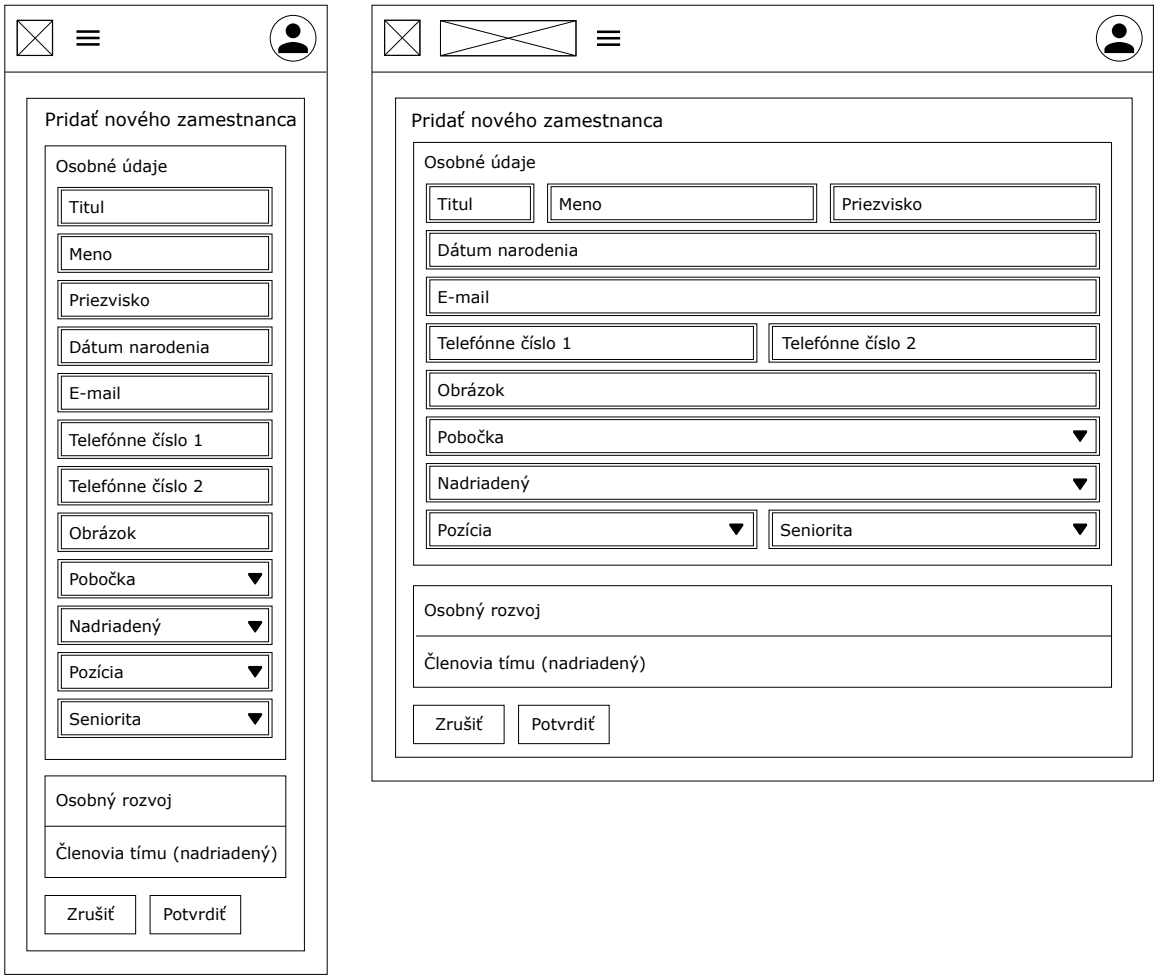
Obr. C.19: Organizačná štruktúra (vľavo), detail zamestnanca (vpravo) – mobilné zariadenia

Obr. C.20: Pridanie, resp. upravenie jednoduchých štruktúr (jazyky, pozície, technológie a zručnosti) – mobilné zariadenia

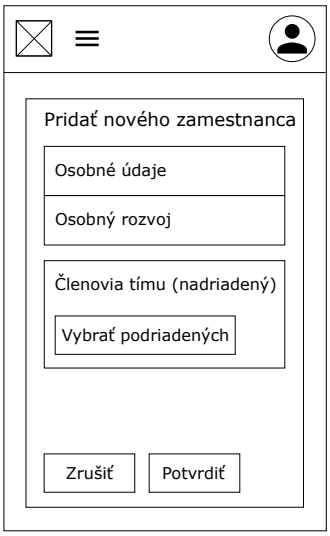
Obr. C.21: Pridanie, resp. upravenie pobočky (vľavo), výber zamestnancov (vpravo) – mobilné zariadenia



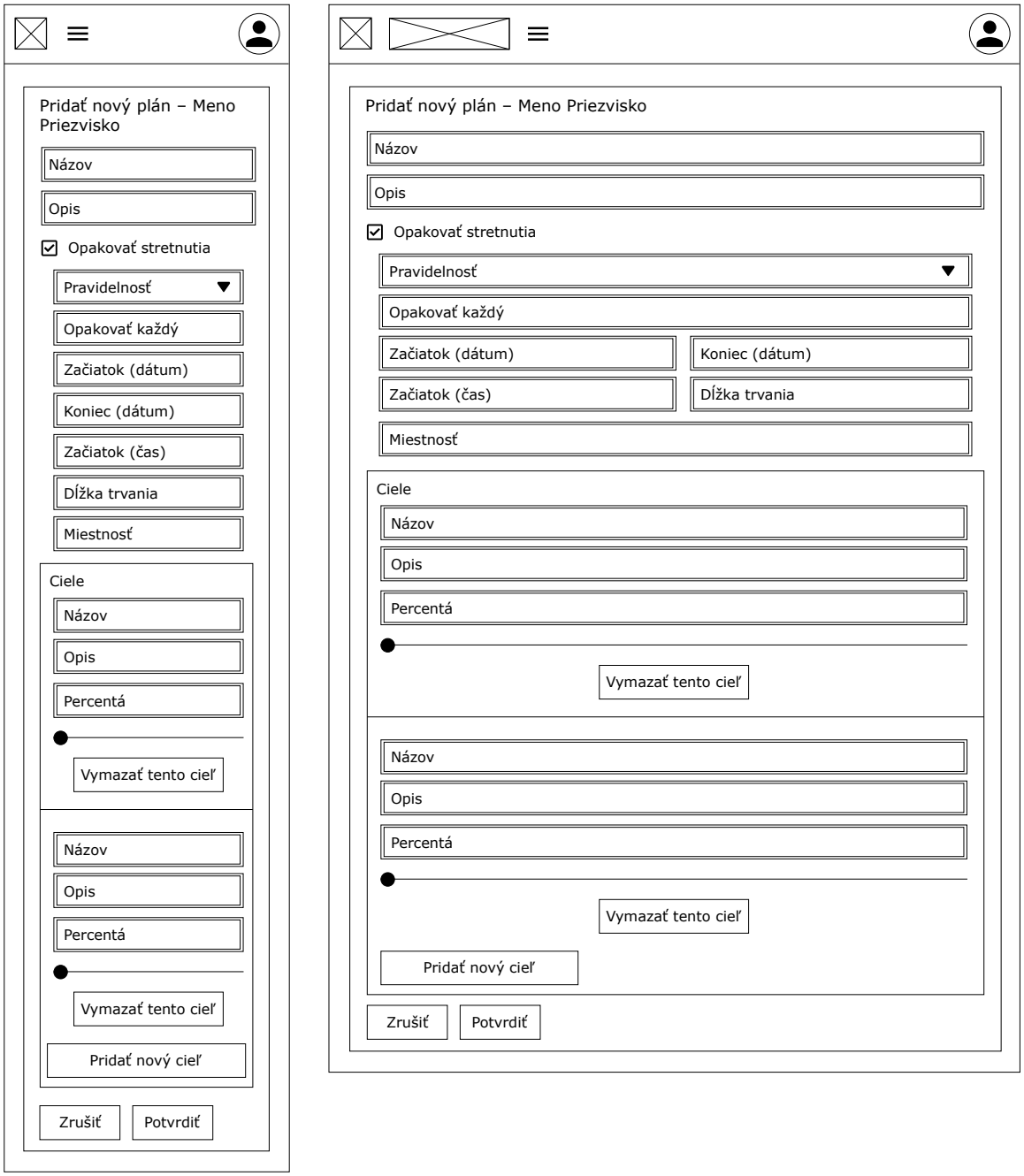
Obr. C.22: Pridanie, resp. upravenie zamestnanca (osobný rozvoj) – mobilné zariadenia



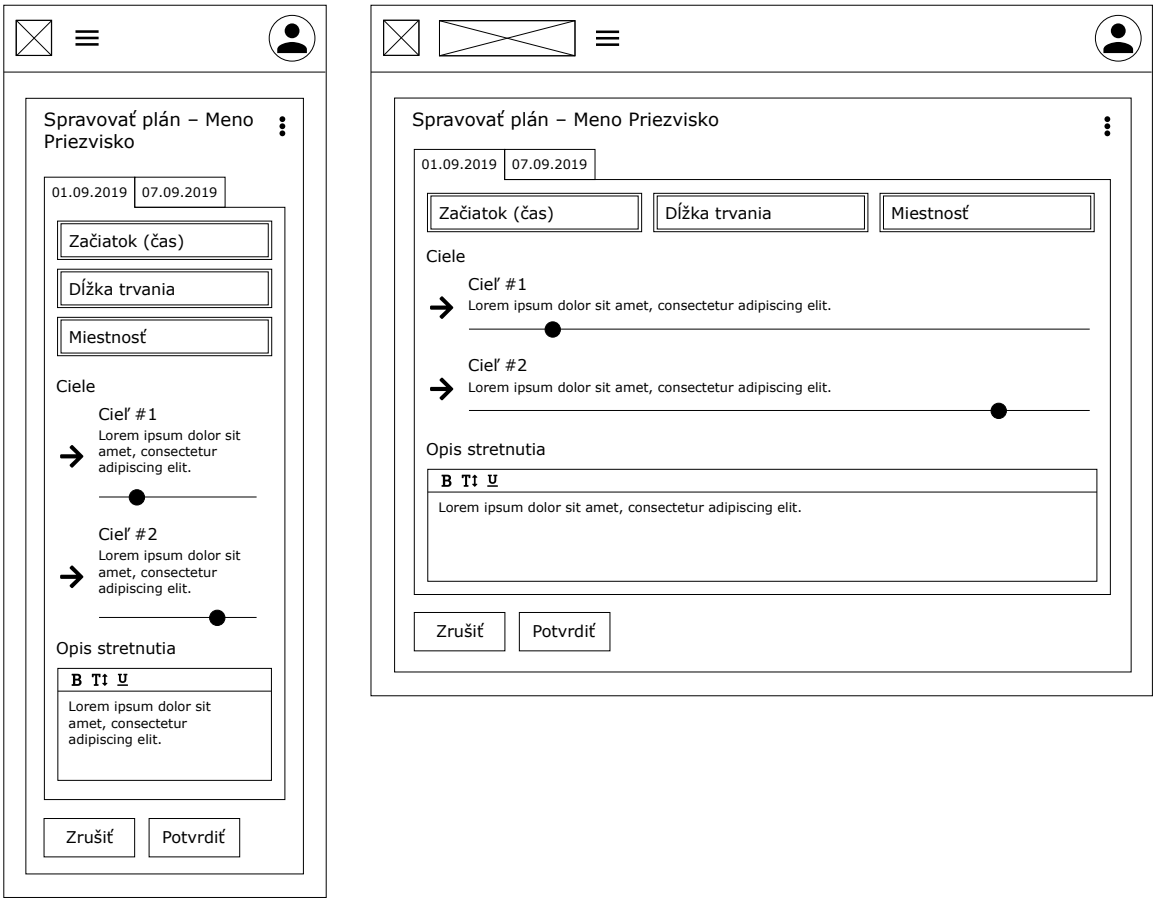
Obr. C.23: Pridanie, resp. upravenie zamestnanca (osobné údaje) – mobilné (vľavo) a desktop (vpravo) zariadenia



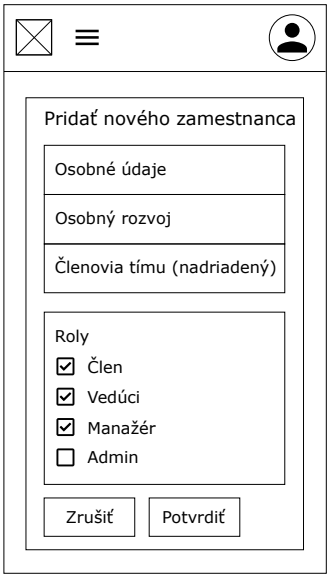
Obr. C.24: Pridanie, resp. upravenie zamestnanca (členovia tímu) – mobilné zariadenia



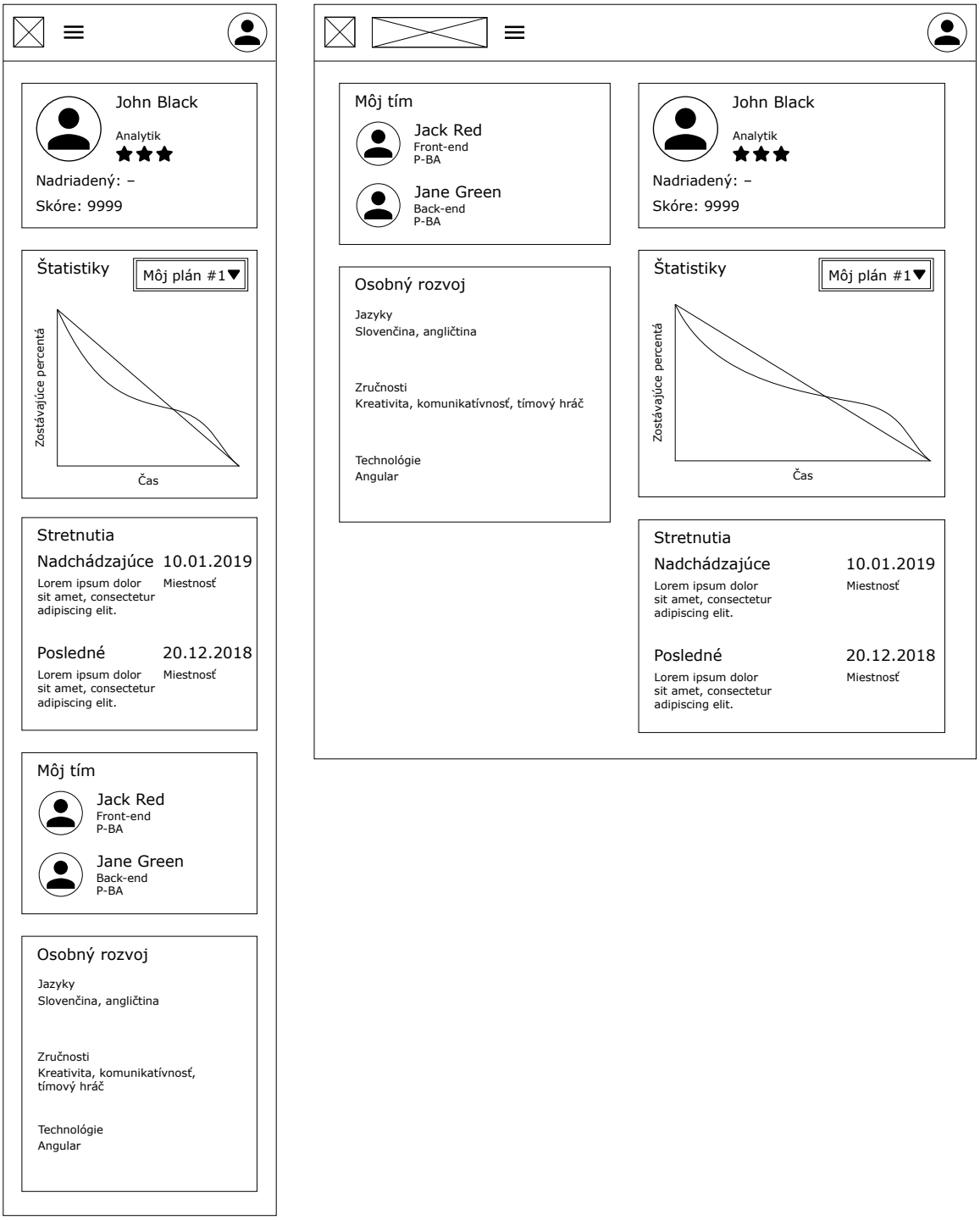
Obr. C.25: Pridanie, resp. upravenie plánu – mobilné (vľavo) a desktop (vpravo) zariadenia



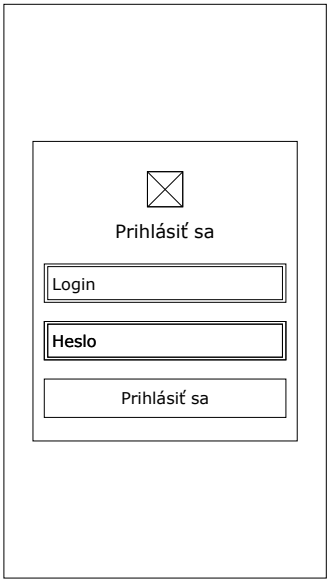
Obr. C.26: Spravovanie plánu – mobilné (vľavo) a desktop (vpravo) zariadenia



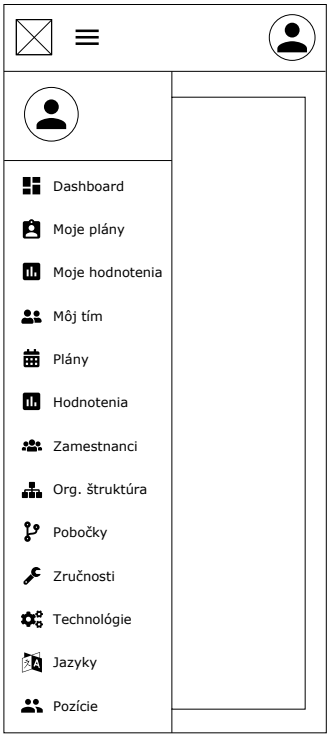
Obr. C.27: Pridanie, resp. upravenie zamestnanca (používateľské roly) – mobilné zariadenia



Obr. C.28: *Dashboard* – mobilné (vľavo) a desktop (vpravo) zariadenia

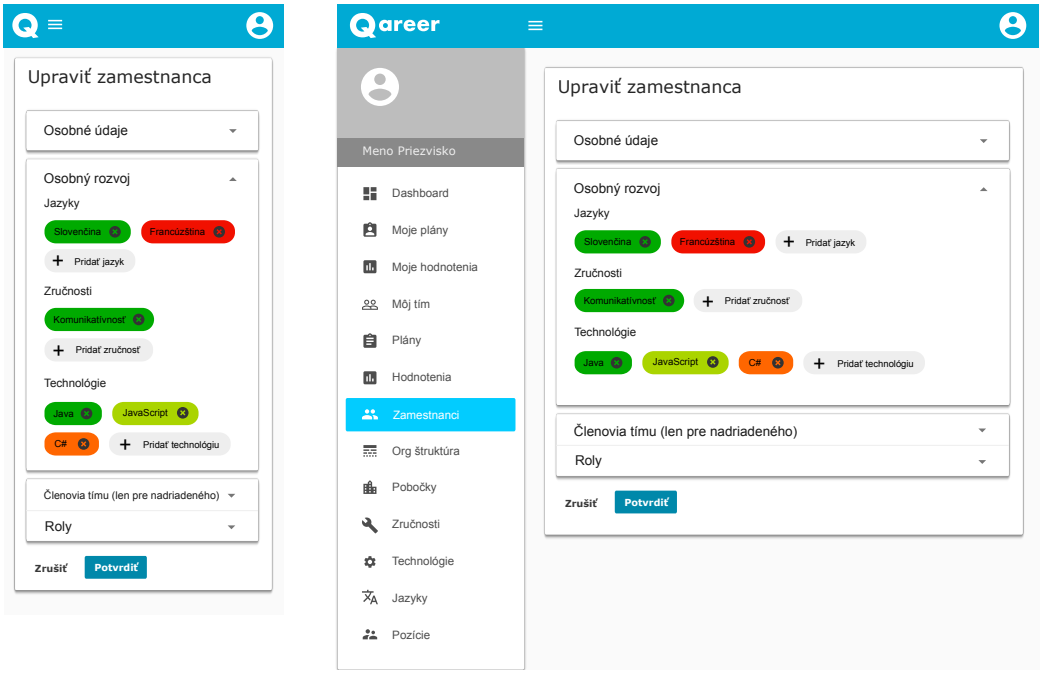


Obr. C.29: Prihlásenie – mobilné zariadenia

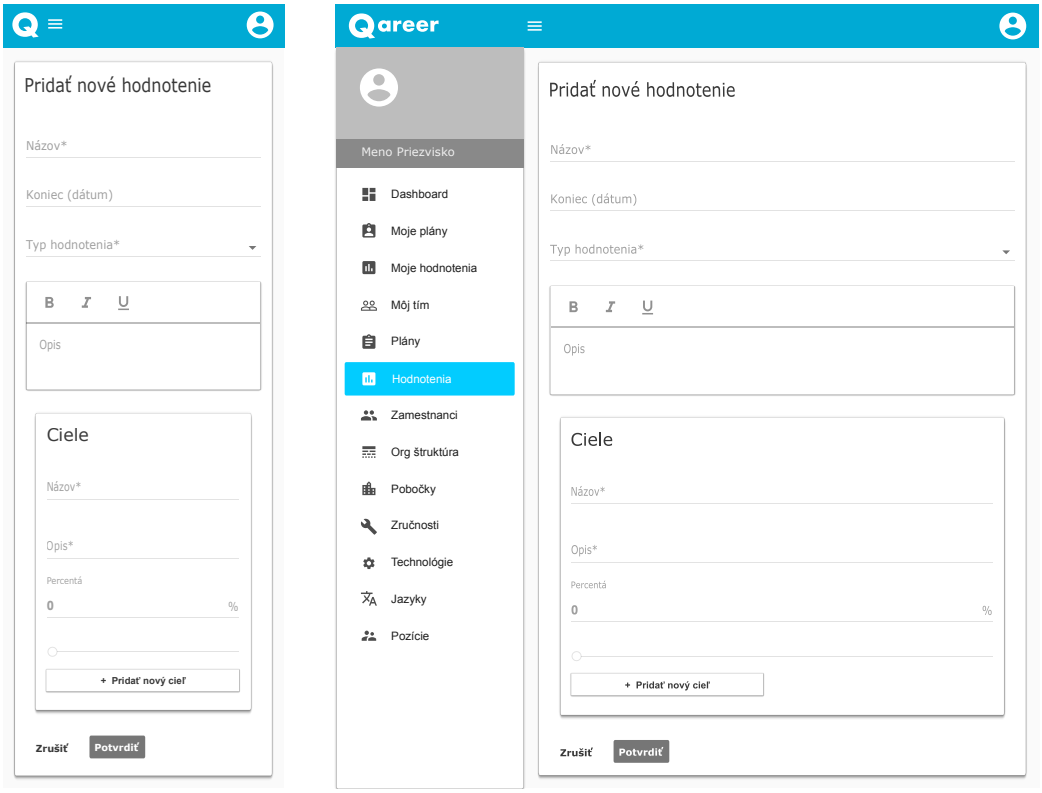


Obr. C.30: *Wireframe* aplikácie pre mobilné zariadenia – základná štruktúra

C.3 *Mock-upy*



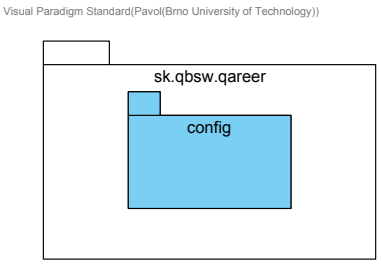
Obr. C.31: Úprava zamestnanca – mobilné (vľavo) a desktop (vpravo) zariadenia



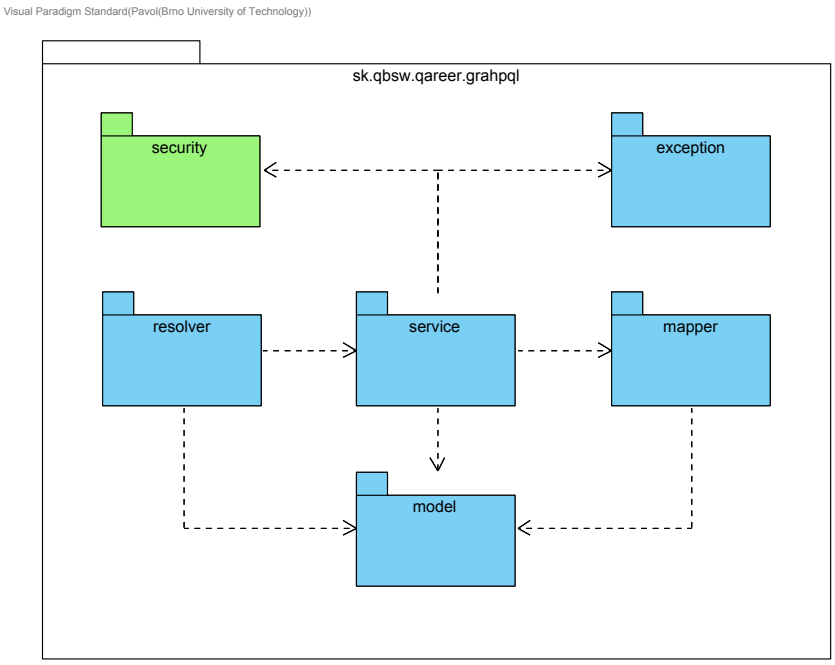
Obr. C.32: Pridanie hodnotenia – mobilné (vľavo) a desktop (vpravo) zariadenia

Príloha D

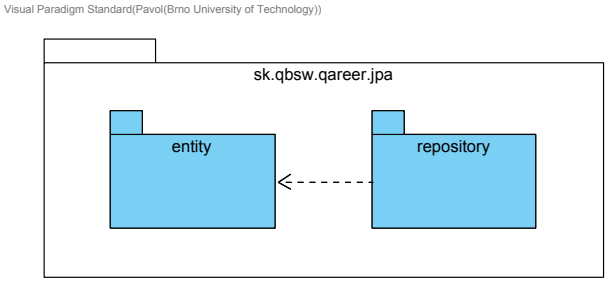
Štruktúra serverovej časti aplikácie



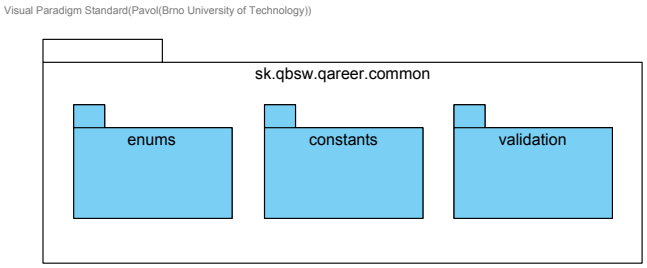
Obr. D.1: Balíčky modulu `qareer-boot` a závislosti medzi nimi



Obr. D.2: Balíčky modulu `qareer-graphql` a závislosti medzi nimi (balíček `security` je súčasťou 2. iterácie)



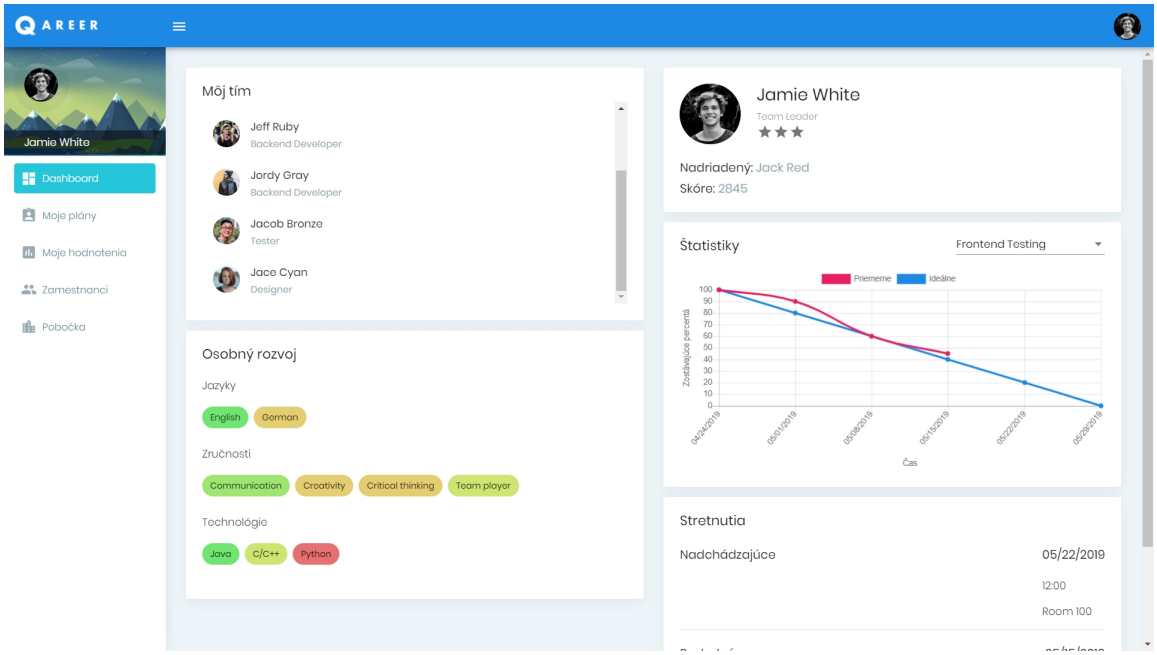
Obr. D.3: Balíčky modulu `qareer-jpa` a závislosti medzi nimi



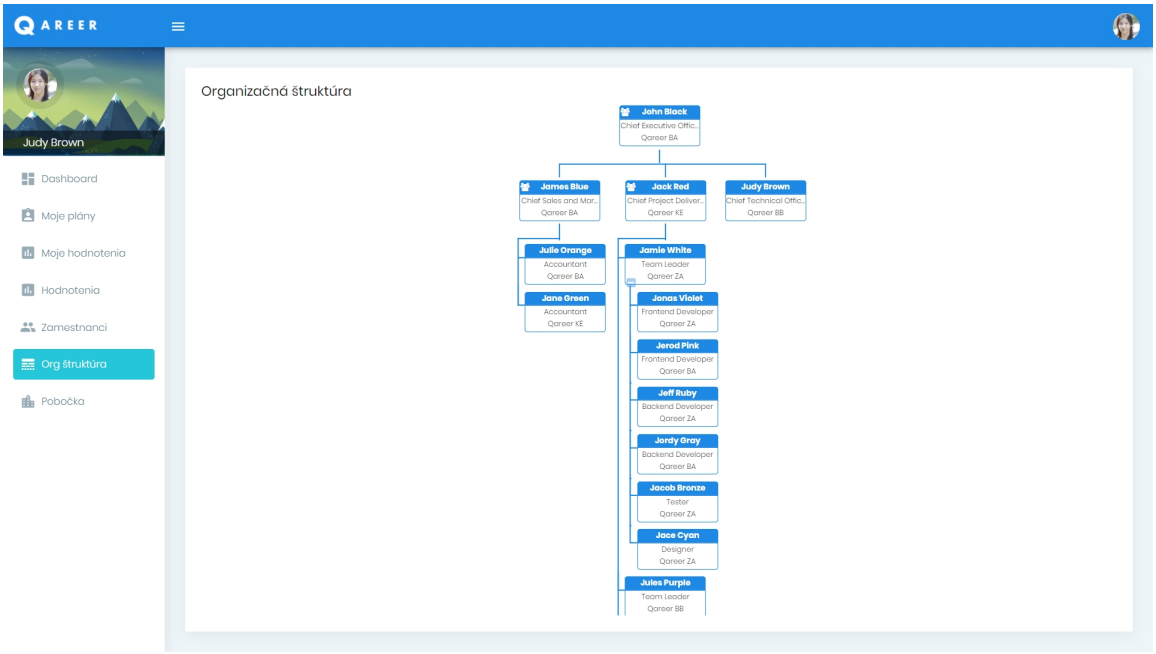
Obr. D.4: Balíčky modulu `qareer-common` a závislosti medzi nimi

Príloha E

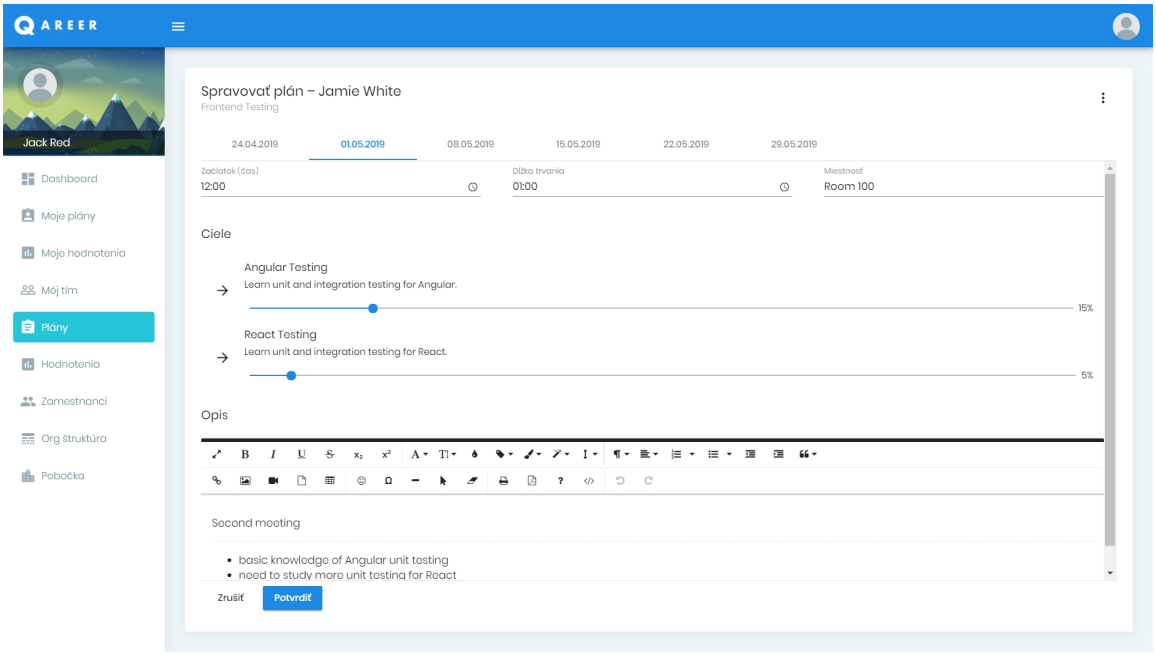
Ukážky z aplikácie



Obr. E.1: Sekcia pre *dashboard*



Obr. E.2: Sekcia pre organizačnú štruktúru



Obr. E.3: Sekcia pre správu plánu

Príloha F

Záverečné testy

Č. testu	Názov testu	Kroky testu	Očakávaný výstup	Skutočný výstup	Úspešnosť	Poznámky
1	Prihlásenie	1. Navigujte sa na stránku prihlásenia. 2. Vyplňte správne prihlasovacie meno a heslo. 3. Kliknite na tlačidlo "Prihlásiť sa".	Zobrazí sa stránka prihlásenia. Údaje môžu byť zadané. Používateľ je prihlásený.	Stránka sa korektne zobrazila. Po vyplnení korektných údajov bol používateľ presmerovaný na obrazovku dashboardu.	úspešný úspešný úspešný	
2	Vytvorenie hodnotení	1. Prihláste sa podľa testu 1 ako používateľ s rolou "Manažér". 2. Navigujte sa do sekcie "Hodnotenia". 3. V pravom hornom rohu zoznamu kliknite na tlačidlo pre menu. 4. Vyberte možnosť pridať nové hodnotenie. 5. Vyplňte požadované údaje a vyberte typ hodnotenia. 6. Vyberte príslušných zamestnancov, pre ktorých sa majú hodnotenia vygenerovať. 7. Kliknite na tlačidlo "Potvrdiť".	Používateľ je prihlásený. Zobrazí sa zoznam hodnotení (môže byť prázdny). Zobrazí sa ponuka. Zobrazí sa formulár pre nové hodnotenie. Údaje môžu byť zadané a zobrazí sa možnosť pre výber zamestnancov hodnotenia. Zamestnancov je možné vybrať. Formulár je možné potvrdiť, zobrazí sa zoznam (vytvorených) hodnotení.	Používateľ je prihlásený. V zozname sa nachadza jeden záznam. Ponuka korektne zobrazená. Formulár sa zobrazil. Po vyplnení požadovaných vstupných údajov sa formulár korektne potvrdil. V zozname pribudol nový záznam.	úspešný úspešný úspešný úspešný úspešný úspešný	

Obr. F.1: Ukážka záverečných testov pre zamestnanca so všetkými používateľskými rolami

Č. testu	Názov testu	Kroky testu	Očakávaný výstup	Skutočný výstup	Úspešnosť	Poznámky
3	Vytvorenie plánu	1. Prihláste sa podľa testu 1 ako používateľ s rolou "Vedúci". 2. Navigujte sa do sekcie "Plány" a v pravom hornom rohu vyberte z ponuky požadovaného člena tímu. 3. V pravom hornom rohu zoznamu kliknite na tlačidlo pre menu. 4. Vyberte možnosť pridať nový plán. 5. Vyplňte požadované údaje a zmeňte vedúceho plánu. (nepovinné) 6. Označte možnosť "Opakovať stretnutia" a vyplňte požadované údaje. (nepovinné) 7. Kliknite na tlačidlo "Potvrdiť". 8. V prípade, že ste v kroku 5. nemenili vedúceho, kliknite na tlačidlo "Detail" pri vytvorenom pláne.	Používateľ je prihlásený. Zobrazí sa zoznam plánov vybraného člena tímu (môže byť prázdny). Zobrazí sa ponuka. Zobrazí sa formulár pre nový plán člena tímu. Údaje môžu byť zadané a zobrazí sa výber vedúcich zamestnancov. Zobrazí sa formulár pre generovanie stretnutí a údaje môžu byť vyplnené. Formulár je možné potvrdiť, zobrazí sa zoznam plánov zamestnanca. V prípade zmeny vedúceho v kroku 5. sa v zozname plánov zamestnanca nezobrazí vytvorený plán (zobrazí sa pridelenému vedúcemu). Ak bola v kroku 6. zadaná možnosť generovania stretnutí, zobrazia sa vygenerované stretnutia, inak sa zobrazí prázdny plán (žiadne stretnutia).	Používateľ je prihlásený. Zoznam je prázdny. Formulár sa zobrazil korektne. Po vyplnení požadovaných vstupných údajov sa formulár korektne potvrdil. Zobrazil sa plán pre vsetky zadané stretnutia.	úspešný úspešný úspešný úspešný úspešný úspešný	

Obr. F.2: Ukážka záverečných testov pre zamestnanca so všetkými používateľskými rolami (pokračovanie)