VŠB TECHNICKÁ UNIVERZITA OSTRAVA
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Automatizované testování frontendu moderních webových aplikací

Automated Frontend Testing of Modern Web Applications

Dan Godula

Bakalářská práce

Vedoucí práce: Ing. Jakub Beránek

Ostrava, 2022

VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Dan Godula**

Studijní program: B0613A140014 Informatika

Téma: Automatizované testování frontendu moderních webových aplikací
Automated Frontend Testing of Modern Web Applications

Jazyk vypracování: čeština

Zásady pro vypracování:

Cílem práce je vytvořit netriviální webovou aplikaci a aplikovat na ni moderní způsoby automatizovaného testování frontendu (unit testy, end-to-end testy, BDD testy apod.) pomocí frameworků jako je např. Jest, Selenium, cypress atd. Pro tvorbu aplikace a testů se předpokládá využití moderních frontendových frameworků (např. React, Vue) a jazyka Typescript.

1. Analyzujte existující přístupy pro tvorbu SPA (single-page application) webových aplikací
2. Navrhněte a implementuje webovou aplikaci s komplexním uživatelským rozhraním
3. Analyzujte a vyberte vhodné metody pro testování webových aplikací
4. Vytvořte sadu automatizovaných testů, která bude testovat správné fungování frontendu vytvořené aplikace
5. Vytvořte analýzu kvality otestování aplikace (např. pomocí metriky code coverage)
6. Zdokumentujte kód a k jeho vývoji použijte verzovací systém (např. git)

Seznam doporučené odborné literatury:

[1] Sasha Vodnik: HTML5 and CSS3, Illustrated Complete, Course Technology, 2015, ISBN: 978-1305394049
[2] Accomazzo Anthony, Murray Nathaniel, and Lerner Ari: Fullstack React: The Complete Guide to ReactJS and Friends, Fullstack.io, 2017, ISBN: 978-0-9913446-2-8
[3] Lucas da Costa: Testing JavaScript Applications, Manning, 2021, ISBN: 978-1617297915

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Jakub Beránek**

Datum zadání: 01.09.2021

Datum odevzdání: 30.04.2022

doc. Ing. Petr Gajdoš, Ph.D.
vedoucí katedry

prof. Ing. Jan Platoš, Ph.D.
děkan fakulty

Abstrakt

Tato bakalářská práce se zabývá automatizovaným testováním webových aplikací, které umožňuje jistější proces vývoje aplikací, snižuje šance na výskyt těžce odhalitelných chyb a poskytuje metriky pro evaluaci kvality výsledné aplikace. Práce analyzuje možné metody testování webových aplikací a různá úskalí, která se při tvorbě a používání automatizovaných testů mohou vyskytnout. Součástí práce je také návrh a implementace webové aplikace a různých sad testů, které tuto aplikaci pokrývají, za pomocí moderních technologií. Kvalita implementace vzniklé aplikace a automatizovaných testů je zhodnocena pomocí několika typů evaluačních metrik.

Klíčová slova

automatizované testování; webová aplikace; frontend; backend; pipeline; Continuous Integration

Abstract

This bachelor thesis deals with automated testing of web applications, which allows for more assuring process of application development, lowers the potential risks of presence of difficult to detect bugs and provides diverse metrics for quality evaluation of final application. The main goal of this thesis is to analyze possible methods of testing web applications and various challenges, which may occur during implementation and usage of automated testing. Part of the thesis is also desing and implementation of web application and different test suites, that will cover this application, using modern technologies. The quality of resulting application and automated tests is evaluated through several types of evaluation metrics.

Keywords

automated testing; web application; frontend; backend; pipeline; Continuous Integration

Poděkování

Rád bych poděkoval svému vedoucímu, Ing. Jakubovi Beránkovi, za přátelský přístup, poskytnuté konzultace, a cenné rady, nejen pro tuto bakalářskou práci.

Obsah

Seznam použitých zkratek a symbolů

API	–	Application Programming Interface
BE	–	Back End
CI	–	Continuous Integration
CLI	–	Command Line Interface
CORS	–	Cross Origin Resource Sharing
CRA	–	Create React App
CRUD	–	Create Read Update Delete
CSS	–	Cascading Style Sheets
DOM	–	Document Object Model
E2E	–	End-to-End
FE	–	Front End
HTML	–	Hypertext Markup Language
HTTP	–	Hypertext Transfer Protocol
JS	–	JavaScript
JSX	–	JavaScript Syntax Extenstion nebo JavaScript XML
MVC	–	Model-View-Controller
MVT	–	Model-View-Template
OOP	–	Object Oriented Programming
ORM	–	Object Relational Mapping
REST	–	Representational State Transfer
SEO	–	Search Engine Optimization
SPA	–	Single Page Application
SSR	–	Server Side Rendering
TDD	–	Test Driven Development
URL	–	Uniform Resource Locator
XML	–	Extensible Markup Language

Seznam výpisů zdrojového kódu

Seznam obrázků

Seznam tabulek

Kapitola 1

Úvod

V současné době se webové aplikace stávají stále populárnější a používanější mezi nejrůznějšími obory pro usnadnění či automatizaci práce. Čím více se tyto aplikace vyvíjejí, tím více se stávají schopnější, ale zároveň i komplexnější. Komplexnější a rozsáhlejší systémy jsou obvykle složeny z velkého množství na sobě závislých částí, či navzájem komunikujících komponent, a tak mohou být zdrojem těžce identifikovatelných a složitě odstranitelných chyb, které mohou zbrzdit a prodražit vývoj aplikace. Při vývoji aplikací se však často opomíná testování, které může častokrát zamezit vzniku případných chyb, nebo usnadnit jejich identifikaci a opravu. I když mnoho vývojových týmů může psaní testů, například z důvodu časové náročnosti, ze začátku odradit, v pozdějších fázích vývoje se ukazuje, že právě pokrytí aplikace testy dokáže spoustu času a trápení při nápravě chyb ušetřit.

Aby se tedy těmto problémům alespoň částečně předcházelo, je potřeba aplikaci už od začátku vývoje postupně pokrývat testy, které ověřují správnost naimplementované funkcionality a dodávají vývojářům důvěru a jistotu v dalších fázích vývoje. Testy by měly být rozděleny do více kategorií, tak aby aplikaci pokryly z co možná nejvíce směrů. Kromě samotné funkcionality by se také měla průběžně testovat i rychlost, bezpečnost a použitelnost aplikace. Pro získání dodatečné vrstvy důvěry v aplikaci a taktéž usnadnění a zrychlení procesu vývoje, testování a finálního nasazení aplikace by se během vývoje měla použít automatizovaná integrační pipeline. Ta by měla umožnit automaticky provádět sadu akcí pro ověření kvality implementace, jako je spuštění testů, sestavení aplikace atd. na vzdáleném virtuálním stroji.

Hlavním cílem této bakalářské práce je analyzovat možné způsoby automatizovaného testování, zvolit vhodné kombinace způsobů testování, upozornit na problematiky související s automatizovaným testováním a představit testovací nástroje. Zároveň je také cílem vytvořit komplexní webovou aplikaci na které bude demonstrováno použití dříve zvolených testů a následně ověřit a vyhodnotit kvalitu otestování výsledné aplikace.

Aplikace, jenž bude sloužit pro demonstrování testování, je systém sloužící pro správu zaměstnanců pracovní agentury, jenž zaměstnává velké množství lidí, kterým nejen hledá pracovní příležitosti, ale u mnohých z nich se i stará o plánování směn v dané firmě a mnoho dalšího. Společnost v současné době funguje bez jakéhokoliv interního systému, který by jejím zaměstnancům usnadňoval jejich práci a taktéž umožňoval digitalizaci velkého množství dokumentů, které firma o zaměstnancích musí uchovávat. Z tohoto důvodu jsem navrhl systém, který se bude starat nejen o správu všech zaměstnanců a možnost je přiřazovat do partnerských firem, ale i o usnadnění plánování směn pro tyto zaměstnance v jednotlivých firmách a následné tvorbě a vizualizaci statistik o efektivitě práce jednotlivých zaměstnanců ale i obecné prosperitě firmy jako takové.

Tato bakalářské práce má následující cíle

- Analyzovat existující přístupy pro tvorbu SPA (single-page application) webových aplikací.
- Navrhnout a implementovat webovou aplikaci s komplexním uživatelským rozhraním, která bude vyhovovat všem požadavkům cílového uživatele.
- Analyzovat a vybrat vhodné metody pro testování webových aplikací.
- Vytvořit sady automatizovaných testů, které budou testovat správné fungování front-endu vytvořené aplikace.
- Vytvořit analýzu kvality otestování aplikace několika různými metodikami.

Kapitola 2 se zabývá obecným popisem automatizovaného testování a taktéž seznámením čtenáře se základní problematikou a jednotlivými kategoriemi testů, které se pro testování frontendu používají. Kapitola 3 na ni navazuje popisem již konkrétních technologií použitých pro vývoj a testování samotné aplikace. Dále bude čtenář proveden návrhem aplikace a testů, jenž pro ni byly použity a popisem architektury aplikace v kapitole 4. Tu následně doplní kapitola 5 zabývající se již konkrétní implementací včetně různých úskalí, které se při tvorbě této webové aplikace vyskytly a jejich řešeními. Kvalita implementace aplikace, a především tedy automatizovaných testů bude nakonec evaluována podle různých kritérií v kapitole 6. Poslední kapitola 7 shrnuje dosažené výsledky a nastiňuje další možná vylepšení aplikace.

Kapitola 2

Testování webových aplikací

Tato kapitola se bude zabývat obeznámením s možnými typy automatizovaných testů webových aplikací a jejich popisem, aby bylo možné z nich následně vybrat ty nejvhodnější pro implementaci a použití v této konkrétní aplikaci. Implementace automatizovaných testů a jejich používání však sebou může přinášet různá úskalí a výzvy, které je potřeba překonat. Proto bude následně nejčastější z nich popsány a nabídnuta jejich možná řešení. Nakonec bude čtenář obeznámen s možností zjednodušení a dodatečné automatizace celého vývojového a testovacího procesu v podobě Continuous Integration a alternativního způsobu vývoje řízeného testy.

2.1 Typy testů

Při testování front-endu u webových aplikací existuje mnoho způsobů a druhů testů, přičemž každý z nich pojímá testování trochu jiným způsobem a má jiné cíle. V ideálním případě by se měl vývojář snažit pokrýt svou aplikaci širším spektrem testů, zaměřených na rozdílné aspekty aplikace a zajistit, že se aplikace bude vždy chovat přesně tak, jak se od ní očekává. Samozřejmě není vždy možné z důvodu časových a rozpočtových omezení použít všechny možné typy testů a není to ani potřeba. Mezi hlavní a nejdůležitější kategorie patří [1, 2]:

2.1.1 Statické testy

Statické testy jsou testy, které se na rozdíl od ostatních kategorií nespouštějí, ale probíhají automaticky již během psaní kódu. Mezi statické testování patří typová, či syntaktická kontrola kódu. Typovou kontrolu poskytuje typový jazyk (v tomto případě TypeScript) a syntaktickou Linter (ESLint). Statické testy umožňují již v průběhu psaní kódu odhalit chyby jako nesprávné typy proměnných, nepoužitý kód, proměnné s nedefinovanými hodnotami, či špatnou syntaxi. Tím snižují pravděpodobnost vzniku chyb [3].

2.1.2 Unit testy

Základem veškerého testování jsou unit testy. Jedná se o nejzákladnější (atomický) typ testu [3]. Jejich cílem je v izolovaném prostředí otestovat jednotlivé komponenty a všechny situace, které se v těchto komponentách mohou naskytnout. Umožňují odhalit a ošetřit všechny možnosti chování komponenty, už na té nejzákladnější úrovni a díky tomu se z nich mohou daleko snadněji skládat větší logické celky, bez nutnosti obav, že komponenta narazí na stav, který neočekávala. Zároveň unit testy pomáhají psaní komponent tak, aby byly více samostatné a nezávislé. Jsou používány především pro validaci vstupů a na nich závislých výstupů, výpočtů nebo např. získávání dat z backendu pomocí API volání. Jednoduchý příklad unit testu si lze prohlédnout na výpisu 1.

```
1 test('Gets correctly formatted date', async () => {
2     const store = new CalendarStore(new RootStore());
3     store.setSelectedDate(moment('2022-01-15 22:45'));
4     expect(typeof store.formattedDate).toBe('string');
5     expect(store.formattedDate).toEqual('sobota 15. leden 2022');
6 });
```

Výpis kódu 1: Příklad unit testu pro kontrolu formátování data

Výhodou Unit testů je především to, že jsou obvykle jednodušší a méně časově náročné na vytvoření a údržbu než Integrační, či End-to-end testy. Zároveň také díky testování v naprosté izolaci poskytují definitivní informaci o tom, zda jim příslušící kód odpovídá požadavkům a správně funguje nezávisle na okolí, čímž dodávají větší jistotu při následném začleňování daného kódu do aplikace a psaní navazujících (Integračních, či End-to-end) testů.

Největší výhoda unit testů je zároveň také jejich slabinou. Unit testy totiž ověřují jak se kód chová v předem definovaných situacích, což však vždy neodpovídá tomu, jak aplikaci skutečně používají uživatelé. Zároveň kvůli ignorování závislostí a integrace s ostatními komponentami, nejsou unit testy schopné poskytnout pohled na to jak se chování testované komponenty, či funkce změní v případě změny nějaké ze závislostí. Z tohoto důvodu je vhodné unit testy doplňovat alespoň ještě integračními testy.

2.1.3 Integrační testy

I když unit testy dopadnou úspěšně, bez doplnění o integrační testy nemají dostatečnou výpovědní hodnotu o celkovém chování dané části aplikace. Pro to je ještě potřeba ověřit, zda fungují správně po integraci s dalšími komponentami. Zde přichází na řadu Integrační testy, které jsou cílené na testovaní skupin (logických celků) kódu (komponent) [4]. To dělá tuto skupinu testů extrémně důležitou, jelikož komponenty webové aplikace jsou často složené z velkého množství menších komponent, které spolu musí být provázány a fungovat společně. Tyto testy jsou už náročnější než unit testy, protože musí ověřit zda interakce mezi jednotlivými komponentami probíhá správně. Příklad integračního testu je možné vidět na výpisu 9.

Výhodou Integračních testů oproti Unit testům je fakt, že Integrační testy testují provázanost jednotlivých částí aplikace, tudíž se více blíží skutečnému používání aplikace, kdy spolu musí tyto části (komponenty) komunikovat. Díky tomu poskytují větší jistotu než Unit testy.

Nevýhodou oproti Unit testům může být větší časová náročnost na tvorbu i údržbu těchto testů.

2.1.4 End-to-end testy

E2E testy simulující reálného uživatele používající danou webovou aplikaci. Jejich cílem je „proklikat se“ celou aplikací a ověřit, zda aplikace, jakožto celek, funguje tak, jak se od ní očekává ve všech možných scénářích, které mohou u běžného uživatele nastat. Důležitým faktorem u E2E testů je např. integrita dat. Tyto testy totiž narozdíl od předchozích zmíněných testů nepoužívají mockování, tedy nahrazování závislostí, ale pracuje se zde se skutečnými daty, což zahrnuje komunikaci s back-endem, databází, API, atd. Z čehož vyplývá, že nám umožnují sledovat chování front-endu v situacích, kdy je porušena integrita dat poskytována backendem. Tento druh testů se nestará o samotnou implementaci. Pracuje pouze s tím, co je zobrazeno na straně prohlížeče.

Největší výhodou End-to-end testů je jejich podobnost skutečnému používání aplikace koncovým uživatelem. Díky tomu, že se u nich nepoužívá žádné mockování a pracuje se se skutečným backendem, databází atd., poskytují nejbližší možný pohled na to, jak se bude aplikace chovat ve skutečném produkčním prostředí. Nevýhodou naopak může být jejich komplexnost a časová náročnost ve srovnání např. s Unit, či Integračními testy. Navíc mohou taktéž být výsledky testů v krajních případech znehodnoceny nefunkčností nějakých závislostí (např. backend).

2.1.5 Visual regression testy

Jedna z věcí, které předchozí testy moc dobře neumí, je poznat, zda se na webu nějak změnilo to jak web vypadá. Při vývoji webových aplikací se totiž často stává, že nějakou úpravou se nedopatřením změní vzhled stránky. Např. se někam posune vstup pro přihlašovací údaje, změní se barva tlačítka atd. V ten moment přichází vhod Visual regression testy. Jejich úkolem je sledovat vizuální změny na webu. Toho lze snadno docílit tak, že se pořídí fotografie webu aktuální verze a při provedení změn se vygeneruje fotografie nová, která se následně v softwaru pro detekci rozdílů v obrázcích porovná s tou předchozí. Pokud test objeví rozdíl mezi těmito obrázky, upozorní o tom vývojáře.

Silnou stránkou těchto testů je rychlá a snadná detekce vizuálních změn, či chyb. Naopak nevýhodou je, že není možné spoléhat pouze na tyto testy, jelikož se vůbec nestarají o jakékoliv změny v logice.

Tento typ testů také dokáže poskytnout rychlou formu evaluace. Na každou testovanou stránku, či komponentu je možné pořídit buď HTML, nebo image snapshot. V případě jeho změny dojde ihned k upozornění, že došlo k možné chybě. Pokud je změna žádoucí, dojde k aktualizaci snapshotu. V opačném případě je možné chybu snadněji identifikovat a odstranit. Na základě počtu pořízených

snapshotů a počtu jejich aktualizací, je možné třeba zjistit, které komponenty jsou nejčastěji upravovány.

2.1.6 Accessibility testy

Starají se o zajištění efektivity při práci s webovou aplikací u lidí s nějakými postiženími, či jinými omezeními pohybu, zraku, sluchu a dalších smyslů. Řadíme zde např. optimalizaci barevného schématu aplikace, aby ji byli schopni použít barvoslepí lidé, nepoužívání příliš malého textu (a v případě jeho použití, zajistit, aby se nerozbilo rozložení stránky při zoomování), či podporu pro Screen Reader (zařízení starající se o čtení stránky pro nevidomé). V některých zemích je Accessibility testing nařízen zákonem a v případě, že produkt není možné používat lidmi s nějakými omezeními hrozí dokonce i pokuta.

2.1.7 Compatibility (Cross-Browser) testy

Tyto testy se starají o poskytnutí stejného zážitku při používání webové aplikace v různých internetových prohlížečích, na různých operačních systémech, ale i např. na mobilních zařízeních. V dnešní době existuje spousta prohlížečů, z nichž každý kompiluje stránky trochu jinak a některé funkce či styly, které fungují v jednom prohlížeči, už nemusí fungovat v druhém. To samé platí pro OS, jelikož především ty starší z nich nemusí podporovat všechny moderní API a grafické technologie. V neposlední řade se zde řadí i podpora pro tisk, tedy zajištění správného formátu textu, rozložení stránky a toho, aby se stránka vlezla na tisknutou stránku.

2.1.8 Performance testy

Cílem performance testů je analýza rychlosti webové aplikace (v tomto případě jen front-endové části). Spadá pod ně rychlost načtení webové aplikace v prohlížeči, doba blokování (čekání na dokončení asynchronních operací), responzivita UI (čekání na animace), stabilita, analýza velikosti načítaných souborů, či množství API volání a jejich dopad na celkovou rychlost aplikace. Tyto testy jsou na rozdíl od většiny dříve zmíněných druhů testů obvykle prováděny automaticky programově, jelikož jejich kritéria a postupy jsou předem definovány a univerzální pro všechny aplikace. Jedním z možných nástrojů pro provedení je například Google Lighthouse.

2.1.9 Smoke testy

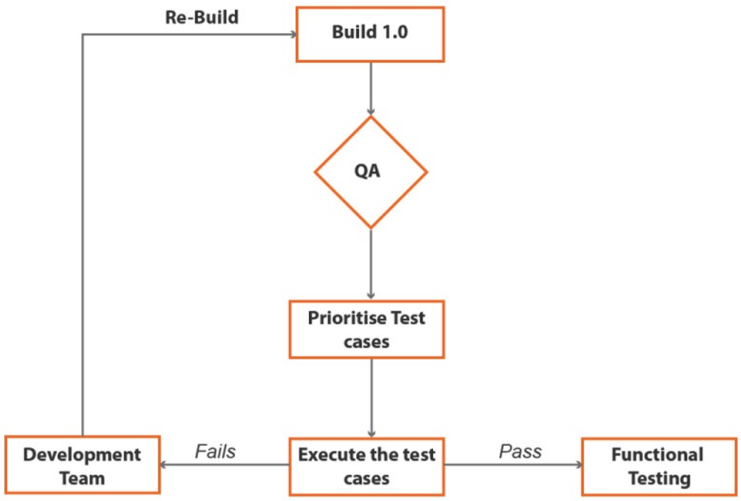
Jedná se o minimální sadu testů, obvykle spuštěnou po dokončení integračního testování, kterými aplikace musí projít, aby mělo smysl další, např. funkcionální testování. Zaměřuje se především na hlavní a nejzákladnější části a funkce aplikace, které se často nemění. I proto se často označuje jako „Confidence Testing“, jelikož se zabývá i tím, zda je „jádro“ aplikace plně stabilní a umožnuje pokračovat s dalším testováním. Smoke testy odhalí většinu chyb v raných fázích vývoje a tím mohou

ušetřit čas strávený při testování dalších funkcí, které by mohly být ovlivněny nefunkčností některé ze základních komponent aplikace. Základní princip smoke testu si lze prohlédnout na obrázku 1.

Např. pro obecný e-shop by mohl jeden Smoke test zahrnovat volbu kategorie zboží a následné zobrazení seznamu zboží odpovídajícího této kategorii, bez zobrazení jakýchkoliv chybových hlášek. Další příklad je například zobrazení přihlašovací stránky, vyplnění korektních údajů a získaní přístupu do uživatelského účtu.

Cílem Smoke testování je tedy odhalit většinu chyb, které buď znemožňují spuštění, či rozšiřování základní funkcionality. Šetří čas vývojovému týmu, který díky nim odhalí spoustu chyb již v ranných fázích testování a nemusí zbytečně pokračovat s rozsáhlejšími testy. Zvyšují důvěru v již existující kód a tím dělají další vývoj snadnější.

Tyto testy by neměly být příliš rozsáhlé, kontrolovat pouze základní funkce, být rychle spustitelné a probíhat na pravidelné bázi a při každém rozšířením aplikace, nebo jejímu nasazení do produkčního prostředí.



Obrázek 1: Princip smoke testu [5]

2.1.10 Security testy

Testují bezpečnost aplikace (především na úrovni aplikační vrstvy) a její odolnost proti hackerským útokům. Kontrolují, zda uživatelská data po celou dobu aplikace zůstávají soukromé a neviditelné pro ostatní uživatele a zda mohou uživatelé provádět pouze ty operace, ke kterým jsou autorizováni. Jednou z nejčastěji používaných metod je Penetration Testing.

Penetration Testing – Penetrační testy spočívají v použití etického profesionálního hackera, či automatizovaného nástroje, který na aplikaci zaútočí a pokusí se tím simulovat skutečný útok, a tak odhalit co nejvíce nedostatků, které potom pomáhá vývojářům odstranit.

2.2 Výzvy automatizovaného testování

Automatizované testování s sebou přináší různé výzvy, které je pro úspěšnou implementaci a dosažení správných výsledků potřeba překonat. V opačném případě by mohly napříč testy vznikat nežádoucí problémy, které by mohly zapříčit nepřesnost, či nespolehlivost těchto testů. Dvě nejčastější výzvy automatizovaného testování budou v následující podkapitole popsány.

2.2.1 Test Flakiness

Jako "flaky" se označuje test, jehož výsledek může být při každém spuštění testu odlišný. Může tedy být jak úspěšný, tak neúspěšný, aniž by se jakýmkoliv způsobem změnil testovaný kód [6]. To znamená, že test, který několikrát za sebou úspěšně projde, může zdánlivě náhodně selhat a naopak. Flaky test může vzbudit dojem, že někde v aplikaci vznikla nová chyba a zbytečně zdržet vývojáře, kteří se budou snažit chybu nalézt. Při vývoji za pomocí automatizovaných testů musí být zajištěno, že taková situace nebude nastávat a testy budou vždy fungovat tak, aby se na ně mohl vývojářský tým při vývoji spolehnout. Hlavní příčinou vzniku flaky testů je nedeterminismus v samotných testech. Při implementaci testů je poté potřeba dát pozor, aby nedeterminismus v testech nevznikal, jinak nemají testy téměř žádnou výpovědní hodnotu.

Zde jsou hlavní příčiny nedeterminismu způsobující flakiness testů a jejich možná řešení [7]:

Čekání (Asynchronous Wait) – Při testování často potřebujeme počkat, než se dokončí nějaká akce, na které je závislá další testovací podmínka, či samotný test (např. při vytvoření nového záznamu o zaměstnanci se musí počkat na kontaktování API, vložení záznamu do tabulky v databázi, vrácení odpovědi o úspěšném vložení, nebo čekání na dokončení animace překrývající testovaný element). Při psaní testu by vývojáře mohlo napadnout spustit příkaz pro uložení záznamu, použít čekání na 15 sekund (ví že tato akce obvykle tak dlouho trvá) a potom provést kontrolu, zda záznam přibyl v tabulce. Pokud by však vkládání záznamu trvalo déle, tak se čekání po 15 sekundách ukončí a test bude očekávat, že v tabulce přibyl záznam, který tam ale není a test tak selže. Nebo se vložení záznamu podaří stihnout za 10 sekund a test bude zbytečně dalších 5 sekund čekat a zdržovat vykonávání dalších operací (např. v případě použití CI).

Možným řešením tohoto problému je čekání na splnění konkrétní podmínky. Test se bude v nějaké dané časové periodě (např. 0,5 sekundy) dotazovat, zda je podmínka splněna, do té doby, než splněna bude, nebo vyprší stanovený počet pokusů. Ukázku čekání na splnění podmínky poskytuje výpis 2.

```
1       cy.waitUntil(() => chart().should('be.visible'))
```

Výpis kódu 2: Ukázka čekání na splnění podmínky

Pořadí vykonávání testů – Problémy při změně pořadí vykonávání testů mohou vzniknout, pokud provádíme nějaké manipulace s daty, nebo úpravy vnitřních, či skrytých stavů aplikace, na kterých jsou závislé další testy. Tudíž pokud jeden test ověřuje vkládání záznamu do tabulky a druhý, jestli se zobrazují záznamy v tabulce korektně a počítá i se záznamem, který vložil předchozí test, tak při změně pořadí těchto dvou testů ten ověřující zobrazení záznamů selže. Testy takto na sobě závislé být nesmí a každý z nich by mělo být možno spustit jak samostatně, tak i s ostatními testy v libovolném pořadí.

Řešením je nikdy mezi testy nesdílet data, či prostředky. Jinými slovy každý test musí pracovat izolovaně ve svém unikátním prostředí. Toho docílíme tak, že před každým průběhem libovolného testu nastavíme „statické“ prostředí testu, které na začátku bude vždy stejné (toho můžeme ideálně docílit např. použitím testovací databáze, která se vždy na počátku naplní fixními hodnotami a po ukončení se smaže). A po doběhnutí testu toto prostředí ideálně resetujeme do původního stavu, v jakém bylo, než byl test spuštěn. V případě že se prostředí mezi testy pouze resetuje nemohou být testy spouštěny paralelně. Proto aby se daly spouštět paralelně je potřeba po dokončení každého testu prostředí zcela zahodit a na začátku dalšího použít nové (načtené).

Přípravu prostředí si lze prohlédnout na výpisu 3. Zde je vidět funkce `beforeEach`, která běží před každým testem a na začátku vymaže tabulku obsahující směny, následně přidá 2 směny se kterými budou testy pracovat, a nakonec navštíví stránku na které se směny zobrazují. Entity zaměstnanců se kterými testy pracují jsou načteny z fixtures.

```
1   beforeEach(() => {
2       cy.waitUntil(() => deleteAllShifts());
3       cy.waitUntil(() => createShift({ date: testDate, time: ShiftTypeEnum.Rano, companyID:
            1, employeeIDs: [34] }));
4       cy.waitUntil(
5           () => createShift({ date: testDate, time: ShiftTypeEnum.Vecer, companyID: 1,
                employeeIDs: [2, 19] }));
6       cy.visit('/company-list');
```

Výpis kódu 3: Ukázka přípravy fixního testovacího prostředí

Souběžnost – Při souběžném vykonávání více testů může docházet k tomu, že data se kterými testy pracují jsou mezi těmito testy sdíleny. Tím může nastat nekonzistence testovacích dat, v případě že například jeden test předpokládá, že hodnota proměnné bude X, ale druhý test ji o pár okamžiků dříve stihne změnit na Y. Nerozhodnost poté způsobuje to, že se akce ovlivňující sdílené prostředky u souběžných testů mohou provést někdy o něco rychleji, či pomaleji. Druhý problém, který zde může nastat je deadlock. Ten vzniká v případě, kdy testy navzájem čekají na změnu nějakého stavu, bez jehož kontroly nemohou dále pokračovat.

Řešením tohoto problému může být použití mockování, resp. použití dat, které jsou pro každý test individuální a nesdílejí se mezi nimi.

Síť / API volání – Pokud to není smyslem testu, nebo naprosto nezbytné, tak by testy neměly komunikovat po síti, či kontaktovat nějaká API. Docházelo by tak k závislosti testu na externích stavech. V případě že dojde k výpadku sítě, či nějakého důvodu nebude fungovat API, nebo bude mít příliš dlouhou odezvu, dojde ke špatnému vyhodnocení testu.

Řešením je nepracovat se samotným API, ale používat mockování.

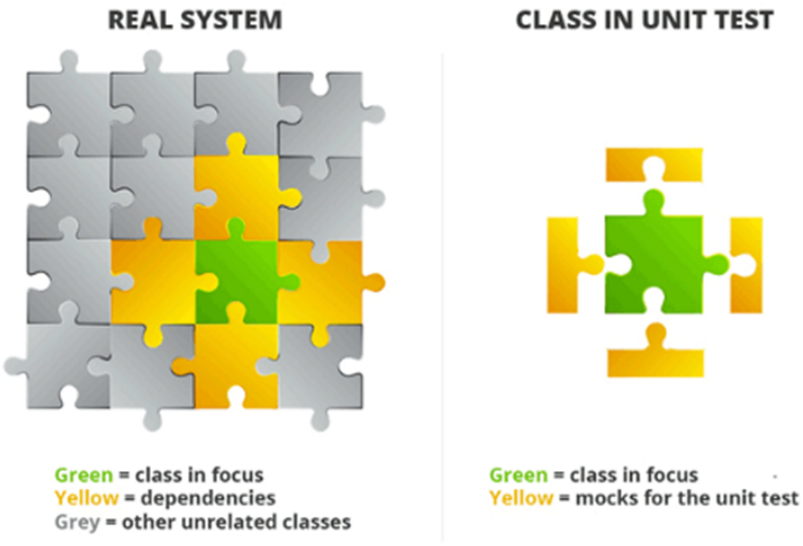
Generování náhodných čísel – Při testování hraje jakákoliv forma náhody, nebo nedeterminismu nežádoucí roli. Proto i generování náhodných čísel často v testech vyvolává flakiness. Jednou totiž může vygenerované číslo nějakou podmínku splnit a příště už ne. Čísla také mohou být mimo podporovaný rozsah a vyvolat další neočekávané chování.

Možným řešením tohoto problému je nepoužívat náhodná čísla vůbec a nahradit je konstantami. V případě že je generování náhodných čísel v testu nezbytné, je při generování náhodných čísel potřeba používat fixní seed, který zajistí, že vygenerovaná sekvence bude při každém průběhu testu stejná.

2.2.2 Mocking

Při testování jednotlivých komponent, či samostatných funkcí obvykle chceme zjistit, zda tyto části kódu fungují správně nezávisle na svém okolí, abychom v ně měli větší jistotu předtím, než je budeme začleňovat do zbytku kódu. Avšak velká spousta komponent, či funkcí je obvykle závislá na nějakých vnějších vlivech, které by v testech mohly způsobit nerozhodnost. Aby nedocházelo k testování závislostí, či znehodnocení testů těmito závislostmi je vhodné použít mockování [8]. Grafické znázornění principu mockování nabízí obrázek 2.

Mock je něco co je uměle vytvořeno a funguje jako náhrada závislostí daného úseku kódu, za účelem izolace testu na pouze ten konkrétní úsek, aniž by byl ovlivněn nějakými vnějšími vlivy. Tudíž simulují chování skutečných objektů, či funkcí. Bez použití mockování by v případě selhání testu nemuselo být jasné, zda selhal testovaný subjekt, nebo jedna z jeho závislostí. To znamená že například nevyužíváme žádné API, čímž snižujeme pravděpodobnost flaky testů, šetříme čas, který bychom strávili čekáním na odpověď a zároveň zbytečně nezatěžujeme API. Mockování také umožňuje jednodušší konfiguraci a reprodukci požadovaných stavů, na základě kterých danou funkcionalitu chceme testovat. Mockováním je také vhodné nahradit závislosti třetích stran, jelikož jsou s největší pravděpodobností už otestovány jejich tvůrci [9]. Používání mockování dává největší smysl při Unit testech, může ale být použito i při Integračních a E2E testech.



Obrázek 2: Grafické znázornění principu mockování [10]

2.3 Continuous Integration

Při vývoji softwaru se obvykle na projektu podílí desítky, či dokonce stovky lidí pracující každý na nějaké své funkcionalitě, či případě na opravách chyb. Každý tento člověk má ze sdíleného repositáře vytvořenou svou větev, do které zpracovává svůj momentální úkol. Po dokončení úkolu je potřeba nově vytvořenou větev s hotovou prací připojit zpátky do výchozí větve. Zde však vzniká problém v tom, že když na projektu pracuje více lidí, kteří také připojují své větve po dokončení zpátky do výchozí, stává se výchozí větev čím dál odlišnější od kopie, ze které vychází větev vývojáře, který se chce zpátky napojit. To znamená že při pokusu o spojení může dojít ke konfliktům v kódu (např. někdo v hlavní větvi odstranil řádek, který vývojář ve své větvi modifikoval a vyžaduje jej), které mohou způsobit nekompatibilitu kódu, jeho dočasné rozbití a tím velké zdržení, či dokonce dočasné zastavení vývoje [11].

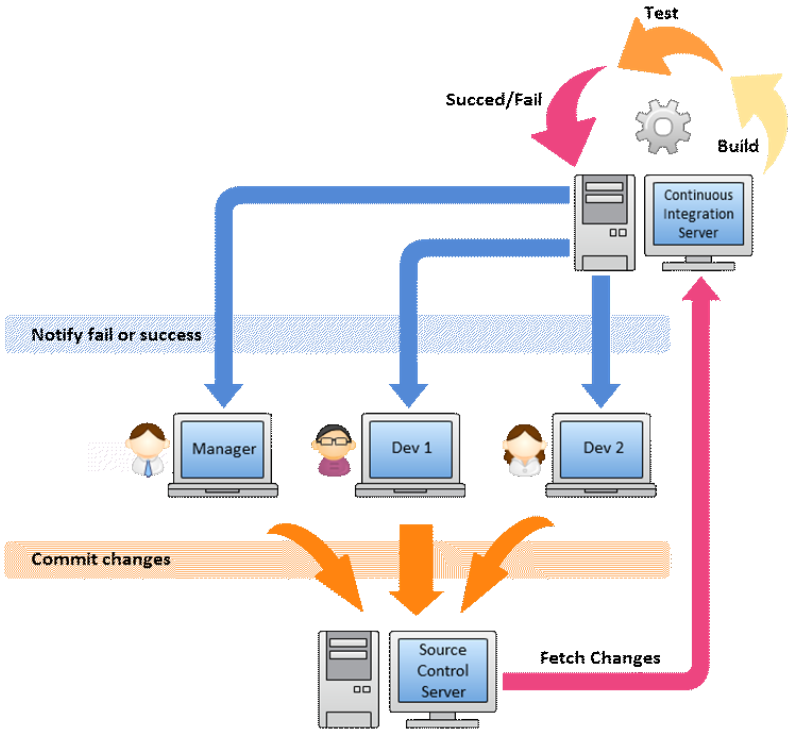
Continuous Integration je metodika, která má za cíl tento problém vyřešit. Vyžaduje použití verzovacího systému (Git) a dedikovaného CI serveru. Řešením je kód pravidelně po malých kouscích pushovat, ideálně na denní bázi a tím docílit větší aktuálnosti kódu, což sníží šanci na konflikt. A v případě že už ke konfliktu i přesto dojde, je tento konflikt daleko menší problém vyřešit, jelikož je množství nového kódu menší, než by tomu bylo bez CI.

Tímto přístupem však vzniká další problém. Jelikož lidé sdílejí svou práci častěji a v tom případě i práci která není hotová, nebo nefunguje zcela správně, tak bude v kódu větší výskyt chyb, nebo aplikace dokonce nepůjde ani zkompilovat.

Zde přichází na řadu automatizace, která se stará o to, aby se nic takového nestalo. Hlavní myšlenkou je při každé nové aktualizaci (commit) provést automatizovanou sadu akcí, které ověří

správnost a funkčnost celé aplikace po nově přidaném kódu. Tyto akce nejčastěji bývají minimálně BUILD a TEST, tedy sestavení produkční verze aplikace a otestování na předdefinované sadě testů. V případě že jedna z části této sady selže, bude zasláno oznámení týmu o tom, že je třeba provést opravy. V případě úspěšného dokončení je možné automaticky spojit větev s úpravami do rodičovské větve, bez jakýchkoliv potíží. Tyto automatizované sady operací běží na dedikovaném serveru, který tyto akce spouští pokaždé ve stejném prostředí, aby zajistil konzistenci výsledků. Způsob fungování CI je možné vidět na obrázku 3.

Hlavními výhodami tohoto přístupu je rychlé odhalení chyb, které jsou zachyceny a řešeny hned při prvním výskytu, čímž se snižuje složitost jejich následné opravy. Automatické testování a kontrola kvality kódu šetří kapacity lidí, kteří by museli provádět testy, či code review. Dále se také zvyšuje aktuálnost větví díky častějšímu přístupu vývojářů k novým změnám v kódu, zvyšuje se přehlednost projektu a umožňuje vydávat nové verze software častěji [12].



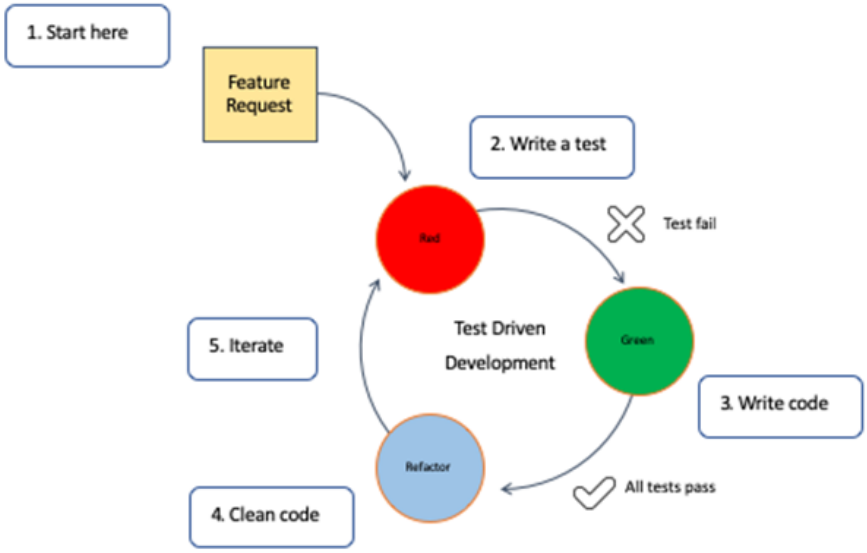
Obrázek 3: Princip fungování Continuous Integration [13]

2.4 Test Driven Development

Test Driven Developement, zkráceně TDD, neboli Vývoj řízený testy je jedna z metod vývoje softwaru, jejíž hlavní myšlenkou je pro každou funkcionalitu nejdříve napsat testy, jenž nadefinují co musí být splněno a jak se má software chovat a až potom příslušný kód, který se bude chovat tak, jak testy očekávají, aby těmito testy prošel. Tento přístup je přesným opakem běžného způsobu

vývoje software, kdy je na základě zákazníkem zadaných požadavků nejprve napsán kód poskytující danou funkcionalitu a až následně testy, jenž ověří správnou funkčnost tohoto kódu. Tento způsob vývoje software vychází z *Agilního přístupu* [14] a *Extreme Programming* [15].

Způsob vývoje je tedy následující. Prvním krokem této metodologie je zpracování specifikace požadované funkcionality od zákazníka a na jejím základě převedení tohoto požadavku na testy, které ověří správnost dané funkcionality. Testy se obvykle rozdělí na malé kousky, které jasně zadávají, co musí být splněno a fungovat (Unit případně Integration testy). Po vytvoření nového testu a jeho následném spuštění se očekává jeho selhání, jelikož funkcionalita, kterou testuje ještě není implementovaná, či je případně nesprávně. Následně je úkolem programátora přidat pouze minimální množství nezbytného kódu, tak aby byly splněny požadavky daného testu. Po úpravě kódu a „uspokojení“ testu se následně provádí refaktor, který by měl odstranit možný přebytečný kód, či jinak optimalizovat danou funkcionalitu, samozřejmě za předpokladu, že tento kód nemění chování funkcionality a stále splňuje (dříve splněný) test. Testy v této fázi dodávají programátorovi jakousi jistotu, protože zajišťují, aby se právě tato funkcionalita nezměnila. Po dokončení tohoto procesu se vývoj vrací zpět ke zpracování dalšího požadavku od zákazníka a celý tento proces se opakuje [16]. Grafické znázornění způsobu vývoje pomocí TDD je možno vidět na obrázku 4.



Obrázek 4: Test Driven Development lifecycle [17]

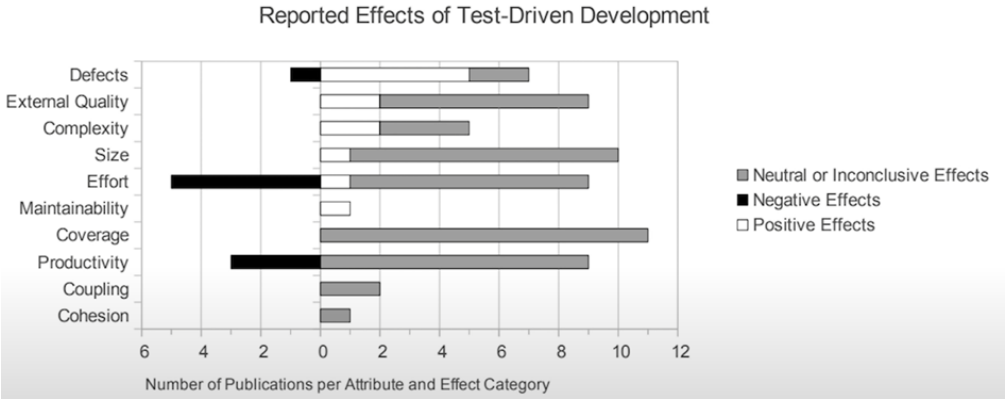
Jaké výhody tato metoda přináší

Na první dojem tento způsob zní zbytečně komplikovaně a zdlouhavě, ale z dlouhodobého pohledu značně zvyšuje kvalitu kódu a snižuje množství nahlašovaných chyb v budoucnu. V první řadě tento princip nutí vývojáře psát modulární kód, který častokrát musí být bez závislostí (jelikož se testuje každý malý kousek kódu), čímž zlepšuje celkovou architekturu aplikace. Díky rozsáhlému pokrytí

kódu testy je zachytávání chyb snadnější. Taktéž je i jednodušší je následně opravit. To usnadňuje i případný refaktor, či rozšíření funkcionality, jelikož testy dodávají programátorovi jistotu, že se při změně kód nic nerozbíjí. Navíc díky průběžnému refaktoru je výsledný kód i přehlednější, což výrazně zlepšuje jeho udržitelnost. Kdyby kód nebyl dostatečně přehledný, tak jako jistá forma dokumentace slouží právě i testy, díky kterým lze snadněji zjistit, co má daný kód dělat, jakým požadavkům vyhovět, a tudíž proč byl napsán tímto způsobem. Finální kód je tedy lépe pokrytý testy, jinými slovy má vyšší code coverage, tvoří si jakousi vlastní dokumentaci, dodává jistotu při případném rozšíření funkcionality a výrazně snižuje počet chyb, které se dostanou mimo vývojové prostředí (k zákazníkovi) [18, 19].

Nevýhody

Široké pokrytí kódu testy obvykle znamená velké množství testů, které musí být udržovány (např. refaktorování), nebo dokonce zcela změněny, v případě že se změní funkcionální požadavky, což může být časově náročné. Testy musí být škálovatelné a pokrývat všechny možné situace, které mohou nastat. Navíc i samotné psaní testů není vůbec jednoduché, právě naopak je to velmi zdlouhavý proces, pokud mají být testy napsány opravdu správně. Pokud nejsou testy dostatečně robustní a nepokrývají všechny možné případy, tento přístup ztrácí smysl. V případě že se pro použití TDD rozhodne, musí se jim řídit celý vývojový tým a musí být striktně dodržován. To může být pro spoustu vývojářů těžké (např. byli celý život zvyklí to dělat naopak) a tak jim může zabrat spoustu času, než se na tento způsob adaptují. Tato metoda vývoje není příliš vhodná pro malé aplikace, či aplikace, které musí být velmi rychle vyvinuty a nasazeny. Taktéž se velmi těžko zavádí do již rozběhlého, rozsáhlého projektu. Výhody i nevýhody TDD použitého v praxi je možné vidět na obrázku 5.



Obrázek 5: Efekty použití TDD v praxi [20]

Kapitola 3

Použité technologie

Předtím než bylo možné vybrat technologie pro realizaci této webové aplikace (především tedy frontendu), bylo nezbytné nejdříve provést analýzu dostupných řešení. Těmito řešeními se myslí nejen použité knihovny, ale hlavně UI framework.

Před volbou UI frameworku jsem analyzoval a zvažoval tři hlavní možnosti, které se mi zdály zajímavé a vhodné: React, Angular a Svelte.

Hlavním faktorem při finální volbě však byla samotná testovatelnost frontendu.
Jednou z výhod Angularu oproti Reactu je fakt, že pro testování celé aplikace, od unit testů po end-to-end, potřebuje méně nástrojů než React a navíc má všechny tyto nástroje již standardně přibalené. Co se týče samotné testovatelnosti komponent jsou na tom React i Angular velmi podobně. Malou výhodou Reactu jsou funkcionální komponenty, které umožňují snadnější unit testování. Svelte narozdíl od Reactu a Angularu zatím nenabízí žádné oficiální nástroje pro testování. Většinu kódu lze otestovat libovolnou testovací knihovnou pro JavaScript, avšak komponenty vyžadují specifičtější přístup. Jediná momentálně existující knihovna pro testování Svelte komponent je Svelte Testing Library, která je však stále ve fázi vývoje.

Nakonec byl zvolen React především kvůli své popularitě a široké podpoře komunity v podobě dokumentace, ale i nástrojů a rozšíření (knihoven). Důležitým faktorem při výběru byla taktéž má předchozí zkušenost s Reactem a nedostatečná vyspělost Svelte. To totiž v současné době selhává na tom, co je pro tuto bakalářskou práci nejdůležitější. Neposkytuje zatím totiž dostatečnou dokumentaci, ani nástroje pro testování.

3.1 Frontendové technologie

3.1.1 React

React je front-endová knihovna (ne framework) napsaná v JavaScriptu (plně podporující TypeScript) vytvořena Facebookem pro tvorbu (nejen) SPA. Stará se o „V“ v Model-View-Controller architektuře. Architektura Reactu je založena na komponentách, ze kterých je složena celá aplikace. Komponenta může být samotná stránka, tabulka s daty, či jedno samotné tlačítko. Každá komponenta má svou vlastní logiku popsanou v JS. Komponenty jsou navíc mezi sebou propojeny, díky čemuž je mezi nimi možno přenášet data, bez nutnosti ovlivnit DOM. Mezi další benefity používání komponent patří hlavně znovupoužitelnost kódu na různých místech v aplikaci a jeho snadnější testování.

React dále používá deklarativní způsob psaní kódu, což znamená, že programátor se stará o popsání toho, co se má stát a jak má co vypadat, místo toho, jak se to má stát a jak k tomuto výsledku dojít. To například znamená, že React umožnuje manipulovat s DOMem, či zpracovávat jeho Eventy (události), aniž by ze strany programátora došlo k jakékoliv přímé komunikaci s DOMem (přímý opak je např. jQuery, které přistupuje k elementům z DOMu a manipuluje přímo s nimi). Díky deklarativnímu způsobu psaní UI, je kód napsaný v Reactu mnohem přehlednější a jednodušší na pochopení.

JSX — Umožnuje kombinovat logiku a markup. Výsledek velmi připomíná HTML, čímž velmi usnadňuje čitelnost kódu a zjednodušuje celkovou tvorbu komponent.

Podmíněné vykreslování — Další z mnoha výhod použití JSX syntaxe je možnost přímo uvnitř komponenty použít podmínky jako IF a při splnění, či nesplnění podmínky vykreslit jiné komponenty, nebo komponentu nevykreslit vůbec. Například místo `IF` a `ELSE` můžeme použít „?“ a „:“, jak je možné vidět na výpisu 4.

```
1   <div>
2       The user is <b>{isLoggedIn ? 'currently' : 'not'}</b> logged in.
3   </div>
```

Výpis kódu 4: Ukázka podmínečného vykreslování

Komponenta — Komponenta se stará o vykreslení nějakého malého znovupoužitelného kusu HTML. Může být zanořena uvnitř další komponenty. Každá React aplikace je složena za pomocí komponent. Každá komponenta pak může mít svůj interní stav (state) a props. Díky separaci kódu do více modulárních částí, umožňují jednodušší testování.

Props — Atributy přiřazené komponentě, mohou být proměnné, funkce, či jiné komponenty. Jsou neměnné a nastaveny rodičovskou komponentou. Na výpisu 5 lze vidět předávání props `type`, `size`, `icon` a dalších, do komponenty `Button`.

```
<Button
    type="primary"
    size="large"
    icon={<PlusOutlined />}
    onClick={() => rootStore.companyStore.openToAdd()}
    title={'Přidat firmu'}
/>
```

Výpis kódu 5: Ukázka předávání props komponentě

State — State si můžeme představit jako objekt obsahující všechny pozorovatelné proměnné ovlivňující chování, či obsah komponenty. Stav komponenty je nepřístupný a neměnný mimo komponentu a může být použit pouze uvnitř příslušné komponenty. Zároveň se stav během „života“ komponenty může několikrát změnit. Změna stavu se obvykle provádí Event Listenery. Stav nikdy nemění programátor explicitně, jedinou výjimkou je stav počáteční při vytváření komponenty.

Hooky — Relativní novinka Reactu, která zcela mění způsob, jakým lze pracovat s komponentami, správou jejich interních stavů a Reactem obecně. Do té doby, pokud komponenty využívaly vnitřní stavy, či life-cycle metody, musely být psány ve třídách, které ale nejsou příliš přehledné, a navíc nás nutí pracovat s klíčovým slovem `this` a provádět bindování funkcí. Hooky umožňují psát všechny komponenty jako funkce a zároveň využívat interní stavy a life-cycle metody. Používání React hooků není povinné a lze je integrovat do aplikace, která již používá třídní komponenty.

Virtual DOM — Při aktualizaci stavu většinou dochází i k aktualizaci ovlivněných komponent. Zde přichází na řadu Virtual DOM. Ten umožňuje vytvořit a uložit kopii skutečného DOMu a při vyvolání nějaké akce porovnat DOM s Virtual DOMem a aktualizovat tak jen ty části webu (komponenty), které se nějakým způsobem změnily (na rozdíl od většiny ostatních frameworků, kde by se musela znovu načíst celá stránka). Programátor při psaní kódu toto vůbec neřeší, o práci s Virtual DOMem se stará React. Toto je velká výhoda především u větších a komplexnějších webů, kde tato funkce výrazně urychluje interakci s webem, jelikož aktualizování celého DOMu je velmi zdlouhavá operace.

Router — V případě, že dojde ke změně stránky, je potřeba aby aplikace měla nějaký způsob routování, který se o tyto změny postará. React Router zajišťuje routování na úrovni klienta a umožňuje přepínat „stránky“, bez nutnosti načítat pokaždé celou novou stránku a tím způsobit probliknutí obrazovky a častokrát zdlouhavé čekání. Ve skutečnosti totiž nedochází k přepnutí stránky jako takové, ale pouze záměně vykreslené komponenty a části URL. Zároveň je zachována možnost používat šipky Zpět a Vpřed v prohlížeči jako u běžného webu.

Výhodami jsou rychlost routování (přepínají se a vykreslují pouze komponenty, ne celá stránka) a možnost použití přechodů a animací v průběhu načítání nového pohledu. Nevýhodou je doba prvního načtení, jelikož se musí načíst kompletně celá aplikace, včetně všech rout.

Create React App — Nakonec by bylo vhodné zmínit, jak založit a nastavit nějakou React aplikaci. Přesně o to se stará Create React App. Jedná se o CLI nástroj pro snadné založení React projektu a jeho následné aktualizování. Stará se o nainstalování všech potřebných knihoven a nástrojů potřebných pro tvorbu React aplikací a zajišťuje jejich aktuálnost a plnou kompatibilitu, aniž by se o cokoliv musel starat programátor. To znamená při instalaci provede nastavení webpacku a jeho Dev serveru pro lokální testování aplikace, Babel, který se stará o překládání JSX syntaxe na běžný JS (pro podporu starších prohlížečů), ESLint pro kontrolu a standardizaci formátování kódu skrze celou aplikaci na základě definovaných pravidel a Jest pro unit testování, kterému se bude text věnovat později. CRA nabízí i několik upravených šablon, které mění některá nastavení, či zahrnuté balíčky (např. šablona pro TypeScript). Další součástí CRA jsou skripty, které se po instalaci automaticky vytvoří. [1] Tyto skripty jsou:

start – Tento skript spouští lokální vývojový server, který umí reagovat na změny v kódu a na základě těchto změn automaticky znovu-načíst stránku.

build – Skript pro sestavení produkční verzi aplikace (pomocí webpacku). Ta je (pomocí Babelu) převedena do syntaxe podporované všemi prohlížeči a optimalizována tak, aby minimalizovala velikost souborů a tím i načítací časy výsledné aplikace.

test – Slouží pro spuštění vestavěného Jest test runneru.

eject – Umožňuje „převést“ projekt na více nastavitelnou verzi, umožňující libovolné nastavení webpacku, Babelu a dalších nástrojů. Tento krok je nevratný a některé funkce CRA nemusí nadále fungovat správně.

Výhody

Jedna z největších výhod Reactu, která ho odlišuje od ostatních populárních knihoven (a frameworků) jako je např. Vue, či Angular, je jeho Virtual DOM (který významně zvyšuje rychlost celé aplikace) a znovupoužitelné komponenty usnadňující a urychlující celkový proces vývoje. Mezi další velké výhody patří obrovská komunita, která vytváří další návazné knihovny, návody, články dokumentující nové aktualizace atd. React je také relativně snadný na naučení (základů pro vytvoření jednoduché a funkční aplikace) a používání. V neposlední řadě je třeba zmínit plugin prohlížeč zvaný React Developer Tools, který umožňuje sledování stavu aplikace a komponent, čímž usnadňuje ladění. Díky rozdělení do přehledných komponent je i testování aplikace jednodušší. Příjemným bonusem je zmíněný nástroj CRA, který zjednodušuje a zkracuje čas prvotního nastavení a jeho přednastavené skripty pro nejčastěji používané operace.

[1] https://create-react-app.dev/

Nevýhody

Vzhledem k rostoucí popularitě a komunitě Reactu zažívá knihovna rychlý vývoj. Kvůli rychlému růstu, ale vývojáři Reactu, či rozšiřujících knihoven, nezvládají vždy poskytovat zcela aktuální a bezchybnou dokumentaci. Jak již bylo zmíněno React není kompletní framework, ale pouze „UI“ knihovna a pro vytvoření rozsáhlejší aplikace napojené na backend, tak bude muset vývojář použít další technologie. V poslední řadě by stál za zmínku přetrvávající SEO problém s indexováním stránek, které jsou vykreslovány na straně klienta.

Důvody pro volbu Reactu

Jedním z mnoha důvodů pro volbu právě Reactu je jeho jednoduchost ve srovnání s jinými frameworky. Dále znovupoužitelnost komponent (díky props a další logice), které zvyšují přehlednost kódu a šetří čas při vývoji. Díky své popularitě také existuje nesčetné množství knihoven, které přidávají již hotové komponenty, či usnadňují správu stavů, zpracovávání formulářů, routování atd. Díky své velké popularitě je React aktivně vyvíjen a aktualizován. Velkou výhodou komponent, a především jejich funkcionálnímu zápisu je také jednoduchost automatizovaného testování. To proto že React umožňuje testování v příkazové řádce přes Node.js (např. za pomocí Jestu) bez použití prohlížeče. Navíc když je každá komponenta funkce, tak často vyžaduje menší množství mockování a má při testování dostatečnou izolaci. V neposlední řadě by také stál za zmínku React Native, jelikož do budoucna zvažuji dodělat mobilní aplikaci. Ten umožňuje psát mobilní aplikace v JavaScriptu téměř stejným způsobem jako běžný React a často umožňuje dokonce použít stejné komponenty, či knihovny.

3.1.2 MobX

Správa stavů jednou z největších překážek při práci na frontendu. Dvě největší výzvy při správě stavů jsou zajištění toho, aby nedocházelo k přímé změnám na datech bez předchozího vytvoření referencí a dodržení jednosměrného proudu dat skrz aplikaci. Dodržení těchto dvou principů je v rozsáhlejší aplikaci bez použití pomocné knihovny velmi složitá úloha a porušení těchto principů často vede k nepředvídatelným chybám.

Jednou z takových knihoven je právě knihovna MobX. MobX slouží pro správu stavů a je nezávislá na použité FE knihovně, či frameworku. Správu stavů umožnuje provádět „izolovaně“ mimo UI framework, čímž zajištuje lepší rozdělení kódu, a snadnější testování. Stará se o zprostředkování a synchronizaci dat získaných z BE do UI. Je rozdělen do tzv. stores, které jsou běžné JS třídy, které navíc v konstruktoru volají metodu makeObservable, za pomocí které vytvářejí tzv. Observable State.

Díky této knihovně se celková práce na frontendu velmi zpřehlednila a zjednodušila, a to především díky možnosti přistupovat ke stavu celé aplikace kdekoliv v komponentě, bez nutnosti používání `useState` hooku a předávání stavů mezi komponentami pomocí `props`. Navíc oddělení byznys lo-

giky do storů mimo komponenty umožňuje jednodušší (unit) testování metod pro práci se stavem (MobX akcí). Příkladem testu metody z MobX store je například metoda pro formátování data, dostupná na výpisu 1.

Velkou výhodou MobXu je také jeho automatické sledování změn stavu použitého v komponentě, které při aktualizaci stavu umožňuje překreslovat jen ty nezbytné komponenty, či jejich části. [2]

Důvod pro volbu MobXu

React samotný sice nabízí způsoby, jak stav aplikace spravovat, většina z nich ale vyžaduje předávání jako props do všech potřebných komponent a jsou pevně svázány s komponentou (součástí stejného kódu), čímž komplikují testování.

Důvod, proč jsem se rozhodl použít MobX je především jeho jednoduchost (spousta věcí se děje tzv. pod pokličkou a programátor se o ně nemusí starat, pokud nechce) a možnost správu stavů přehledně rozdělit do několika různých storů, ke kterým je možno přistupovat a aktualizovat je kdekoliv v aplikaci.

Dalším důvodem je fakt, že MobX má na rozdíl od většiny state-management knihoven OOP syntaxi, která mi byla bližší a působila přehledněji. MobX také díky své schopnosti sledování komponent a podmíněnému překreslování, svou rychlostí překonává nativní řešení Reactu i většinu ostatních knihoven.

3.1.3 Ant Design

Ant Design je open-source designová knihovna napsaná v jazyce TypeScript a byla vytvořena společnostmi Alibaba, Baidu, Tencent apod., jako konkurence pro Material UI. Ant Design byl vytvořen přímo pro React a má široký rozsah nabízených komponent s rozsáhlými možnostmi přizpůsobení pomocí props. Je výborně zdokumentovaný a dostává pravidelné aktualizace od komunity. Nabízí lokalizaci pro více než 30 jazyků a snadno přizpůsobitelné barevné schéma. [3]

Důvod pro volbu Ant Designu

Tato knihovna byla vybrána z několika důvodů. Jedním z nich byl vzhled, který mi přišel pro danou aplikaci vhodnější, než Material UI a další knihovny, které jsem našel. Dalšími důvody byla výborná dokumentace, široký výběr komponent a možnosti různých lokalizací.

3.1.4 Formik

React jako takový neumožnuje nenabízí zcela kompletní řešení pro uživatelské formuláře, ale umožňuje kontrolovat uživatelské vstupy a ukládat stavy vstupů do stavu komponenty, či aplikace. Bez použití jakéhokoliv frameworku, či knihovny pro formuláře, musí řešit provádění validací, změnu

[2] https://mobx.js.org/README.html
[3] https://ant.design/docs/react/introduce

stavu vstupů, odesílání formuláře řešit od základu programátor. Formik se dokáže právě o věci jako je získávání stavu formuláře a jeho aktualizování, řešení validace vstupů a zobrazování chybových hlášek a v neposlední řadě odesílání formuláře. Nabízí více způsobů použití, a to buď za použití komponent, či hooků. Formik je podle velikosti balíčku velmi malá knihovna s jednoduchým a minimalistickým API, která nemá prakticky žádný negativní vliv na výkon aplikace a je rychlejší než většina konkurenčních knihoven. Dále má taky vestavěnou podporu pro validační knihovnu Yup. Pomocí Yup knihovny lze vytvořit validační schema, které umožní pro každý vstup formuláře nastavit datový typ (z TypeScriptu), minimální, maximální hodnotu, zda je vstup povinný a další. Pro každý typ validace, pak umožnuje definovat chybovou hlášku, která se přenese do kontextu Formiku, pokud validace skončí neúspěchem. [4]

Důvod pro volbu Formiku

Formik jsem se rozhodl použít pro jednodušší tvorbu znovupoužitelných formulářů, kde se nemusím starat o udržování stavů formuláře a jejich aktualizaci. Taky jsem využil jeho schopnosti validace vstupu, spravování chybových hlášek formuláře, za pomocí Yup knihovny a jeho nezávislost na použité designové knihovně (komponentách) a tudíž široké kompatibilitě.

3.1.5 Recharts

JavaScriptová knihovna pro tvorbu grafů vizualizaci dat, vyvinuta konkrétně pro použití s Reactem. Je samotná postavena na Reactu a D3js, avšak je kompletně otypovaná v TypeScriptu. Knihovna pro použití nabízí 11 typů grafů. Tyto grafy jsou však kompletně složeny z různě upravitelných, či volitelných komponent. [5]

Mezi hlavní výhody této knihovny patří deklarativní přístup tvorby grafů, složení grafů z několika nezávislých, modifikovatelných komponent, přehledná dokumentace a široká popularita a možnost úprav vzhledu.

I když tato knihovna umožňuje upravovat vzhled jednotlivých grafů a jejich komponent, není to tak jednoduché jako u konkurenčních knihoven. Animace grafů však již upravovat nelze. Grafy taktéž nejsou automaticky responzivní a musí být obaleny do dodatečné speciální komponenty.

3.2 Testovací nástroje

3.2.1 Jest

Jest je populární JavaScript framework vyvinut společností Facebook (původně) pro testování React aplikací a jeho komponent, ale podporuje i další populární frameworky. Lze použít pro testování jak front-endové, tak back-endové části aplikace napsané v JavaScriptu. Jeden z mnoha důvodů,

[4] https://formik.org/docs/overview
[5] https://recharts.org/en-US/

proč je Jest téměř pokaždé na prvních příčkách jakéhokoliv srovnání testovacích frameworků je jeho úplnost (kompletnost). U frameworků jako je např. Mocha, nebo Enzyme jsou často vyžadovány dodatečné balíčky, které už má Jest častokrát zabudované. Podporuje TypeScript díky `ts-jest` balíčku NPM. Obsahuje i zabudované CLI, které umožňuje pouštět testy dle různých filtrů, např. jen ty testy které selhaly, pouze pro soubory, ve kterých došlo od posledního testu k nějaké změně, atd. Testy se také automaticky spouštějí po změně v testovacích souborech, pokud je Jest spuštěn s `-watch` parametrem.

Další velkou výhodou Jestu, především pro méně zkušené uživatele, je jeho jednoduchost nastavení a prvního spuštění oproti ostatním frameworkům.

Jest si zakládá na izolovanosti testů, což znamená že testy běží paralelně, každý ve svém vlastním procesu a nemohou se tak navzájem ovlivnit.

Umožnuje snapshot testování, tj. generuje HTML kód vykreslené komponenty, který následně porovnává s dalšími verzemi a kontroluje, zda nedošlo k nějaké změně. Tím značně usnadňuje testování UI.

Výborně vizualizuje, kde přesně k testu došlo k chybě. Zobrazí řádek a přesné místo na řádku, kde k chybě došlo. Např. při porovnání objektů, či jiných složitější datových struktur přehledně zobrazí rozdíly mezi poskytnutým objektem a tím, jenž se očekával.

Jest je několikanásobně rychlejší než druhý nejpopulárnější framework Mocha a to především díky paralelnímu spouštění testů a spouštěním nejpomalejších testů jako prvních v pořadí. [6]

3.2.2 React Testing Library

React Testing Library je malá testovací knihovna pro podporu při testování React komponent, která se na rozdíl od jiných populárních frameworků , jako např. Enzyme, soustředí na testování toho co skutečně vidí koncový uživatel, tj. DOMu a toho v jakém stavu se DOM momentálně nachází. Zatímco Enzyme kontroluje interní stavy React komponent, což podporuje psaní testů ne úplně korektním způsobem. Snaží se co nejméně zabývat implementací samotných komponent, ale naopak tím, jak provádí interakci s aplikací sám uživatel, a i díky tomu zajistit, že i při refaktoru komponent, budou testy stále správně fungovat dále. Tím vytváří větší stabilitu aplikace a šetří čas při jinak nutném refaktoru komponent, či testů, v případě že se změní implementace. Přistupuje k DOM elementům na webu pomocí rolí, aria parametrů, textu, nebo TestID (`data-testid`), které lze přiřadit každému elementu.

React Testing Library není test runner a je doporučené ho používat v kombinaci s Jestem, avšak je kompatibilní s mnoha dalšími frameworky. [7]

[6] https://jestjs.io/

[7] https://testing-library.com/docs/react-testing-library/intro/

3.2.3 Cypress

Cypress je E2E (který ale umožňuje i Integrační a Unit testování) testovací framework založený na Mocha a NodeJS, s podporou TypeScriptu. Jeho funkce pro manipulaci s DOMem z velké části vychází z jQuery. Běží na většině moderních prohlížečů včetně Google Chrome, Firefoxu, Edge nebo Safari. Má svůj vlastní test runner a simuluje skutečný běh a používání aplikace, přesně v takovém smyslu jako by s ním pracoval skutečný uživatel. Mezi jeho dvě specifické vlastnosti patří automatické čekání na dokončení asynchronních operací, čímž z velké části eliminuje flakiness testů a dále automatické spouštění testů po úpravě souboru (podobně jako u Jestu).

Při provádění testů postupuje vždy krok za krokem, což znamená že po akci, vyčká na provedení nějakého ověření a až potom pokračuje s prováděním dalších akcí. Při provádění těchto akcí pořizuje průběžně snímky obrazovky a snadno čitelné chybové zprávy do konzole prohlížeče, čímž umožňuje jednodušší ladění, jelikož uživatel může přesně vidět, co se v jakém kroku na obrazovce stalo.

Díky tomu že skutečně na rozdíl od většiny ostatních frameworků jako např. Selenium, Cypress běží v reálném čase v prohlížeči a má tak přístup k veškerému API prohlížeče, může například zjistit polohu, stav baterie, nebo vytvářet HTTP dotazy.

Má svou vlastní „galerii“ pluginů, které už tak obsáhlý framework rozšiřují o další užitečné funkce jako např. pořizování a porovnávání obrázkových snapshotů, psaní testů za pomocí Cucumber syntaxe a mnoho dalších.

3.2.4 Github Actions

Github Actions je cloudová CI platforma, běžící na virtuálních strojích Githubu, umožň ující vytvářet, spouštět a jednoduše integrovat CI pipeline do repositáře na Githubu. CI pipeline je schopna se automaticky spouštět sama v reakci na různé akce v repositáři jako jsou například `push` či `pull`. Po dokončení akce je uživateli prezentován přehledný log a taktéž možnost v případně úspěchu automaticky připojit vývojovou větev do hlavní. Na Githubu je možné nalézt již vytvořené šablony pro většinu nejpoužívanějších frameworků, nebo si může uživatel v jazyce YAML vytvořit vlastní pipeline. Github Actions jsou nabízeny zcela zdarma, v případě že repositář nepřekročí 2000 minut běhu pipeline měsíčně. [8]

3.3 Backendové technologie

3.3.1 Django

Django je Python framework založený na MVT (Model-View-Template) architektuře určený pro web development. Obsahuje vlastní webserver, ORM podporující MySQL, PostgreSQL, SQLite a další, vestavěný autentifikační systém a ochranu proti nejčastějším typům útoků. Má výbornou

[8] https://docs.github.com/en/actions

dokumentaci a je relativně jednoduchý na pochopení. Poskytuje přehledný Admin panel pomocí kterého lze provádět CRUD operace. Díky své bezpečnosti, výbornému ORM, přehlednému Admin panelu, jednoduchou škálovatelností s minimální nutností nastavování se perfektně hodí jako backend.

Model — Model je funkcionalita Djanga, která za pomocí tříd umožňuje snadno definovat SQL tabulky, jejich atributy a tvořit mezi nimi relace. Jedna tabulka obvykle odpovídá jednomu Modelu (výjimkou mohou být M-N relace) a sloupce v tabulce jsou určeny třídními atributy Modelu. Django díky použití Modelů umožní využívat své ORM, které značně zjednodušuje práci s databází. Modely lze jednoduše přidat do Admin panelu a provádět na nich pak CRUD operace z prostředí webového prohlížeče.

Jednou z výhod používání Modelu je vestavěná validace. Ta automaticky ověří korektnost datového typu a v případě použití Django Templates automaticky nastaví typ inputu ve formuláři. Model také umožnuje jednotlivým polím specifikovat, zda mohou být prázdné, NULL, unikátní, nebo jejich výchozí hodnotu.

Další výhodou je provádění migrací. Django totiž umí rozpoznat změny v Modelu oproti databázové tabulce a na základě toho vytvořit SQL skript pro úpravu tabulky. Vytvoření skriptu se provádí příkazem `makemigrations` a jeho spuštění příkazem `migrate`.

View — Je třeba poznamenat, že Django View z pohledu MVT architektury, není to samé, co View v architektuře MVC. View v Djangu odpovídá z pohledu MVC architekturu spíše Controlleru, než čemukoliv jinému. Těchto pohledů může být spousta a každý z nich může být buď funkce, či třída, do níž vstupuje vždy HTTP dotaz a další volitelné parametry. View potom na výstupu vrací HTTP odpověď, která může obsahovat HTML, XML, JSON, či přesměrování na jinou stránku. Aby měly Views smysl musí být přiřazeny nějaké URL definované v souboru `urls.py`. Tím je zajištěno, že při navštívení dané URL se vyvolá požadavek na příslušný View. Zároveň do URL lze dosadit a řetězit volitelné parametry zmíněny výše. [9]

Výhody

Jednou z výhod Djanga je jeho rozsáhlost, resp. množství balíčků, které obsahuje a tím výrazně usnadňuje práci a šetří čas vývojářům (autentifikační balíček, admin interface, pokročilé ORM a další). Další výhodou je fakt, že je tento framework postaven na Pythonu, což je jeden z nejsnadnějších jazyků na naučení. Zabudované bezpečnostní balíčky otestované a používané velkými společnostmi a velká komunita vývojářů opravující chyby, z něj dělají jeden z nejbezpečnějších frameworků na trhu. S použitím Django REST Frameworku umožnuje snadné vytváření REST API.

[9] https://docs.djangoproject.com/en/3.2/

Nevýhody

Díky své velké rozsáhlosti je nastavení Djanga oproti dalším webovým frameworkům poměrně rozsáhlé a komplikované. Django taktéž na rozdíl od velké části ostatních frameworků není schopno zpracovávat více HTTP požadavků najednou, což může vést ke komplikovanější implementaci.

Důvody pro volbu Djanga

Důvod pro volbu Djanga byl především jeho dobrá dokumentace, velmi aktivní komunita (zajišťující pravidelné aktualizace, opravy chyb) a jeho rozsáhlé backendové schopnosti šetřící čas a usnadňující vývoj. Čas šetří právě díky tomu, že je v něm zahrnuto tolik užitečných balíčků. Díky Django REST Frameworku umožňuje snadno vytvořit přehledně dokumentované REST API. V neposlední řadě Django přidává něco málo na zajímavosti a nabízí alternativní pohled na návrh webové aplikace oproti běžněji používaným technologiím na backendu v kombinaci s Reactem jako je např. Node.js.

3.3.2 Django REST Framework

Django Rest Framework je použit pro komunikaci mezi frontendem a backendem. Jedná se o framework pro tvorbu REST API v Djangu. Pro svou funkčnost využívá Django Views. Z HTTP požadavku se vytahují zaslaná data (např. pokud se jedná o POST metodu), nebo se mohou pomocí ORM vybrat z databáze a následně se tyto data předloží serializéru, který provede převod dat na JSON formát, následnou validaci a v případě úspěšné validace navrácení HTTP odpovědi.

Serializace se provádí proto, že Views nemohou vracet komplexní datové typy jako jsou Django modely. Pro každý model se proto vytvoří příslušný serializér, který se stará o převod instance Modelu do JSON formátu a naopak. V případě potřeby je model možno v serializéru rozšířit o dodatečné sloupce. Tento framework dále nabízí webové GUI pro lepší vizualizaci výsledků API volání a případných chyb, nebo možnost zasílání dotazů přímo z GUI. [10]

Django Rest Framework jsem dále rozšířil o OpenAPI (Swagger), pro zlepšení API dokumentace. Swagger taktéž poskytuje možnost testovat jednotlivé endpointy, a to i včetně autentifikace a ověřit tak např. zda nemá nepřihlášený uživatel přístup k nějakým citlivým datům. Toto rozšíření jsem použil především pro rychlé testování rout, bez nutnosti napojení a vysílání požadavků z frontendu. [11]

[10] https://www.django-rest-framework.org/
[11] https://github.com/axnsan12/drf-yasg/

Kapitola 4

Návrh aplikace

Cílem této práce je vytvoření webové aplikace pro usnadnění práce malé pracovní agentuře a poskytnout ji vizualizovaný náhled na její prosperitu. Druhým důležitým cílem je pro aplikaci vytvořit sadu různých typů automatizovaných testů pro průběžnou kontrolu a validaci jejich nejdůležitějších use-casů.

4.1 Specifikace požadavků

Webová aplikace by měla umožňovat kompletní správu, jak zaměstnanců, tak firem, do kterých lze zaměstnance přiřazovat. Tím se myslí možnost tyto entity libovolně přidávat, upravovat, vyhledávat, či třídit. Pro každého zaměstnance by aplikace měla umožnit vložení příloh v podobě PDF dokumentů, či obrázků zaměstnance, pro zjednodušení identifikace. Pro každou firmu by měla nabídnout kalendář pro zobrazení a plánování směn. Samotné plánování směn by aplikace měla umět řešit formou přesouvání zaměstnanců mezi dvěma tabulkami formou drag&drop. Při plánování by aplikace měla umět vyhodnotit, zda je daného zaměstnance možno přiřadit a v opačném případě to uživateli znemožnit a poskytnout mu vysvětlení. Aplikace by taktéž z dostupných dat měla uživateli zobrazovat průběžné statistiky o obecné prosperitě firmy, či výkonech jednotlivých zaměstnanců. Tyto statistiky by měla taktéž vizualizovat například pomocí grafů.

- Správu firem včetně přidání, editace, smazání, zobrazení a řazení v tabulce, včetně vyhledávání
- Správu zaměstnanců rozšířenou o možnost nahrávání obrázků a dokumentů, rozdělování do firem
- Zobrazení směn pro jednotlivé firmy v podobě kalendáře pro každou firmu, včetně možnosti přidání, editace a mazání
- Plánování směn formou přidávání zaměstnanců na směnu pomocí drag&drop operace

- V rámci plánování směn validovat, zda je možné jednotlivé zaměstnance na směnu přiřadit a případně poskytnout vysvětlení, proč to není možné, a zaměstnance barevně odlišit
- Zobrazení obecných statistik, či pro specifického zaměstnance včetně jejich vizualizace

Aplikace a především její nejčastější use-casy by měly být pokryty a validovány různými druhy a sadami testů. Těmito nejčastějšími use-casy jsou:

Vytvoření nového zaměstnance – Zobrazení formuláře pro vytvoření zaměstnance, vyplnění všech potřebných údajů, včetně vložení obrázku a dokumentu, přidělené do dostupné firmy a následné odeslání formuláře.

Hledání firmy v seznamu – Hledání firmy na základě různých parametrů, s možností nesprávného zadání malého, či velkého písmene, vyfiltrování jednoho či více výsledků

Přidání nové směny do kalendáře – Přesměrování na kalendář po výběru firmy, zvolení požadovaného dne a zobrazení modalu, výběr požadovaného termínu a přesměrování na stránku plánovače

Úprava směny v kalendáři – Zobrazení kalendáře pro požadovanou firmu, zobrazení seznamu existujících směn pro daný den, volba dne, přesměrování na zvolenou existující směnu, či smazání existující směny

Naplánování směny – Přesunutí několika požadovaných zaměstnanců z tabulky nabízených zaměstnanců do tabulky pro směnu, uložení a přesun zpět do kalendáře

Pokus o přiřazení nevalidního zaměstnance – Zobrazení plánovače směn, pokus o přesunutí zaměstnance nesplňujícího podmínky pro přiřazení, znemožnění akce a zobrazení vysvětlení

Zobrazení statistik konkrétního zaměstnance Navštívení hlavní stránky, zobrazení grafu nejlepších zaměstnanců měsíce a kliknutí na jednoho z nich, nebo použití vyhledávání na vrcholu stránky

4.2 Architektura aplikace

Webová aplikace byla rozdělena na dvě nezávislé části: backend a frontend. Ty mezi sebou komunikují výhradně skrz REST API, které zprostředkovává Django Rest Framework.

4.2.1 Propojení backendu a frontendu

Jelikož je Django využíváno čistě jako backend a o frontend se stará React, je potřeba mezi nimi zajistit komunikaci a způsob výměny dat. Tato kombinace technologií navíc není příliš obvyklá a běžná jako jiné kombinace (např. React + Node.js), proto zde budou popsány způsoby propojení těchto dvou částí aplikace. Pro „propojení“ Djanga a Reactu existují dva hlavní způsoby realizace, kterým se bude věnovat tato podkapitola. [21, 22]

1. První z těchto dvou způsobů je použití Django templatů. To znamená vytvoření Django a React projektu v jednom adresáři a spouštění obou „aplikací“ dohromady, tudíž backend i frontend poběží na stejném serveru. Django bude vykreslovat statický obsah vytvořený Reactem pomocí svého šablonového systému. V praxi to obnáší sestavení React aplikace, což vytvoří statické soubory, a následného vložení sestaveného statického obsahu do Django templatů. Pro vytvoření statických souborů je možné použít například webpack, konkrétně tedy `django-webpack-loader`. Django poté bude používat tyto statické soubory (CSS, JS atd.) vytvořené webpackem a pracovat s nimi jako se svými templaty. Django template tedy vytvoří pouze kostru HTML dokumentu, do níž se vykreslí JS obsah. Přes tyto templaty je možné předávat data přímo z backendu na frontend jako JSON, stejně jako by tomu bylo bez Reactu.

   Hlavními výhodami tohoto přístupu je jednoduchá infrastruktura projektu, tudíž není příliš složité existující Django projekt rozšířit o React obsah. Druhou výhodou, navazující na první, je rozšíření možných způsobu „přeposílání“ dat z backendu na frontend skrz templaty. Tento způsob propojení také umožňuje snadnou implementaci SSR.

   Bohužel je ale z mého pohledu tento způsob převážen nevýhodami, a to především komplikovaností provázání obou aplikací a prvotního nastavení celé frontendové části aplikace. Dále je nutno při každé úpravě na frontendu provádět sestavení (což je u rozsáhlejší aplikace časově náročný proces), aby se vytvořily všechny potřebné statické soubory pro Django a změna se mohla projevit. V neposlední řadě dochází ke kombinaci jazyků, technologií, dvou částí aplikace (frontend a backend) a repositář se tak rychle může stát nepřehledným.

2. Druhým způsobem je rozdělení na dvě nezávislé aplikace, frontend (React) a backend (Django), které mohou (ale nemusí) být odděleny do svým vlastních repozitářů. Každá z nich běží na vlastním serveru. Backend je tedy stále tvořen Djangem jako u prvního způsobu, jen se zde již systém templatů. Frontend je naprosto samostatná aplikace (nejčastěji používající client-side rendering) obsahující vlastní routování. Frontend a backend o sobě tedy navzájem nic neví (Django neví, že na druhé straně je React a React zase neví že na druhé straně Django) a

komunikují spolu pouze přes REST API.

Výhodou tohoto způsobu je především vzájemná nezávislost frontendu a backendu, což znamená že např. při pádu serveru backendu může stále v omezeném režimu fungovat frontend (pokud se data získaná z backendu ukládají do mezipaměti prohlížeče). Dále je daleko menší problém změna technologie na backendu, či frontendu. Stačí když komunikace mezi aplikacemi pomocí REST API zůstane stejná. Řeší se zde i problém rozdělení aplikace do více repositářů, čímž se zvyšuje celková přehlednost systému. Jednodušší je také nastavení frontendu. Není zde třeba se zabývat webpackem, ani linkováním templatů a statických souborů mezi aplikacemi. Frontend navíc může běžet na lokálním serveru ve vývojovém módu (a nesestavovat tak celou aplikaci stále dokola, ale pouze změněné části).

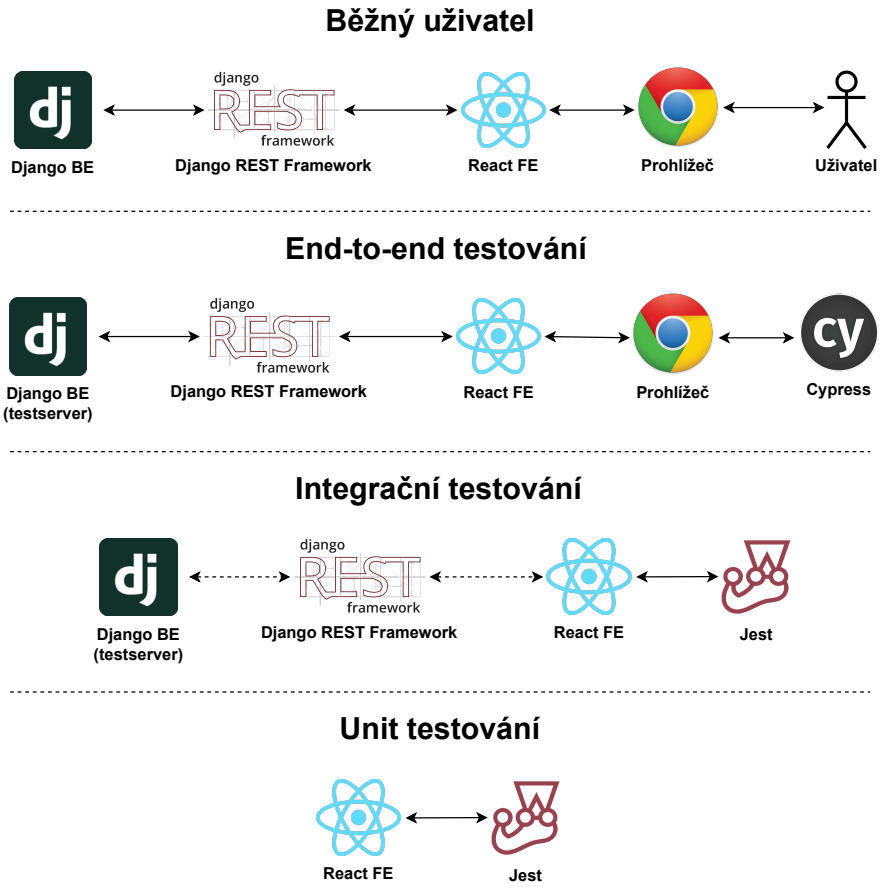
Jednou z možných nevýhod tohoto řešení je nemožnost použití SSR, a tudíž je aplikace odkázána na client-side rendering.

Druhým problémem tohoto řešení je CORS. CORS je mechanismus, který je umístěn v hlavičce HTTP požadavku, umožňující přístup ke zdrojům mimo současnou doménu serveru.

Jelikož aplikace běží na separátních serverech s jinými adresami a porty, tak při příchozím požadavku z frontendu Django zamítne z bezpečnostních důvodů komunikaci, jelikož se pro něj jedná o zdroj neznámého původu. Toto lze vyřešit použitím `django-cors-headers` middlewaru přidávajícího CORS hlavičku do všech HTTP odpovědí a přidáním adresy frontendu na seznam povolených URL.

Při implementaci této aplikace bylo pro komunikaci mezi backendem a frontendem zvoleno a použito druhé zmíněné řešení. Důvody byly především jednodušší nastavení a práce na straně frontendu a větší oddělenost aplikace poskytující lepší přehlednost.

To, jak mezi sebou komunikuje backend a frontend při běžném používání aplikace koncovým uživatelem, je možné vidět na obrázku 6. Na stejném obrázku je taktéž vidět, jak jsou s aplikací propojeny různé druhy použitých způsobů testování a s jakými částmi aplikace komunikují.



Obrázek 6: Diagram propojení aplikace

Kapitola 5

Implementace

Tato kapitola popisuje hlavní prvky (stránky) aplikace, včetně jejich ukázky, krátkého popisu a vysvětlení toho, jak fungují. Taktéž zde bude rozebrána implementace nejdůležitějších částí aplikace, včetně implementace testů, jenž tyto části pokrývají. Aplikace je rozdělena na dvě hlavní části, tedy frontend a backend, které mezi sebou komunikují pouze přes REST API. Aby tedy bylo možné začít popisovat klientskou část aplikace (frontend), je potřeba nejprve i v krátkosti popsat backend, zprovoznění nástrojů pro testování a vytvoření testovacího prostředí.

5.1 Backend

Backend byl realizován pomocí Djanga a Django REST Frameworku. Jako databázový systém bylo zvoleno SQLite. Jedná se rychlou a odlehčenou souborovou databázi, která je nastavena jako výchozí volba v Djangu. Případná změna databázového systému (například na MySQL) může být provedena pouhým přepsáním několika řádků v konfiguračním souboru Djanga provedením migrace dat ze staré databáze do nové.

Pro účely testování se na backendu používá speciální testovací prostředí, které zajišťuje to, že se při každém spuštění sady testů bude pracovat se stejnými daty a ve stejných podmínkách. Byla pro to použita funkce (příkaz) z Django knihovny jménem `testserver` [12]. Ten spouští běžný lokální vývojový server, avšak s tím rozdílem, že při každém spuštění vytvoří novou čistou databázi, do které poté nahraje záznamy vyplněné v libovolném JSON souboru. Po dokončení testů (nezávisle na výsledku) se testovací databáze vždy kompletně celá smaže. Vygenerování testovacích dat bylo řešeno použitím příkazu `dumpdata -format json`, který převede všechny databáze spojené s danou aplikací do formátu JSON a uloží je do souboru. Z těchto záznamů pak byly nepotřebné vyřazeny a ostatní upraveny pro naplnění potřeb testování. Díky tomuto postupu bylo možné jednoduše testovat aplikaci, aniž by docházelo k nechtěným úpravám produkční databáze.

[12] https://docs.djangoproject.com/en/4.0/ref/django-admin/#testserver

5.2 Nastavení testovacích nástrojů

Jelikož samotný React neobsahuje přibalené žádné testovací nástroje, bylo potřeba tyto nástroje dodatečně nainstalovat a nastavit.

5.2.1 Jest

Pokud je aplikace vytvořena pomocí CRA, jako je tato, Jest už je přibalen balíčku knihoven.

Při prvotním spuštění testů zahrnujících komponenty z knihovny Ant Design došlo akorát k jednomu problému. Jest totiž nebyl schopen importovat a pracovat s Ant Design komponentami napsanými ES6 syntaxí. Pro vyřešení tohoto problému bylo nutné v souboru `.babelrc` specifikovat, že z této knihovny bude vždy používat a importovat pouze komponenty napsané pomocí syntaxe ES5, které naštěstí Ant Design stále poskytuje.

5.2.2 Cypress

Jelikož Cypress sám o sobě neumí generovat code coverage, bylo pro generování code coverage potřeba doinstalovat plugin Istanbul [13]. Aby mohl Istanbul code coverage generovat, bylo nutné upravit skript pro spouštění serveru, tak aby bylo umožněno sbírat informace potřebné pro diagnostiku.

Výsledek code coverage se ukládá jako HTML soubor do složky `frontend/coverage/index.html`. Po zobrazení tohoto souboru v prohlížeči se zobrazí interaktivní GUI, umožňující navigaci mezi soubory a složkami a detailního zobrazení zvoleného souboru včetně zvýraznění nepokrytých řádků kódu.

5.3 Nastavení CI pipeline

Pro další automatizaci vývoje, a především testování byla pro tuto aplikaci vytvořena již dříve zmíněná CI pipeline. Pro používání pipeline je v první řadě potřeba zajistit server na kterém tato pipeline poběží. Jelikož je tento projekt již verzován na GitHubu, byla použito řešení CI přímo od GitHubu zvané GitHub Actions.

To, co se má na virtuálním stroji GitHubu v CI pipeline vykonávat je specifikováno pomocí jazyka YAML v souboru `.github/workflows/node.js.yml`. Jako první je zde určeno, při jaké akci, v jaké větvi se pipeline spouští. V případě pipeline tohoto projektu se akce spouští při každém pushi v hlavní větvi (master) a větvích s předponou `feat/`, `fix/` a `test/`. Dalším krokem je volba operačního systému, na kterém poběží virtuální stroj a verze používaných programovacích jazyků, tedy Node.js a Pythonu. Po nastavení jsou v souboru podle pořadí vykonávání definovány jednotlivé kroky. Prvním krokem je nainstalování dříve specifikované verze Node.js. Poté se instalují všechny

[13] https://github.com/cypress-io/code-coverage#readme

potřebné front-endové balíčky definované v souboru `package.json`. Prvním skutečným krokem je poté spuštění unit testů. Každý z těchto kroků je popsán dvěma až třemi body:

- name – název dané akce, který se bude zobrazovat v průběhu a výsledku běhu pipeline
- working-directory – cílový adresář, ve kterém se spouští daná akce (v tomto případě `./frontend`)
- run – samotný příkaz, který se má v tomto kroku spustit (např. `npm run test`)

V případě úspěšného průběhu unit testování se pokračuje na další operaci. V případě neúspěchu jakékoliv akce se pipeline následně automaticky ukončí a další kroky již ignoruje. Další operací je sestavení aplikace (`npm run build`). To ověří, zda se v aplikace nenachází typové chyby, nebo se například v konzoli nezobrazují nějaká varování. Poté co se úspěšně dokončí sestavení se pokračuje na E2E testování. To zahrnuje nainstalování Pythonu. Po nainstalování Pythonu se nainstalují všechny potřebné Python závislosti, včetně Djanga a Django Rest Frameworku. Po nainstalování závislostí se spouští poslední operace. Ta obsahuje hned několik příkazů, spouštěných paralelně. Nejprve se ze složky `./api` spustí příkaz pro spuštění Django serveru s testovací databází a zároveň se ve složce `/frontend` spustí příkaz `npm run ci`. Tento skript pomocí npm balíčku `start-server-and-test` [14] nejprve spustí vývojový server, počká na jeho spuštění (tedy dokud není přístupná adresa localhost:3000), a poté spustí E2E testy pomocí příkazu `test:e2e:ci`. Tento krok je možné vidět na výpisu 6.

Posledním krokem této pipeline je vygenerování reportu nástrojem Google Lighthouse, který je popsán v kapitole 6.

```
1 - name: Start Django server with test DB, frontend and run E2E tests
2   working-directory: ./api
3   run: |
4     python3 manage.py testserver test-data &
5     cd ../frontend
6     npm run ci
```

Výpis kódu 6: Ukázka YAML souboru nastavení CI pipeline

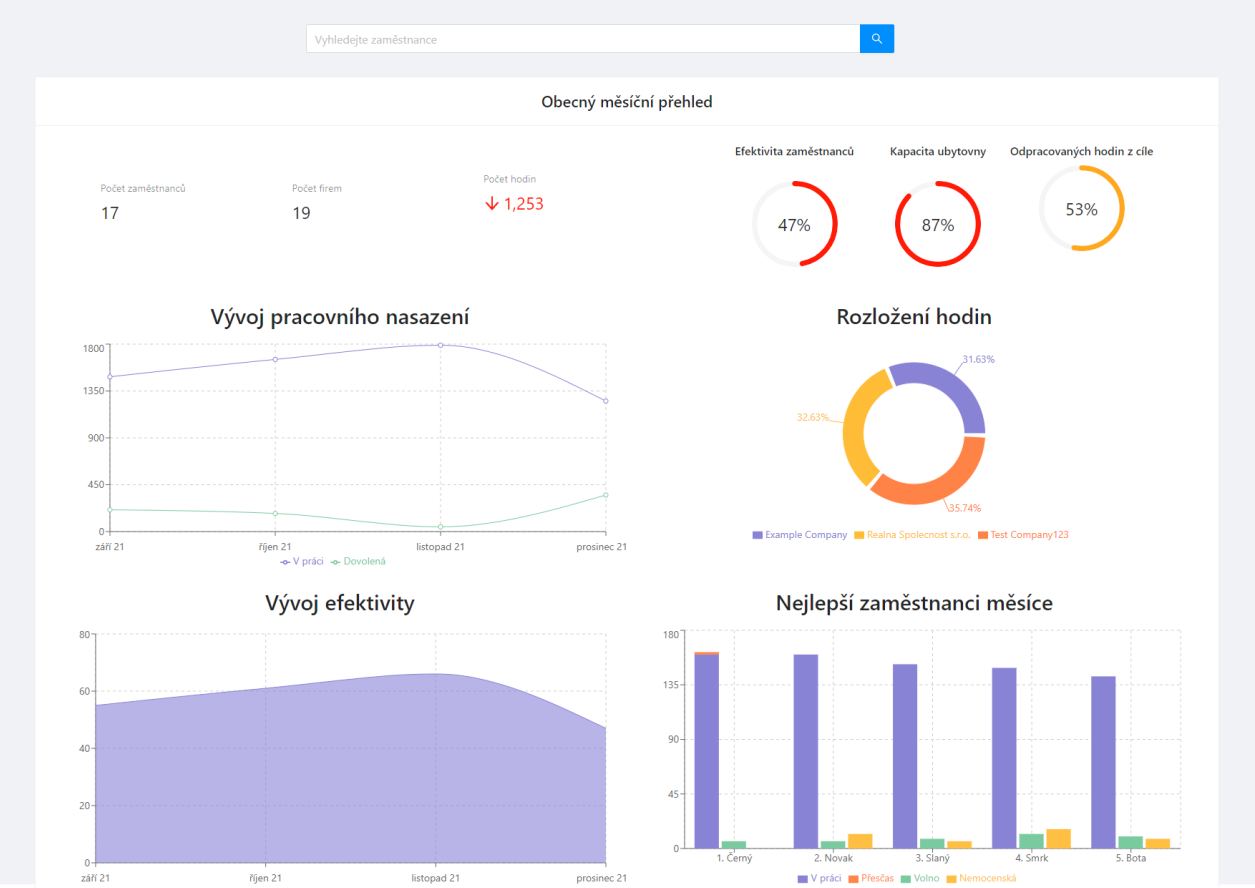
Při implementaci CI v části end-to-end testování docházelo k hlášení chyb o nevalidních (rozdílných) image snapshotech. To bylo zapříčiněno tím, že snapshoty byly na lokálním prostředí pořizovány v GUI režimu a v CI pipeline v headless režimu. Z tohoto důvodu se u obou testů pořizovaly snapshoty jinak (jiné rozlišení a velikost). Problém byl vyřešen tím, že všechny snapshoty se začaly pořizovat, a tedy i porovnávat, pouze v headless režimu, čímž konflikty již nadále nevznikaly.

[14] https://www.npmjs.com/package/start-server-and-test

5.4 Frontend

Po seznámení s backendem je nyní možné se přesunout k popisu implementace frontendu a samotných testů.

5.4.1 Dashboard



Stránka Dashboard slouží jako hlavní stránka, která poskytuje uživateli přehledné shrnutí všech relevantních statistik za poslední měsíc, včetně grafů, zprostředkovávajících vizualizaci těch nejdůležitějších údajů. Grafy zvyšují přehlednost daných statistik a umožnují uživateli lépe vidět historický vývoj statistik. Ve vrchní části stránky, pod hlavní nabídkou může uživatel vidět pole pro vyhledávání. To se na stránce nachází z toho důvodu, že Dashboard umožnuje přepínat mezi pohledem pro obecné měsíční statistiky a pohledem pro konkrétního zaměstnance, kterého je možno nalézt a zvolit právě pomocí tohoto vyhledávácího pole. Po přepnutí do režimu pro konkrétního zaměstnance jsou uživateli prezentovány stejně vypadající grafy jako na obecném přehledu, avšak již s upravenými daty pro konkrétního zaměstnance. Pod vyhledávacím polem se nachází shrnutí hlavních statistik

v textové podobě informujících o počtu zaměstnanců, firem a celkových odpracovaných hodin v daném měsíci. Tyto statistiky na základě zlepšení, či zhoršení oproti minulému měsíci zobrazí malý barevný indikátor. Vpravo od nich se nacházejí malé kulové grafy, které se zbarvují na základě toho, jak moc se současné hodnoty blíží těm ideálním.

V druhé části obrazovky se nachází další, již větší grafy, rozmístěné do tabulky a vytvořené pomocí knihovny Recharts, zobrazující výběr nejdůležitějších statistik.

Jako příklad grafu lze zvolit například komponentu MyBarChart, která na obecném přehledu slouží jako graf pro zobrazení pěti nejlepších zaměstnanců podle výkonu za poslední měsíc, seřazených od nejlepšího po „nejhoršího“. Tato komponenta se skládá z několika menších komponent, reprezentujících například tooltip, popisnou Y a X osu, legendu pod grafem a následně samotnými sloupci. Tyto sloupce při kliknutí umožnují přesměrování na měsíční přehled daného zaměstnance.

Testy

Testování grafů a obecně této stránky bylo poněkud odlišné a složitější oproti ostatním stránkám. Zde bylo potřeba především otestovat vizuální stránku celého dashboardu a konzistenci grafů na základě vstupních dat, namísto testování nějaké funkcionality. Proto byly během testování hojně využity (image) snapshot testy.

E2E

Pro všechny grafy je proveden image snapshot test, ověřující vizuální konzistenci grafu. Jelikož ale při každém načtení stránky grafy provádějí různé načítací animace (např. u spojnicového grafu se postupně vykreslují čáry z leva doprava, koláčový graf se postupně rozevírá...), mohou při tomto druhu testování vznikat problémy s nekonzistencí pořízených obrázků, když se např. nestihne dokončit animace dříve, než je pořízen obrázek, a tím docházet k flaky testům. Řešením tohoto problému bylo již dříve zmíněné čekání na splnění nějaké podmínky a až poté vykonání další akce v kapitole o výzvách při testování. To zde například obnáší vyčkání na ověření, zda je graf z pohledu DOMu viditelný, má požadovanou výšku, očekáváný počet čar, či sloupců (u sloupcového grafu) atd. Až po vyčkání na splnění všech definovaných podmínek se pořídí výsledný snímek, který se dále porovnává s posledně pořízeným.

Pro spojnicový grafu bylo ověřeno, zda se správně zobrazují datumy na X-ové ose, zda graf dodržuje specifikované barevné schéma (barvy jednotlivých čar odpovídají požadovaným HEX kódům), a zda se pod grafem zobrazuje legenda se správnými atributy (počtu odpracovaných hodin a hodin na dovolené).

U koláčového grafu bylo navíc oproti spojnicovém, testováno, zda se zobrazuje správný počet dílů v grafu, podle počtu firem v testovací databázi.

Taktéž je zde testováno, zda se po najetí myší na libovolný dílek správně zobrazí tooltip obsahující název firmy a počet odpracovaných hodin odpovídající testovacím datům.

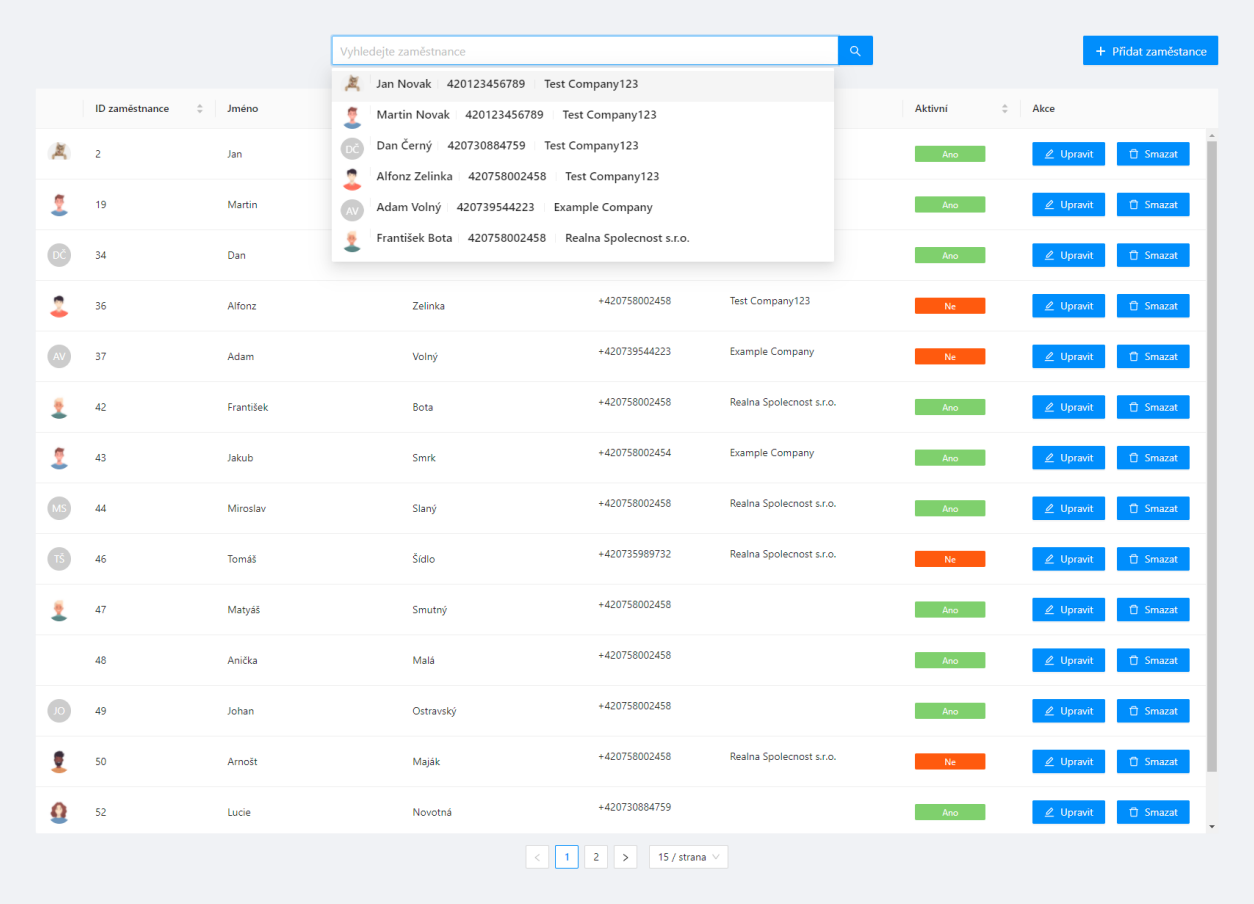
U sloupcového grafu se ověřuje, zda je zobrazen správný počet sloupců a nevykreslují se sloupce pro atributy s nulovými hodnotami.

Dále se testuje, zda se po najetí myší na libovolný graf ze skupiny zobrazí tooltip. U tooltipu se podobně jako u sloupců dodatečně ověřuje, že se nezobrazují sloupce s nulovými hodnotami.

Pro samotný tooltip je proveden ještě image snapshot test.

Poslední test ověřuje, zda se po kliknutí na libovolný sloupec zobrazí měsíční přehled pro zaměstnance, jenž tomuto sloupci náleží. To obnáší kontrolu správného jména, příjmení a názvu firmy po přesměrování na pohled pro zaměstnance.

5.4.2 Seznam zaměstnanců



Na stránce seznam zaměstnanců se nachází jen pár důležitých prvků. Hlavním prvkem (komponentou) je tabulka se zaměstnanci. Záznamy v tabulce lze libovolně řadit pomocí klikání na příslušný nadpis sloupce. Tabulka kromě všech potřebných informací o zaměstnanci obsahuje i avatar zaměstnance a dvě akční tlačítka. Po kliknutí na tlačítko pro smazání se uživateli pro jistotu ještě zobrazí varovné okno, informující o nenávratnosti akce.

V případě kliknutí na tlačítko „Upravit“ z pravé strany obrazovky vyjede modální okno obsahující formulář předvyplněný daty o zaměstnanci.

Nad tabulkou se nachází ještě dvě důležité komponenty. Jednou z nich je stejná komponenta vyhledávácího pole, kterou uživatel mohl zahlédnout již na hlavní stránce. Tato komponenta bude detailněji popsána na stránce s firmami. Druhou komponentou je tlačítko „Přidat nového zaměstnance“, jenž po kliknutí zobrazí stejnou komponentu s formulářem jako tlačítko „Upravit“, avšak s prázdným formulářem.

V neposlední řadě se ve spodní části stránky nachází komponenta, která umožnuje přepínat počet záznamů zobrazených v tabulce na jedné stránce a taktéž se mezi stránkami přesouvat.

Formulář

Formulář pro editaci/přidání zaměstnance je realizován za pomocí Formik knihovny. Ten poskytuje formuláři kontext obsahující současné hodnoty všech vstupů, případné chybové hlášky, údaje o tom, zda je formulář aktuálně validní atd. Validace vstupů se provádí na základě validačního schéma vytvořeného za pomocí knihovny Yup.

Samotné vstupy formuláře jsou poté řešeny jako jednotlivé nezávislé komponenty, které mohou být použity v jakémkoliv formuláři, aby nedocházelo ke kopírovaní kódu.

Formulář pro zaměstnance obsahuje ještě dvě zajímavé komponenty. ImageUpload se (jak již z názvu vyplývá) používá pro nahrání obrázku do formuláře. FileUpload se stará o nahrávání a stahování PDF příloh připojených k záznamu o zaměstnanci. Jelikož Formik neumí nativně pracovat s typy `File`, či `Blob`, sloužící pro reprezentaci souboru, musí se soubor před uložením do formuláře převést na Base64 řetězec. U obrázků a PDF příloh se provádí validace. Jedna z nich při nahrávání kontroluje, zda se jedná o obrázek podporovaného typu, následně se po zpracování ověří, zda se jedná o Base64 řetězec a jestli nebyl obrázek někdy mezi nahráním a odesláním formuláře náhodou odstraněn. Tento nahraný obrázek se zobrazí v komponentě místo původního tlačítka s ikonou pro nahrání.

Podobný postup je použit i pro PDF přílohy, které pro nahrávání využívají sice jinou komponentu, ale pravidla a postup vycházejí z tohoto řešení.

Práci s obrázky a PDF soubory bylo potřeba vyřešit ještě na backendu. Do modelu pro tabulku zaměstnance přibyly dva nové sloupce typu `ImageField` a `FileField`. Tyto pole jsou poté v databázi reprezentovány jako odkazy na media soubory. Dodatečně musí být v nastavení Djanga (settings.py) definovány kořenové složky pro tyto soubory (MEDIA_ROOT) a URL ze kterého bude soubory načítat API (MEDIA_URL). Pro práci s obrázky na backendu byl doinstalován balíček `Pillow`.

Jelikož výměna dat mezi FE a BE prostřednictvím API probíhala pouze formou JSONu, musely být obrázky i PDF soubory převáděny na Base64 textové řetězce. Z těchto zakódovaných textových řetězců však na backendu musely být data poskládány zpátky na soubory. O převod se na backendu stará serializér, konkrétně metody `Base64ImageField` a `Base64FileField` knihovny `django-extra-fields`[15] pro Django REST Framework. V případě ukládání PDF souborů bylo nutné rozšířit vestavěnou metodou vlastní implementací filtrující soubory s příponou `.pdf` a reali-

[15] https://pypi.org/project/django-extra-fields/

zující dekódování Base64 textu na PDF soubor za pomocí balíčku `PyPDF2`.

Testy

Testy realizovány na této stránce jsou zaměřeny především na práci s formulářem. Konkrétně byla ověřována správná validace a případné zobrazování chybových hlášek ve formuláři, nahrávání příloh a obrázku.

Unit/Integrační

Pro formulář byly provedeny Unit testy zaměřující se otestování správné funkčnosti komponent realizujících nejčastěji používané vstupy: TextInput a NumberInput.

U NumberInputu se testuje hlavně to, zda vstup přijímá pouze numerické hodnoty a v případě zadání nenumerické hodnoty se na vstup nic nepřidá.

Taktéž se zde provádí ověření, zda vstupní pole obsahuje požadovaný počet znaků. V případě že po dokončení vyplnění pole uživatelem není podmínka splněna, ověří se, zda se správně zobrazila chybová hláška.

Pro simulování uživatelské interakce s komponentou se používají akce React Testing Library, konkrétně `userEvent` a `fireEvent`[16].

Pro selekci elementů je ideální použít `screen`[17] z React Testing Library, který reprezentuje všechny DOM elementy, které jsou pro uživatele viditelné na obrazovce. Elementy v DOMu jsou poté nejčastěji vyhledávány podle atributu `testid`. Ten totiž umožňuje element vždy nalézt bez nutnosti refaktorovat testy, v případě že by došlo ke změně typu elementu, názvu, či role.

Jelikož jsou všechny uživatelské akce asynchronní operace, provádí čekání se u výběru elementů i ověřování podmínek čekání.

Pro obě zmíněné komponenty se dodatečně provádí ještě HTML snapshot testování.

Dalším bodem je Integrační testování formuláře. Testovaná komponenta je zde EmployeeForm, která však pro svou funkci potřebuje dva parametry: výchozí hodnoty a funkci pro odeslání validního formuláře. Aby byly testy konzistentní a zároveň nebyly nijak ovlivněny závislostmi, musíme tyto dva parametry namockovat. Pro reprezentaci výchozích hodnot byl vytvořen objekt reprezentující zaměstnance. Pro funkci onSubmit vytvoříme pomocí Jestu tzv. mock funkci, kterou lze zavolat, zjistit kolikrát a s jakými parametry byla volána, avšak samotná funkce nic nedělá.

První dva integrační testy formuláře se zabývají pouze tím, zda kompletní komponenta formuláře odpovídá předchozímu DOM snapshotu a zda se při vykreslení správně zobrazí se všemi výchozími hodnotami.

Další test ověřuje funkčnost nahrávání souboru, jak běžným, tak i drag&drop způsobem. Pro test byly použity 2 soubory, jeden obrázek, druhý PDF.

[16] https://testing-library.com/docs/user-event/intro
[17] https://testing-library.com/docs/queries/about/

Po vložení obou souborů a odeslání formuláře se provede kontrola, zda byla funkce zavolána (tzn. ve formuláři se neobjevila chyba) a zda ji jako parametr byl vložen objekt zaměstnance včetně korektně vyplněných atributů pro obrázek a přílohu.

Následující test slouží pro kontrolu validace formuláře. Nasimuluje se vyplnění několika vstupů nevalidními hodnotami a vynechají se povinná pole. Formulář se poté pokusí odeslat a test ověří, zda dojde k zobrazení chybových hlášek u všech nevalidních polí a namockovaná funkce se tím pádem nezavolá. Tento test je možné si prohlédnout na výpisu 9.

Finální test představuje postup pro minimální úspěšné vyplnění formuláře, tak aby byly vyplněny všechny nutné pole a formulář úspěšně volal namockovanou onSubmit funkci.

Zde bylo potřeba pro vyplnění kategorizace práce namockovat i komponentu Select z knihovny Ant Design, jelikož list s možnostmi výběru této komponenty není z pohledu DOMu viditelná komponenta a `screen` hledá pouze komponenty, které viditelné jsou. Vytvořený mock se chová jako běžný HTML `select` element.

E2E

U E2E testů se podobně jako u stránky pro dashboard používají testovací data, což umožňuje otestovat, zda tabulka se zaměstnanci obsahuje správný počet záznamů.

Aby bylo možné testovat práci s formulářem, je nutné, aby se Cypress Test Runner před spuštěním každého z testů nejprve na stránku Seznam zaměstnanců dostal.

Proto se funkce pro navštívení stránky vloží do beforeEach[18] hooku. Tento Cypress hook spustí všechny vložené funkce při spuštění každého testu v dané sadě.

Pro vizuální kontrolu se jeden z testů stará o pořízení image snapshotu komponenty. Pořizovat snapshot celé stránky není potřeba, navíc se díky tomu snižuje šance na to, že testy skončí neúspěšně, i když se samotná komponenta formuláře nezmění.

Pro testování nahrávání souborů se zde na rozdíl od integračních testů již používá skutečná Drag&Drop akce, bez jakéhokoliv mockování, tak jako by to dělal skutečný uživatel.

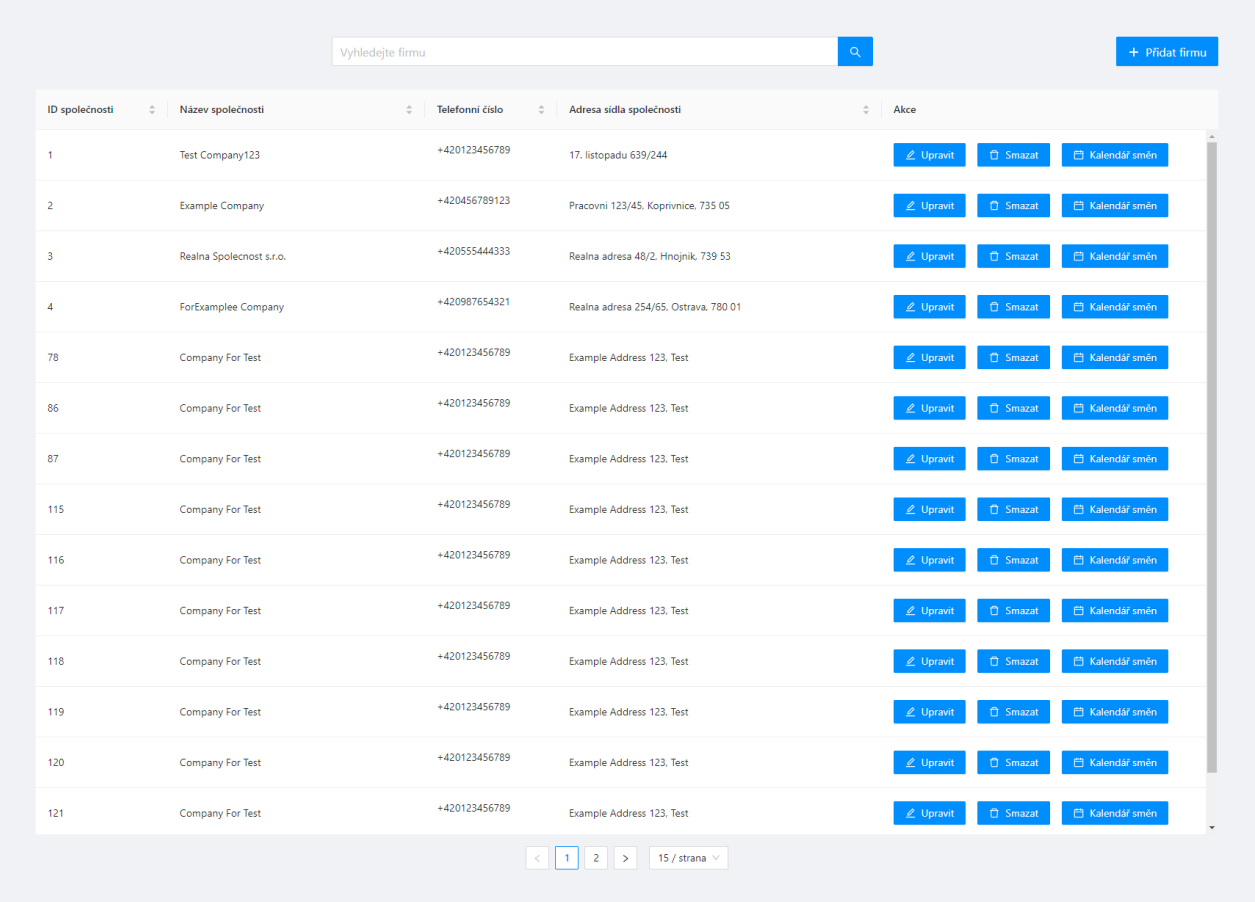
Je zde obsažen i jeden obsáhlý test, ověřující, zda je možné od začátku do konce projít otevřením, vyplněním a odesláním formuláře.

Aby byl test úspěšný, musí zde na rozdíl od integračních testů, dojít ke skutečnému kontaktování API, vložení záznamu do databáze a získaní pozitivní odpovědi nazpátek. Po dokončení se ověřuje, zda se frontendová tabulka rozšířila o tento záznam a po jeho otevření obsahuje dříve vyplněné údaje.

V případě úspěšného výsledku tohoto testu je možné s přehledem určit, že celá tato funkcionalita splňuje minimální funkční požadavky.

[18] https://docs.cypress.io/guides/references/best-practices#Having-tests-rely-on-the-state-of-previous-tests

5.4.3 Seznam firem



Třetí a zároveň poslední stránka na kterou je možno přistoupit přímo z hlavní nabídky je stránka „Seznam firem“. U této stránky je již na první pohled možné zpozorovat, že je velmi podobná předchozí stránce.

Tato stránka ve skutečnosti používá úplně stejnou obecnou komponentu pro tabulku, pouze obsahuje jiné sloupce a data. Další společnou obecnou komponentou je vyhledávací pole nad tabulkou.

Vyhledávací komponenta poskytuje auto-complete funkcionalitu a zároveň na základě prop `type` (pomocí které se určuje, zda má vyhledávat zaměstnance, či firmy), mění své chování.

Vyhledávání lze rozdělit na dva způsoby, na jejichž základě se mění způsob, jakým se následně zaměstnanec, či firma získá a zobrazí. V případě použití vyhledávací ikony, či zmáčknutí klávesy Enter, se současná hodnota na vstupu použije jako filtr. Tento filtr neboli textový řetězec, se poté hledá v již předem načtených záznamech z backendu (při každém obnovení stránky) a vrací všechny shodné objekty, které následně zobrazí v tabulce.

V případě že uživatel zvolí jednu z položek (reprezentována celým objektem zaměstnance/firmy) doporučenou auto-completem, se z předem načteného seznamu vyfiltruje tato konkrétní položka na základě svého unikátního ID a následně se zobrazí ve frontendové tabulce.

Oproti stránce se zaměstnanci zde v tabulce, konkrétně ve sloupci s akcemi přibyla jedna důležitá akce „Kalendář směn“. Tomu bude věnována další sekce.

Testy

Testování pro tuto stránku bylo zaměřeno především na vyhledávání firem, za pomocí dříve zmíněné komponenty.

E2E

Testuje se zde, zda vyhledávač vrátí hledané položky bez ohledu na velikost písmen. Uživatel by měl být schopen najít co hledá i v případě, že přesně nedodrží velikost písmen. Vyhledávaný text se tedy, společně se všemi prohledávanými atributy v záznamech, vždy převádí na malá písmena.

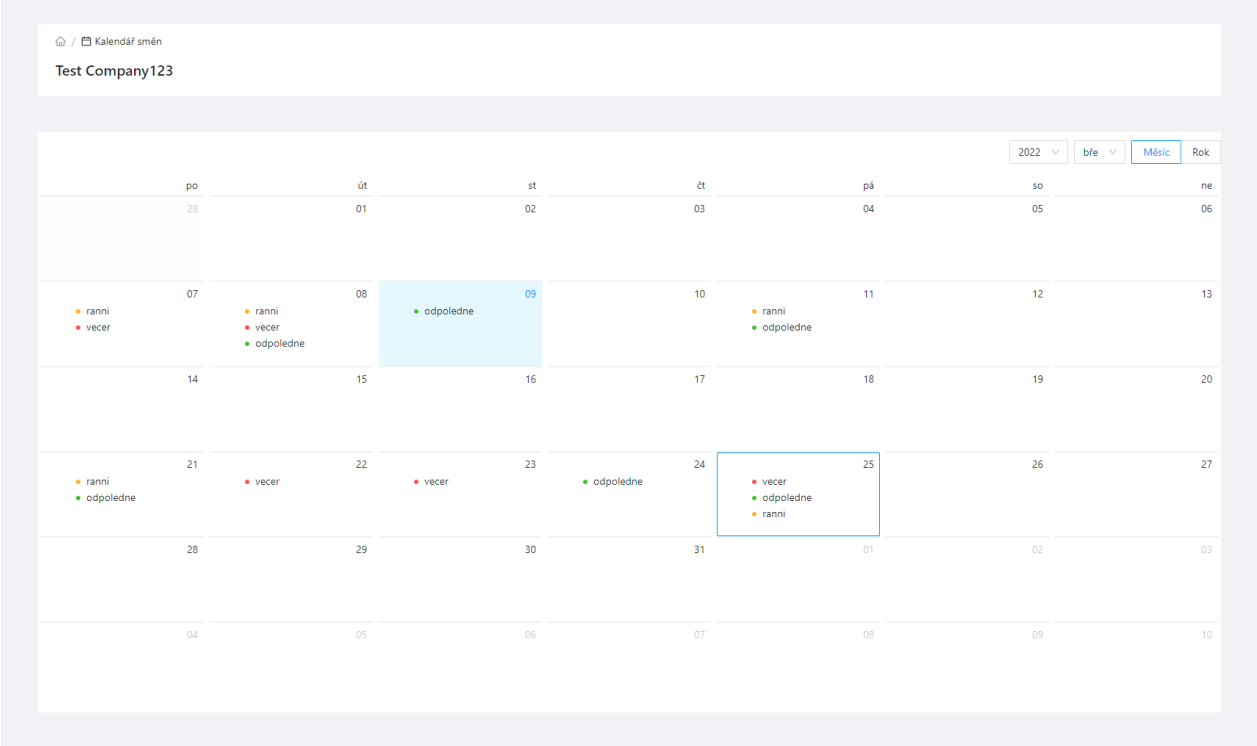
Další test provádí ověření, zda se po zvolení jednoho konkrétního záznamu vrátí pouze jeden a ten správný.

Ten následuje test jemu podobný, který však ověřuje, zda se po zadání klíčového slova, kterému odpovídá vícero záznamů, a stisknutí vyhledávacího tlačítka nebo klávesy Enter zobrazí seznam odpovídajících záznamů.

Přímo v seznamu nalezených výsledků lze v případě potřeby kliknout na avatara zaměstnance, a tak fotku zvětšit. Pro tuto funkcionalitu existuje další test, ověřující, zda se kliknutí na avatara momentální obsah stránky překryl lightboxem, obsahujícím zvětšený avatar.

Pokud se po vyhledávání uživatel rozhodne hledat jinou položku, zmáčkne ve vyhledávacím poli ikonu křížku. Poslední test kontroluje, zda po provedení této akce, dojde k obnovení tabulky do původního stavu a počet záznamů odpovídá počtu, který tabulka měla po načtení stránky.

5.4.4 Kalendář



Do kalendáře se lze dostat již zmíněným tlačítkem na stránce „Seznam firem“. Na této stránce se nachází pouze kalendář pro zvolenou firmu a hlavička. Hlavička obsahuje název firmy a Breadcrumb pro návrat na Hlavní stránku. Výchozí komponenta kalendáře je použita z Ant Design knihovny. Na jednotlivých buňkách jsou pak zobrazeny směny podle času (ranní, odpolední, noční) a barevně rozlišeny. Po kliknutí na buňku se zobrazí modální okno umožňující odstranit již existující směny, upravit je, či přidat novou směnu. Při přidávání nové směny je uživateli umožněno vybrat pouze směnu na čas, který ještě není vyplněn, aby se zabránilo vytváření duplicitních směn (např. dvě směny na ráno). V samotném kalendáři se pak uživatel může přepínat mezi měsíci, či roky, nebo změnit způsob zobrazení kalendáře (Měsíc/Rok).

V případě že dojde ke znovunačtení stránky, či návratu uživatele z plánovače zpět do kalendáře se záznam o současně zvolené firmě dočasně ukládá do LocalStorage[19] v prohlížeči. To proto, že při znovunačtení stránky dochází k resetování stavu aplikace. Při načtení stránky s kalendářem se tedy vždy zkontroluje, zda je v MobXu načten list firem a je firma s ID z URL stránky s kalendářem dohledatelná. Pokud ne, použije se záznam z LocalStorage, aby se nemuselo pokaždé provádět načítání firem z databáze. Pokud ano, záznam v LocalStorage se aktualizuje momentálně zvolenou firmou.

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

Po volbě buď již existující směny, nebo vytvoření nové se uživatel dostane na stránku plánovače směn.

Testy

Na této stránce bylo testování zaměřeno především na správné zobrazování směn v kalendáři a následnou manipulaci se směnami, jako je vytváření, mazání, či přesun do plánovače pro konkrétní směnu.

E2E

V této sadě testů se opět používá `beforeEach` hook, který před spuštěním každého z testů provádí pročištění záznamů v testovací databázi a přípravu standardizovaného prostředí. To znamená že se na začátku vyčká na smazání všech směn a následně se postupně vytvoří dvě směny v testovací databáze. Po dokončení těchto operací se navštíví stránka Seznam firem a pro firmu s ID 1 se otevře kalendář směn.

Na začátku se postupně provedou image snapshot testy hlavičky, samotného kalendáře a poté celé stránky. V případě že nedopadnou úspěšně se mohou testy zastavit, nemá smysl dále pokračovat.

Dále se provádí test hlavičky, konkrétně toho, zda zobrazuje správný název firmy. Ověřuje se i správnost formátu data v přepínacích tlačítkách kalendáře.

Další testy ověřují, zda se v kalendáři správně zobrazují právě ty 2 směny vytvořené před spuštěním testu a mají správně obarvený odznáček, podle toho, na jaký jsou čas.

Poslední 3 testy jsou zaměřeny na funkcionalitu.

První se zabývá mazáním směny. Na dni s existující směnou zobrazí modální okno a ověří zda je v seznamu ranní směna. Poté klikne na tlačítko pro odstranění a zkontroluje, že se otevřelo okno pro potvrzení smazaní (aby se nestalo, že uživatel směnu smaže omylem). Po potvrzení vyčká na dokončení akce a ověří že ranní směna zmizela ze seznamu.

Další je pro vytvoření nové směny. Otevře se v kalendáři okno, ve kterém by se měly nacházet ranní a večerní směna. Dojde k přepnutí na okno pro přidání směny a tam se zkontroluje, zda je tedy možné zvolit jen odpolední směnu. Po zvolení odpolední směny se ověří, zda došlo k přepnutí stránky na plánovač směn (změně URL).

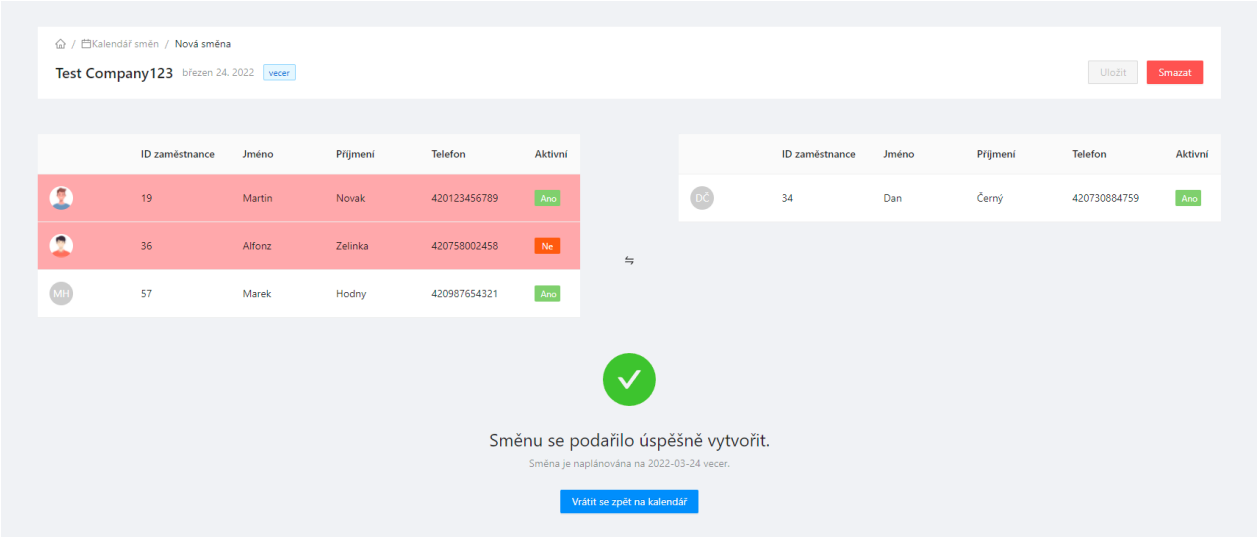
Poslední test je velmi podobný předchozímu, avšak se zde ověřuje už i korektnost (dat) zobrazené stránky plánovače. Po zvolení odpolední směny se spustí funkce ověřující, zda se zobrazila stránka s plánovačem, včetně odpovídajícího seznamu dostupných zaměstnanců, hlavičky s názvem firmy a správným datem.

Obecně u této sady testů byla výzva vyřešit práci s datumy. Jelikož se v průběhu práce na aplikaci posouval čas a měnily měsíce, bylo nutné vyřešit jakým způsobem je posouvat i v testech, jinak by testy při každém novém měsíci končily neúspěchem. Pomohla k tomu knihovna `moment`[20], jenž se stará o práci datem a časem. Při každém spuštění této sady testů se zjistí aktuální datum,

[20] https://momentjs.com/

od toho se odečte jeden den a výsledek se uloží do proměnné. Podle této proměnné se poté vytváří směny v `beforeEach` hooku. Taktéž při hledání a výběru buněk z kalendáře se používá toto datum.

5.4.5 Plánovač směn



Po výběru času směny z modálního okna kalendáře se uživatel dostane do samotného plánovače. Tato stránka je složena ze tří hlavních komponent. Hlavičky, která stejně jako u kalendáře zobrazuje název firmy a breadcrumb. Navíc je zde zobrazen i datum, čas, ID směny a dvě akční tlačítka: „Uložit“ a „Smazat“. Tlačítko „Uložit“ je dostupné pouze v případě že je směna validní, tedy obsahuje alespoň jednoho zaměstnance.

Zbylé dvě komponenty jsou tabulky. Jedna obsahuje seznam dostupných zaměstnanců pro danou firmu a druhá seznam zaměstnanců přiřazených na aktuálně zvolenou směnu. Pro přesun zaměstnanců mezi tabulkami se používá Drag&Drop. Avšak zaměstnance nelze přesouvat, aniž by splnili několik podmínek. V případě že je zaměstnanec v daný den, již přiřazen na jinou směnu, nemůže už být přiřazen na další. Zaměstnance taktéž nelze na směnu přiřadit v případě, že je neaktivní. Zaměstnanci, kteří nesplňují podmínky, jsou v tabulce označeni červeně a po najetí myší na takto označeného zaměstnance se uživateli zobrazí bublina s informací o tom, která podmínka nebyla splněna. Takto označení zaměstnanci nejsou přesunutelní, a tudíž nemohou být přiřazeni na směnu.

Po přiřazení požadovaných zaměstnanců na směnu pak stačí kliknout na tlačítko „Uložit“ a provede se uložení směny do databáze. Po dokončení této akce se uživateli zobrazí potvrzovací zpráva. V případě, že bylo uložení úspěšné zobrazí se zelená ikona a ověřující výpis s informací o tom na jaký termín byla směna naplánována. Pod výpisem se uživateli zobrazí tlačítko pro návrat zpět do kalendáře. V případě neúspěchu se zobrazí červená ikona informující o tom, že došlo k chybě při ukládání.

Kdyby uživatel omylem obnovil stránku, nebo odešel na dlouhou dobu od aplikace a chtěl dokončit rozdělanou práci, ukládá se průběžně stav plánovače při každé změně do LocalStorage, podobně jako na stránce s kalendářem.

Tabulky byly řešeny pomocí knihovny React-Table[21] a React Beatiful Drag-n-Drop[22]. Pro řádky i sloupce tabulek byl vytvořeny jednotné rozhraní, které mohou používat jak tabulky Ant Designu, tak React-Table. Stylování tabulek bylo upraveno tak, aby odpovídalo komponentě tabulky z Ant Design knihovny.

Testy

Testování plánovače spočívá především v ověření správné funkčnosti validace zaměstnanců, přesouvání zaměstnanců do směn pomocí drag&drop a ukládání směn.

Unit/Integrační

Jednou z mnoha věcí, které bylo třeba pro tuto stránku otestovat, bylo především přesouvání zaměstnanců mezi tabulkami, přesněji validace toho, zda mohou být přesunuti. Validace přesunů byla nejdříve testována na úrovni unit testů. Byl proto vytvořen testovací seznam směn, dostupných zaměstnanců a tři různí zaměstnanci, reprezentující 3 možné scénáře při validaci.

První test zjišťuje, zda se v případě, že je zaměstnanec již přiřazen ve stejný den na jiné směně vyhodnotí jako invalidní, tudíž zda bude výstupem validační funkce `isInvalid` chybová hláška. Tato chybová hláška musí obsahovat text, informující o tom, že zaměstnanec je již přiřazen na jiné směně a na které. Druhý test ověřuje, zda se neaktivní zaměstnanec vyhodnotí jako invalidní. Nevalidní zaměstnanci jsou totiž už jen vedeni v databázi, ale nemohou být nikam přiřazováni. V případě že je zaměstnanec neaktivní, musí být vyhodnocen jako invalidní a výstupem funkce musí být opět chybová hláška, tentokrát informující o tom, že je zaměstnanec neaktivní.

Poslední test má za úkol vyzkoušet, zda je možné na směnu přiřadit validního zaměstnance. Tudíž se ověřuje, zda funkce `isInvalid` pro validního zaměstnance vrátí `false`.

E2E

Podobně jako u stránky kalendáře se i zde před každým testem kompletně vyčistí tabulka se směnami a vytvoří se dvě nové. Řešení práce s datumy je zde taktéž použito stejné jako u kalendáře, tudíž se směny vytvoří den před aktuálním datem. Po vytvoření směn se navštíví stránka s firmami, zvolí kalendář firmy s ID 1 a z kalendáře se následně vybere směna buňka s vytvořenými směnami.

První dva testy se zaměřují na vizuální stránku. Jeden pořizuje a ověřuje image snapshot hlavičky, druhý se stará o část s tabulkami.

Další tři testy jsou již zaměřeny na neplatné zaměstnance. První dva testují, zda je zaměstnanec, v případě že není platný, označen červeně a po najetí myší na řádek se zaměstnancem se zobrazuje korektní bublina s vysvětlením. Třetí test zkouší, zda není náhodou možné neplatného zaměstnance přesunout do tabulky směny. U dvou zaměstnanců (neplatných ze dvou odlišných důvodů) se pokusí

[21] https://react-table.tanstack.com/docs/overview
[22] https://github.com/atlassian/react-beautiful-dnd

provést přesunutí do vedlejší tabulky a následně ověří, zda zaměstnanec zůstal v původní tabulce a nepřesunul se do druhé.

Následující test zkouší, zda není možné uložit nevalidní směnu (s prázdnou tabulkou zaměstnanců). Vytvoří novou směnu, kontrolně zkusí přesunout nevalidního zaměstnance na směny. Ten by však měl zůstat v původní tabulce a směna by měla zůstat prázdná. Nakonec se otestuje, zda je tlačítko „Uložit“ zablokováno.

Další, co testy zkoušejí je přesouvání validních zaměstnanců mezi tabulkami pomocí drag&drop. Zvolí se z tabulky volných zaměstnanců jeden validní zaměstnanec a ten se přiřadí na směnu. Po přesunutí se ověří, zda v tabulce směny tento zaměstnanec přibyl. Tento zaměstnanec se poté přesune zpátky do tabulky volných zaměstnanců a opět se ověří, jestli je zpátky v původní tabulce.

Další test rozšiřuje předchozí o to, že po přesunutí zaměstnanců na směnu se směna uloží a z následně zobrazeného tlačítka se provede přesun zpět na stránku s kalendářem. Poté se ověří, zda se směna zobrazila v kalendáři. Dále se směna zobrazí a provede se ověření, zda jsou údaje o směně, včetně rozložení zaměstnanců stejné jako při uložení.

O mazání směny z prostředí plánovače se stará další test. Ten zvolí v kalendáři existující směnu a po přesunu do plánovače klikne na tlačítko „Smazat“. Zkontroluje, zda se zobrazilo potvrzovací okno a akci smazání potvrdí. Po úspěšném smazání test ověří, zda došlo k přesměrování zpět na stránku kalendáře a zmizel z něj záznam o právě smazané směně.

Finální test v této sadě je zaměřen na zachování integrity dat po obnovení stránky. Test vybere existující směnu z kalendáře, zapamatuje si rozdělení zaměstnanců v tabulkách a provede znovunačtení stránky. Po obnovení stránky se záznamy v tabulkách porovnají s původními zapamatovanými hodnotami. Nakonec dojde přes breadcrumb k přesunu zpět na kalendář. Tam se ověří, zda je zobrazena správná firma, tedy jestli je v hlavičce zobrazen správný název firmy.

5.4.6 Testování API

Jak již bylo zmíněno v kapitole o výzvách při testování, se skutečným API během testování není vhodné komunikovat, jelikož to do testů přináší spoustu proměnných a testy by se kvůli tomu mohly stát flaky. Nikde v Unit ani Integračních testech proto nedochází ke kontaktování skutečného API. Pro tyto testy je důležité pouze to, zda a jak se funkce pro komunikaci s API volají a jak následně frontend reaguje na odpovědi z API. Jelikož je ale komunikace mezi frontendem a backendem prostřednictvím API nedílnou součástí této aplikace, je třeba aby byla i důkladně otestována. Testování a simulování zasílání požadavků na API, včetně mockování odpovědí, se bude věnovat následující část.

Pro namockování komunikace s API v prostředí Jestu byla využita knihovna `jest-fetch-mock`[23]. Ta se stará o to, aby se ve všech Jest testech při použití příkazu `fetch` používal mock a ne skutečný `fetch`[24]. V samotné sadě testu je při používání potřeba do `beforeEach` hooku, přidat volání funkce

[23] https://www.npmjs.com/package/jest-fetch-mock
[24] https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API

`resetMocks`, jenž zajistí že se do dalšího testu náhodou nedostanou data z fetche předchozího testu.

Testování nejdůležitějších operací lze popsat na příkladu stránky Seznamu zaměstnanců. Zde se v prvním testu ověřuje načtení listu zaměstnanců z databáze při příchodu na stránku. Pro test se vytvoří pole o dvou objektech simulující zaměstnance načtené z databáze. Následně se za pomocí funkce `mockResponseOnce` nastaví, jakou odpověď bude vracet následující volání metody `fetch`. V prvním argumentu se mu nastaví tělo odpovědi, tedy falešný list zaměstnanců. Jako druhý argument je dán návratový kód, se kterým metoda odpoví. Poté je potřeba zavolat samostatnou funkci pro načtení listu zaměstnanců, která však nepoužije skutečné fetch API, ale předem namockovanou verzi včetně namockované odpovědi. Jedná se o asynchronní funkci, je tedy potřeba počkat na odpověď (`await`).

Po získání odpovědi se ověří, zda byl `fetch` vyslán na správný endpoint a zda byl volán pouze jednou. Poté se zkontroluje, zda odpověď měla očekávaný stavový kód, jenž byl předem nastaven, a jako návratovou hodnotu obdržela definovaný list, reprezentující falešný seznam zaměstnanců, se správným počtem záznamů. Tento test je možné vidět na výpisu 7.

```
1  fetchMock.mockResponseOnce(JSON.stringify(employees), { status: 200 });
2
3  const responseEmployees = await getEmployeeList();
4
5  expect(fetch).toHaveBeenCalledWith('http://localhost:8000/api/employee-list');
6  expect(fetch).toHaveBeenCalledTimes(1);
7  expect(fetch.Response().status).toEqual(200);
8  expect(fetch.Response().statusText).toEqual('OK');
9  expect(responseEmployees).toEqual(employees);
10 expect(responseEmployees).toHaveLength(2);
```

Výpis kódu 7: Ukázka testu úspěšného API dotazu

Testování vytvoření zaměstnance je řešeno podobným způsobem. Vytvoří se objekt zaměstnance, ten se vloží jako tělo do funkce `createEmployee`. Výsledek fetche se opět namockuje pomocí `mockResponseOnce`. Tento test navíc kontroluje, zda se `fetch` volal se správným tělem a především POST metodou. Nakonec se i ověří, že se jako odpověď vrátila kopie nově vytvořeného zaměstnance, nyní již obsahujícího i parametr ID.

V neposlední řadě bylo třeba ověřit chování frontendu, v případě že se během komunikace s API někde vyskytne chyba. Takovýto scénář byl vytvořen při vytváření nového zaměstnance. Odešle se korektně vyplněný formulář na API, tam však dojde k chybě a uživateli se zobrazí chybová hláška. Implementace testu je oproti předchozím mírně odlišná. Volá se zde sice opět funkce `createEmployee`, avšak tentokrát v `try/catch` bloku, protože navrácení chyby z této funkce by automaticky ukončilo test neúspěchem. V bloku `catch` se automaticky očekává a kontroluje že funkce vyhodila chybu. Ověřuje se i správnost chybové hlášky. Pro namockování odpovědi funkce `fetch` zde bylo potřeba použít funkci `mockRejectOnce`, která simuluje vrácení chyby (400, 500...) ze strany API. Ukázka tohoto testu se nachází na výpisu 8.

```
fetchMock.mockRejectOnce(new Error('Unable to create employee'));

try {
    await createEmployee(employee);
} catch (e) {
    await expect(fetch).toHaveBeenCalledTimes(1);
    await expect(fetch).toThrowError();
    expect(e).toEqual(Error('Unable to create employee'));
}
```

Výpis kódu 8: Ukázka testu neúspěšného API dotazu

Kapitola 6

Evaluace

Po dokončení, či v průběhu implementace, je potřeba provádět nějakou formu evaluace dosažených výsledků, aby bylo možné na základě něčeho určit míru kvality implementace a otestování. Pro dosažení relevantních výsledků je vhodné zvolit několik různých metod, tak aby bylo zajištěno, že je evaluace tvořena z více aspektů a pohledů. Proto bude tato kapitola obsahovat výslednou evaluaci dosaženou několika různými způsoby.

6.1 Code coverage

Code coverage je jednou z nejčastěji používaných metrik pro evaluaci zdrojového kódu webové (či jiné) aplikace. Code coverage je metrika určující množství otestovaných řádků kódu programu, tedy řádků, které jsou validovány minimálně jedním testem.

Výhodami používání code coverage je například možnost snadněji identifikovat části kódu, které nejsou validovány žádnými testy a mohly by mít neočekávané chování, poskytnout vývojovému týmu určitý pohled na celkovou kvalitu dosavadního kódu, či objevit nevyužívané části kódu. Samotný code coverage report lze rozdělit na [23]:

- Function coverage – Kolik funkcí v testovaném úseku kódu bylo alespoň jednou zavoláno
- Branch coverage – Množství otestovaných větví kódu. Větvemi se rozumí blok kódu uvnitř nějakého rozhodovacího procesu `if`, `for`, `while` apod.
- Condition coverage – Množství otestovaných vstupů oproti definovaným podmínkám např. v `if`, či `switch` blocích kódu
- Line coverage – Kolik řádků daného úseku kódu bylo otestováno

Code coverage lze použít i pro end-to-end testy. Ikdyž Cypress (ani podobné nástroje) nepracují přímo se zdrojovým kódem, ale pouze simulují uživatelskou interakci na webu, dokáží pomocí dodatečných nástrojů code coverage vygenerovat. Jedním z těchto nástrojů je právě již dříve zmíněný

Istanbul, který během průchodu end-to-end testy zjišťuje a zaznamenává volané řádky zdrojového kódu. Na základě toho, poté dokáže vytvořit stejný code coverage report jako Jest. I přesto že code coverage je užitečná metrika, nelze se ji definitivně řídit. Stoprocentní code coverage totiž nutně neznamená pokrytí všech možných krajních případů, které mohou nastat, a tudíž nevypovídá o samotné kvalitě testů. Taktéž z něj není možné zjistit, zda testy správně vystihují funkcionální požadavky dané aplikace. Dále je možné mít relativně vysoký code coverage, avšak netestovat kritické a klíčové aspekty aplikace (funkcionality) aplikace. V neposlední řadě je získání code coverage nad hranici okolo 80% často velmi náročný proces zahrnující psaní nadbytečných a ne zcela užitečných testů [23].

6.1.1 Výsledky code coverage:

Report vygenerovaný Jestem pomocí příkazu `npm run test:coverage` zahrnující pouze unit/integrační testy. Všechen kód validován v end-to-end testech je zde ignorován a považován za neotestovaný. Naměřené výsledky bez zahrnutých E2E testů jsou zahrnuty v tabulce 1.

Tabulka 1: Výsledky code coverage vygenerované Jestem

Soubor	Příkazy (%)	Větve (%)	Funkce (%)	Řádky (%)
Všechny soubory	18	15	14	18
src/components/dashboard/charts	70	50	56	70
src/components/form	65	100	50	65
CompanyForm.tsx	50	100	0	50
EmployeeForm.tsx	85	100	66	85
src/components/form/elements	85	71	78	85
src/components/search	0	0	0	0
src/stores	17	2	8	18
src/utils	36	29	53	33

Z výsledků lze vidět že je unit, nebo integračními testy pokryta menší polovina všeho zdrojového kódu. Soubory (složky), které nebyly v tabulce zahrnuty jsou buď pouze komponenty obalující jiné komponenty třetích stran, nebo pouhé entity, či jiné nezajímavé soubory. Komponenty z knihoven třetích stran, jsou obvykle již dostatečně otestovány a nemá tedy smysl pro ně psát vlastní testy. Druhým důvodem pro malé pokrytí, je pokrytí většiny zbylého kódu end-to-end testy, které poskytují lepší formu evaluace.

I když unit testy dodávají určitou jistotu při upravování, či rozšiřování aplikace, v praxi jsou většinou užitečnější integrační, nebo spíše end-to-end testy. Teprve ty totiž ověřují, zda všechny komponenty, funkce a služby fungují dohromady správně. Proto je u reportu vygenerovaného Cypressem

pomocí Istanbul knihovny vidět code coverage mnohem vyšší. Zde je celkový code coverage okolo 75%. Obrázek tohoto reportu je možné si prohlédnout na obrázku 9.

Na základě kombinace výsledných reportů je možné tvrdit, že aplikace splňuje minimální vhodnou míru code coverage (okolo 80%) a tudíž lze předpokládat, že podle této metriky je aplikace dostatečně otestována.

Do výsledku code coverage nebyly zahrnuty soubory ze složky `src/models`, jelikož se zde nacházejí pouze enumy, rozhraní, DTO a MobX entity.

6.2 Délka provádění testů

Možnou metrikou pro evaluaci kvality testů je změřit dobu součtu trvání vykonání všech sad testů. Doba celkového vykonávání testů byla ovlivněna především množstvím samotných testů.

V případě, že je doba vykonávání podezřele dlouhá, stojí za zkoušku ověřit dobu trvání jednotlivých sad testů. Existuje několik způsobů, jak dobu vykonávání testů urychlit [24]. Jedním z nich je rozdělit jeden velký a obsáhlý test na několik menších, pokud je to možné. Tím zároveň lze i docílit toho, že bude v případě neúspěšného průběhu testu snadnější určit kde došlo k chybě. Druhým způsobem, navazujícím na první, je využít paralelizaci, tedy vykonávat více na sobě nezávislých sad testů zároveň. Tím lze dosáhnout mnohem kratšího celkového času na strojích s mnoha jádry procesoru, čí více procesory. Naopak na méně výkonném bezplatném CI stroji, s dvoujádrovým procesorem a 7GB RAM, došlo k výraznému zkrácení doby vykonávání při seriálním běhu. Hlavním důvodem tohoto efektu je nejspíše fakt, že méně výkonný stroj při paralelním běhu testů utratí více času (a zdrojů) vytvářením, správou a přepínáním kontextu mezi větším množstvím procesů, než dokáže paralelizací získat [25]. Poslední efektivní technikou se ukázalo být použití cachování, především v případě testování v Jestu.

V případě tohoto projektu obsahujícího dohromady 29 jednotlivých testů v 11 sadách testů byly naměřeny výsledky shlédnutelné v tabulce 2.

Tabulka 2: Porovnání délky provedení Jest testů

Nastavení	Průměrná doba trvání (s)
Lokální stroj	
Výchozí	8,58
Cachování	7,23
Bez paralelizace	10,72
CI stroj	
Výchozí	15,21
Cachování	*nebylo použito*
Bez paralelizace	13,62

Tabulka 3: Porovnání délky provedení Cypress testů

Prostředí	Průměrná doba trvání (s)
Lokální stroj	83
CI stroj	163

U end-to-end testů v Cypressu již nastavení tolik jako u Jestu upravit nelze. Rychlost testů zde ale opět částečně zlepšilo vyhnutí se velkým a dlouhým testům a nahrazení je více menšími, kratšími testy. Zlepšení času vykonání taktéž bylo pozitivně ovlivněno nahrazením čekáním po specifický časový úsek `wait(n)` za již dříve zmíněnou funkci `waitFor()`. To taktéž přineslo konzistentnější výsledky testů a odstranilo velkou část flakiness. Celková naměřená průměrná doba vykonání všech sad end-to-end testů je k vidění v tabulce 3.

6.3 Lighthouse

Jedním z možných způsobů ověření kvality implementace je nástroj od společnosti Google zvaný Lighthouse[25]. Tento nástroj se používá pro analýzu několika zásadních aspektů webové aplikace (či stránky). Mezi ně patří rychlost a responzivita webu, přístupnost (pro lidi s nějakým smyslovým omezením), kvalita SEO a dodržení správných postupů (např. pro zajištění bezpečnosti webu). Lze ho v různých formách použít buď přímo v prohlížeči přes Developer Tools, jako rozšíření prohlížeče, nebo jako Node.js balíček v příkazové řádce. V rámci implementace byla analýza pomocí tohoto nástroje zahrnuta na konec CI pipeline.

Výstupem může být HTML stránka, kterou je možné vidět na obrázku 7. Analýza je rozdělena na výše zmíněné části, přičemž v každé části jsou detailnější informace objasňující finální výsledek. Nástroj taktéž nabízí několik nápadů na zlepšení stavu aplikace.

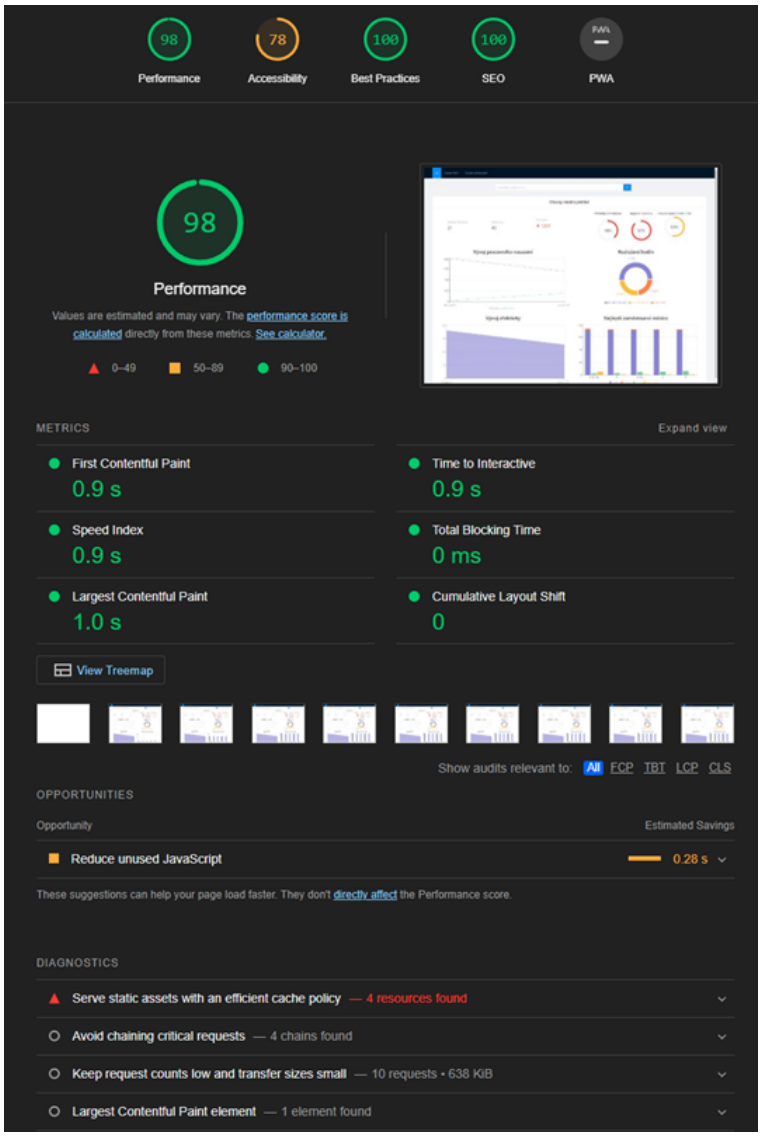
V části o výkonu byl výsledek velmi dobrý, avšak bylo nástrojem zjištěno, že by se dala rychlost načtení aplikace zrychlit odstraněním větších knihoven, cachováním statických souborů, či snížením počtu vysílaných HTTP dotazů.

Největší prostor pro zlepšení nástroj vyhodnotil v sekci „Přístupnost“. Zde bylo nalezeno množství HTML elementů, které nemají náležitě nastavené atributy pro čtečku obrazovky používanou nevidomými uživateli. Jelikož se do budoucna zatím nepočítá s využitím této aplikace osobou s nějakým z těchto postižení, nejedná se o závažný nedostatek.

V sekci o správných postupech nebyly nalezeny žádné potíže. V této sekci je testováno například, zda se do konzole nevypisují žádné chybové hlášky, či nejsou použity žádné zastaralé knihovny obsahující bezpečnostní díry. Jelikož byla frontendová část této webová aplikace vyvíjena v Reactu, o spoustu těchto věcí se nebylo potřeba starat, protože je React řeší na pozadí sám.

[25]https://developers.google.com/web/tools/lighthouse

Část SEO má stejně dobré výsledky jako předchozí část o správných postupech. Mezi testovanými body bylo např. obsažení titulku stránky, navrácení úspěšného HTTP status kódu, či zahrnutí základního popisu stránky. Tato část nebyla pro tuto aplikaci důležitá, jelikož se jedná o interní systém, který by neměl být cizím uživatelům přístupný.



Obrázek 7: Lighthouse report

6.4 Refaktorování komponenty

Užitečnou formu evaluace mohou poskytnout testy i při refaktorování komponenty. V případě že se kód komponenty změní, zatímco logika a vzhled v prohlížeči zůstane stejný, měly by všechny existující testy úspěšně projít dodat jistotu v nový kód.

Pro demonstraci byla zvolena komponenta, která refaktorování potřebovala a zároveň byla pokryta dostatečným množstvím testů. Jedná se o stránku Seznam zaměstnanců, konkrétně formulářová část. Tato stránka totiž obsahovala velké množství kódu, starala se o příliš mnoho funkcionality a nebyla přehledná. Cílem refaktorování bylo tedy rozdělit stránku na větší množství komponent, které budou zároveň obecnější, tudíž použitelné na více místech v aplikaci.

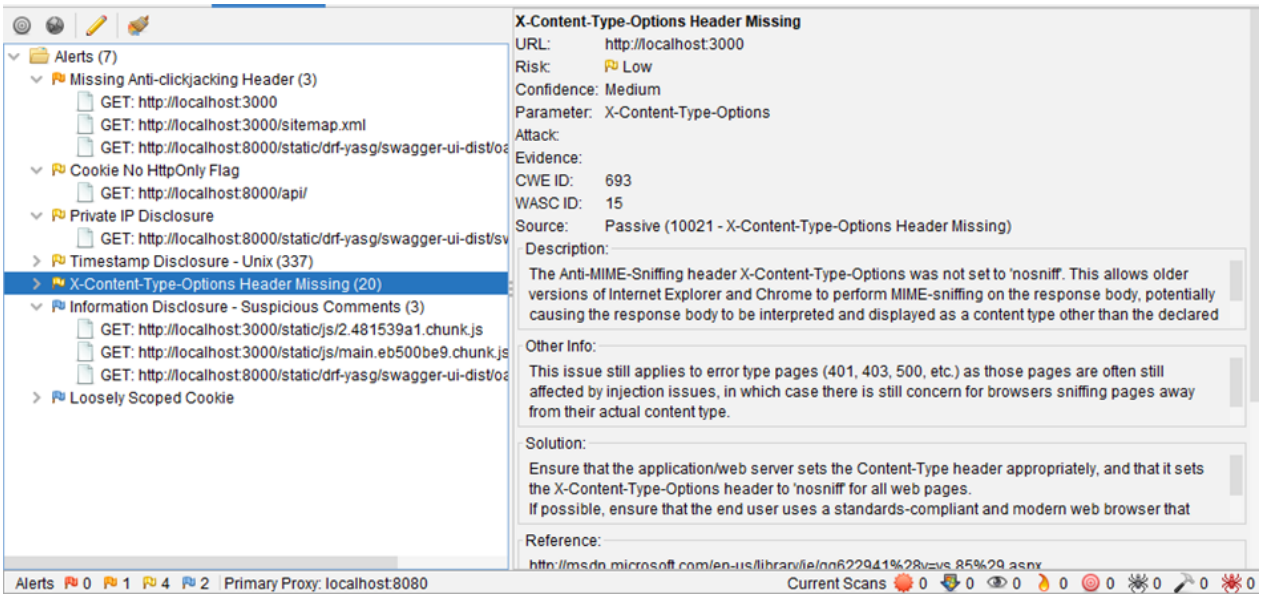
Před zahájením refaktorování byly spuštěny příslušné sady testů, včetně pořízení snapshotů. Hlavním bodem refaktorování byly dvě komponenty formuláře, sloužící pro nahrávání obrázků a souborů (příloh). Pro ně byly vytvořeny vlastní komponenty `ImageUpload` a `FileUpload`, které jsou nyní mnohem obecnější, dodatečně otypovány a komponenta formuláře se již nemusí starat o správu stavů a chování těchto dvou komponent.

Po refaktorování byly opět spuštěny příslušné sady testů, které napoprvé neprošly. Na stránce totiž nebylo možné nalézt komponenty podle hledaného `testid`. To bylo totiž u obou komponent v rámci refaktoru mírně upraveno, aby bylo variabilní. Došlo tedy k úpravě hledaného `testid` v této sadě testů. Po dalším spuštění již všechny testy, včetně snapshot testů, prošly.

Testy tedy v tomto příkladu poskytly ujištění o tom, že byly komponenty refaktorovány správně a nedošlo během refaktorování k nežádoucí úpravě logiky, či vzhledu komponent.

6.5 Bezpečnost

Pomocí nástroje OWASP ZAP[26] byl proveden automatický test bezpečnosti aplikace. Výsledek testu se nachází na obrázku 8.



Obrázek 8: Výsledky bezpečnostního (penetračního) testu

[26] https://github.com/zaproxy/zaproxy

Test neobjevil v aplikaci žádné závažné chyby, avšak nabídl několik doporučení, jak zabezpečení webu ještě vylepšit. Jednou z výtek byl chybějící Anti-clickjacking Header[27] (zamezující vložení stránky jako `iframe` do jiné), chybějící parametr hlavičky pro přísné dodržování typu obsahu odpovědi u několika HTTP dotazů a upozornění na pozůstalé komentáře ve zdrojovém kódu, které by mohly být zneužity.

Většinu bezpečnostních aspektů řeší na frontendu React a na backendu Django samy na pozadí, proto se až na těchto pár detailů nemusí programátor o bezpečnost příliš obávat.

[27] https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/X-Frame-Options

Kapitola 7

Závěr

Cílem této práce bylo vytvořit moderní webovou aplikaci pro správu pracovní agentury. V rámci toho šlo zároveň o analýzu existujících metod testování frontendu webových aplikací a jejich následnou implementaci a demonstraci na naimplementované aplikaci.

V první části práce byla provedena analýza toho, jaké výzvy nejčastěji obnáší automatizované testování webových aplikací a zjištěno jaké jsou možná řešení. Dále byla provedena analýza možných způsobů testování, z nichž několik bylo vybráno a použito. Na základě předchozí analýzy byly zvoleny použité technologie pro implementaci jak samotné aplikace, tak i automatizovaných testů. Před implementací aplikace, byl proveden návrh, popisující hlavní požadavky aplikace, testovací scénáře a celkovou architekturu aplikace. Následná implementace aplikace a testů probíhala zároveň, čímž bylo docíleno průběžné validace implementace. Finální evaluace byla prováděna různými metrikami pro pokrytí více aspektů aplikace.

Finální aplikace zvládá správu firem a zaměstnanců, plánování směn pomocí přehledného GUI včetně validace a tvoření průběžných reportů statistik, včetně grafické vizualizace. Všechny zmíněné části aplikace jsou otestovány a validovány automatizovanými sadami testů. Taktéž celý proces instalace, sestavení, otestování a částečné evaluace aplikace je plně automatizován pomocí Continuous Integration pipeline.

Během realizace této bakalářské práce jsem se naučil několika novým věcem v oblasti tvorby webových aplikací, ale především v oblasti automatizovaného testování, které mi do té doby bylo neznámé. Navíc jsem získal užitečné obecné vědomosti z oblasti testování webových aplikací.

Tuto práci by bylo možné rozšířit následujícími způsoby:

- Dodělat mobilní verzi aplikace za použití React Native
- Vyzkoušet tvorbu automatizovaných testů i pro backend (Django)
- Dodělat systém přihlašování pro zvýšení zabezpečení aplikace

- Zasílání notifikací např. o úpravě směny, či průběžného statistického reportu na email

Kromě výše zmíněných bodů by si aplikace zasloužila částečný redesign, který by mimo jiné více odpovídal barevnému schématu firmy, pro kterou byla tato aplikace vyvíjena.

Literatura

1. KLIMENCHENKO, Evgeny. *Front-end testing is for everyone* [online]. [B.r.] [cit. 2022-04-03]. Dostupné z: https://css-tricks.com/front-end-testing-is-for-everyone.
2. *A comprehensive guide to front end testing* [online]. [B.r.] [cit. 2022-04-03]. Dostupné z: https://www.perfecto.io/blog/comprehensive-guide-front-end-testing.
3. REGNAUD, Alexis. *Front-end testing strategy* [online]. ITNEXT, [b.r.] [cit. 2022-04-03]. Dostupné z: https://itnext.io/front-end-testing-strategy-5fddfd463feb.
4. *What is integration testing* [online]. [B.r.] [cit. 2022-04-03]. Dostupné z: https://www.softwaretestinghelp.com/what-is-integration-testing/.
5. *Smoke testing - explanation with example* [online]. [B.r.] [cit. 2022-04-03]. Dostupné z: https://www.testbytes.net/blog/smoke-testing-explanation-example/.
6. SŁAPIŃSKI, Mateusz. *What is flakiness and how we deal with it* [online]. AzimoLabs, [b.r.] [cit. 2022-04-03]. Dostupné z: https://medium.com/azimolabs/what-is-flakiness-and-how-we-deal-with-it-39b270ed5445.
7. ELOUSSI, Lamyaa. *Flaky tests (and how to avoid them)* [online]. Medium, [b.r.] [cit. 2022-04-03]. Dostupné z: https://engineering.salesforce.com/flaky-tests-and-how-to-avoid-them-25b84b756f60.
8. ZAKHARCHENKO, Artem. *When should I (not) use mocks in testing?* [Online]. DEV Community, [b.r.] [cit. 2022-04-09]. Dostupné z: https://dev.to/kettanaito/when-should-i-not-use-mocks-in-testing-544e.
9. KAMALIZADE, Ali. *How to use mocking in JavaScript tests using Jest* [online]. Better Programming, [b.r.] [cit. 2022-04-09]. Dostupné z: https://betterprogramming.pub/how-to-use-mocking-in-javascript-tests-using-jest-67cf513f47c0.
10. PARALOGARAJAH, Piraveena. *What is mocking in testing?* [Online]. Medium, [b.r.] [cit. 2022-04-09]. Dostupné z: https://piraveenaparalogarajah.medium.com/what-is-mocking-in-testing-d4b0f2dbe20a.
11. *What is continuous integration* [online]. [B.r.] [cit. 2022-04-03]. Dostupné z: https://www.atlassian.com/continuous-delivery/continuous-integration.

12. KATALON. *Top 10 benefits of continuous integration & continuous delivery* [online]. [B.r.] [cit. 2022-04-03]. Dostupné z: https://katalon.com/resources-center/blog/benefits-continuous-integration-delivery.

13. PECANAC, Vladimir. *What is continuous integration and why do you need it?* [Online]. [B.r.] [cit. 2022-04-03]. Dostupné z: https://code-maze.com/what-is-continuous-integration/.

14. *Agilní vývoj software* [online] [cit. 2022-03-27]. Dostupné z: https://en.wikipedia.org/wiki/Agile_software_development.

15. *Extreme Programming* [online] [cit. 2022-03-27]. Dostupné z: https://en.wikipedia.org/wiki/Extreme_programming.

16. OZDIL, M. Hamdi. *What is test-driven development? why TDD is so important?* [Online]. Let's Do It PL, [b.r.] [cit. 2022-04-03]. Dostupné z: https://medium.com/lets-do-it-pl/what-is-test-driven-development-why-tdd-is-so-important-e8ddade7010.

17. STEINFELD, Grant. *5 steps of test-driven development* [online]. [B.r.] [cit. 2022-04-09]. Dostupné z: https://developer.ibm.com/articles/5-steps-of-test-driven-development/.

18. *Advantages of test driven development* [online]. [B.r.] [cit. 2022-04-03]. Dostupné z: https://www.codica.com/blog/test-driven-development-benefits/.

19. WINTER, David. *9 benefits of test driven development* [online]. [B.r.] [cit. 2022-04-03]. Dostupné z: https://www.madetech.com/blog/9-benefits-of-test-driven-development/.

20. MAKINEN, Simo; MÜNCH, Jürgen. Effects of Test-Driven Development: A Comparative Analysis of Empirical Studies. In: 2014-01, sv. 166. ISBN 978-3-319-03601-4. Dostupné z DOI: 10.1007/978-3-319-03602-1_10.

21. SEGAL, Matt. *3 ways to deploy a Django backend with a react frontend* [online]. [B.r.] [cit. 2022-04-03]. Dostupné z: https://mattsegal.dev/django-spa-infrastructure.html.

22. *How to build a react application in a Django Project* [online]. [B.r.] [cit. 2022-04-03]. Dostupné z: https://www.saaspegasus.com/guides/modern-javascript-for-django-developers/integrating-django-react/.

23. PITTET, Sten. *Introduction to code coverage* [online]. [B.r.] [cit. 2022-04-03]. Dostupné z: https://www.atlassian.com/continuous-delivery/software-testing/code-coverage.

24. TANEV, Ivan. *Make your jest tests up to 20% faster by changing a single setting* [online]. [B.r.] [cit. 2022-04-03]. Dostupné z: https://dev.to/vantanev/make-your-jest-tests-up-to-20-faster-by-changing-a-single-setting-i36.

25. ZIELONKO, Alex. *Jest - speed up slow test suites* [online]. [B.r.] [cit. 2022-04-03]. Dostupné z: https://megafauna.dev/jest-speed-up-slow-test-suites/.

Příloha A

Velké obrázky a tabulky

```
1  test('Form method is not called and errors are shown when inputting wrong values', async () => {
2      const handleSubmit = jest.fn();
3
4      render(<EmployeeForm initialValues={new EmployeeDto()} onSubmit={handleSubmit} />);
5
6      const firstNameInput = await screen.findByLabelText('Jméno');
7      const submitButton = await screen.findByTestId('submit-button');
8
9      userEvent.type(firstNameInput, 'Da'); /** Use too short name **/
10     /** Forget last name **/
11     /** Forget to select category **/
12
13     await waitFor(async () => {
14         expect(firstNameInput).not.toBeNull();
15         expect(await screen.findByTestId('text-input-error')).toBeInTheDocument();
16         expect(await screen.findByText('Jméno je příliš krátké')).toBeInTheDocument();
17     });
18
19     userEvent.click(submitButton);
20     expect(await screen.findByTestId('category-input-error')).toBeInTheDocument();
21     expect(await screen.findByText('Kategorie musí být vyplněna')).toBeInTheDocument();
22
23     await waitFor(async () => {
24         expect(handleSubmit).not.toHaveBeenCalled();
25         expect(await screen.findByTestId('invalid-form-error')).toBeInTheDocument();
26         expect(await screen.findByText('Ve formuláři jsou chyby. Opravte je a zkuste to prosím znovu
               .'))
27         .toBeInTheDocument();
28     });
29 });
```

Výpis kódu 9: Příklad integračního testu pro odeslání nevalidního formuláře

All files

73.09% Statements 641/877 **72.92%** Branches 315/432 **76.49%** Functions 257/336 **73%** Lines 622/852

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

File	Statements		Branches		Functions		Lines	
src	100%	5/5	100%	0/0	100%	1/1	100%	5/5
src/api	33.94%	37/109	27.5%	11/40	50%	13/26	33.94%	37/109
src/components	100%	3/3	100%	0/0	100%	1/1	100%	3/3
src/components/calendar/calendar-date-cell	100%	7/7	100%	4/4	100%	4/4	100%	6/6
src/components/calendar/calendar-shift-list/item	100%	17/17	100%	4/4	100%	6/6	100%	17/17
src/components/calendar/modal	50%	3/6	100%	0/0	33.33%	1/3	50%	3/6
src/components/calendar/modal/modal-footer	77.78%	7/9	66.67%	4/6	50%	2/4	77.78%	7/9
src/components/dashboard	100%	6/6	75%	27/36	100%	2/2	100%	6/6
src/components/dashboard/charts	92.59%	25/27	100%	12/12	87.5%	14/16	92.59%	25/27
src/components/form	72.22%	13/18	81.82%	18/22	50%	4/8	72.22%	13/18
src/components/form/elements	93.75%	45/48	90.48%	38/42	94.44%	17/18	93.75%	45/48
src/components/layout	100%	3/3	100%	0/0	100%	1/1	100%	3/3
src/components/layout/form	100%	2/2	100%	0/0	100%	1/1	100%	2/2
src/components/search	70%	21/30	60%	18/30	87.5%	7/8	70%	21/30
src/components/shift-manager	100%	3/3	50%	3/6	100%	1/1	100%	3/3
src/components/table	100%	16/16	100%	10/10	100%	7/7	100%	16/16
src/components/table/elements	100%	17/17	100%	14/14	100%	9/9	100%	17/17
src/components/table/tableColumns	54.84%	17/31	100%	18/18	54.84%	17/31	55.17%	16/29
src/models/entities	100%	2/2	100%	0/0	100%	2/2	100%	2/2
src/models/enums	0%	0/0	0%	0/0	0%	0/0	0%	0/0
src/pages	86.89%	106/122	91.3%	63/69	76.19%	32/42	86.89%	106/122
src/stores	72.91%	253/347	57.73%	56/97	80.3%	106/132	72.73%	240/330
src/utils	67.35%	33/49	68.18%	15/22	69.23%	9/13	65.91%	29/44

Obrázek 9: Code coverage ve webovém GUI vygenerované Cypressem za pomocí pluginu Istanbul