VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
DEPARTMENT OF COMPUTER SYSTEMS

AUTOMATIZACE TESTOVÁNÍ WEBOVÝCH APLIKACÍ NA BÁZI JAVASCRIPTU SE ZAMĚŘENÍM NA FRONTEND A ANGULAR

TEST AUTOMATION OF WEB JAVASCRIPT APPLICATIONS WITH FOCUS ON FRONTEND AND ANGULAR

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

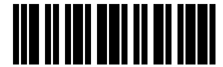
AUTOR PRÁCE — **JAKUB KAVKA**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. JOSEF STRNADEL, Ph.D.**
SUPERVISOR

BRNO 2024

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Zadání bakalářské práce



156825

Ústav: Ústav počítačových systémů (UPSY)
Student: **Kavka Jakub**
Program: Informační technologie
Název: **Automatizace testování webových aplikací na bázi JavaScriptu se zaměřením na frontend a Angular**
Kategorie: Analýza a testování softwaru
Akademický rok: 2023/24

Zadání:

1. Proveďte rešerši v oblasti vývoje aplikací založených na jazyku JavaScript; obzvláště se zaměřte na prostředí Angular, na frontendovou část webových aplikací a její testování.
2. Proveďte rešerši v oblasti automatizovaného testování aplikací na bázi JavaScriptu (ATAJS), zejména možností, prostředků, metod, vlastností a typických případů užití řešení ATAJS; zaměřte se zejména na testování frontendové části těchto aplikací.
3. Po dohodě s vedoucím zvolte vhodnou sadu řešení ATAJS a vhodnou sadu frontendových webových aplikací na bázi JavaScriptu a Angular (FWAJSA) pro jejich následné automatizované testování (AT); připravte podmínky pro AT.
4. S cílem co nejvyššího pokrytí proveďte AT zvolených FWAJSA pomocí zvolených řešení ATAJS v připravených podmínkách.
5. Na základě výsledků AT kriticky zhodnoťte a vzájemně porovnejte vlastnosti použitých řešení ATAJS z různých hledisek (např. možnost přípravy/provádění testů a odhalení chyb, využití pro jednotkové/unit či integrační testy, vizualizace průběhu testu a prezentace jeho výsledků) a navrhněte řešení, pomocí něhož lze maximalizovat pokrytí obecné FWAJSA testy.
6. Zhodnoťte vlastnosti řešení a výsledky pomocí něj dosažené; identifikujte silné a slabé stránky řešení, navrhněte možné směry jeho vylepšení a rozveďte ty, které považujete za nejvíce perspektivní.

Literatura:

- Dle pokynu vedoucího.

Při obhajobě semestrální části projektu je požadováno:

- Splnění bodů 1 až 3 zadání,
- představení kandidátních řešení ATAJS, kandidátních webových aplikací na bázi JavaScriptu, testovaných vlastností a pracovního postupu (workflow) - od přípravy testů až po zhodnocení výsledků testů.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Strnadel Josef, Ing., Ph.D.**
Vedoucí ústavu: Sekanina Lukáš, prof. Ing., Ph.D.
Datum zadání: 1.11.2023
Termín pro odevzdání: 9.5.2024
Datum schválení: 30.10.2023

Abstrakt

Cílem této práce je provést automatické testování různých webových aplikací v programovacím jazyce JavaScript ve frameworku Angular. Pro řešení tohoto úkolu jsou využity testovací framework Jasmine a testovací spouštěč Karma, které umožňují psát jednotkové testy pro aplikace postavené na Angularu. V rámci testování jsou zkoumány základní komponenty, služby a jejich vzájemné závislosti. Dále jsou testovány reaktivní formuláře, které umožňují dynamický přístup ke vstupům uživatele. Tato práce představuje přínosné přístupy k automatizovanému testování webových aplikací v Angularu a poskytuje užitečné poznatky pro vývojáře a testerům zabývající se testováním moderních webových technologií.

Abstract

The aim of this thesis is to perform automatic testing of various web applications in programming JavaScript language in Angular framework. To solve this task, the testing framework is used Jasmine and the Karma test trigger, which allow writing unit tests for applications built based on Angular. Basic components, services, and their interdependencies are explored in the testing framework. Also, reactive forms that allow dynamic access to user inputs are tested. This work presents beneficial approaches to automated testing of web applications in Angular and provides useful insights for developers and testers engaged in testing modern web technologies.

Klíčová slova

Angular, JavaScript, TypeScript, jednotkové testy, webová aplikace, automatické testy, integrační testy, testy od začátku do konce, regresní testy, reaktivní formuláře, testování software, Jasmine, Karma, statické testování, dynamické testování, funkční testy

Keywords

Angular, JavaScript, TypeScript, unit tests, web application, automatic tests, integration tests, end-to-end tests, regression tests, reactive forms, testing software, Jasmine, Karma, static testing, dynamic testing, functional tests

Citace

KAVKA, Jakub. *Automatizace testování webových aplikací na bázi JavaScriptu se zaměřením na frontend a Angular*. Brno, 2024. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Josef Strnadel, Ph.D.

Automatizace testování webových aplikací na bázi JavaScriptu se zaměřením na frontend a Angular

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Josefa Strnadela, Ph. D. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

....
Jakub Kavka
6. května 2024

Poděkování

Tímto bych rád poděkoval vedoucímu bakalářské práce Ing. Josefu Strnadelovi Ph.D. za jeho rychlé odpovědi prostřednictvím zpráv a za poskytování cenných rad během konzultací.

Obsah

1 Úvod **4**

2 Testování webových aplikací **5**
2.1 Webové aplikace 5
2.2 Testování programového vybavení 6
2.2.1 Proč testovat webové aplikace 6
2.2.2 Automatické testování 7
2.2.3 Jednotkové testy (Unit Tests) 8
2.2.4 Integrační testy (Integration tests) 9
2.2.5 Funkční testy (Functional tests) 12
2.2.6 Regresní testy (Regression tests) 12
2.2.7 Testy zátěže (Load tests) 13
2.2.8 Testy od začátku do konce (End to end tests) 14
2.3 Přehled souvisejících technologií 15
2.3.1 Framework Angular pro vývoj aplikací 15
2.3.2 Testovací rozšíření Jasmine 18
2.3.3 Spouštěč testů Karma 19
2.3.4 Reaktivní formuláře 20

3 Optimalizace postupu při psaní testů ve frameworku Angular **21**
3.1 Popis pomocné aplikace pro testování 21
3.2 Metody pro psaní testů ve frameworku Angular 21
3.3 Metody testování Angular komponenty s dětskou komponentou 23
3.4 Metody testování Angular služby (service) 24

4 Implementace testů **25**
4.1 Pomocná testovací aplikace 25
4.2 Pomocná testovací sada 26
4.3 Testování jednoduché Angular komponenty 27
4.4 Testování Angular komponenty s dětskou komponentou 30
4.5 Testování Angular komponenty se službou 32
4.6 Testování Angular služeb (service) 34
4.7 Testování reaktivních formulářů 38

5 Vyhodnocení testů a reakce na jejich výsledky **40**
5.1 Zhodnocení použitého testovacího frameworku 40
5.2 Hodnocení průběhu testování 40
5.3 Vyhodnocení testů 41

6 Závěr **43**

Literatura **44**

A Obsah přiloženého paměťového media a návod ke spuštění testované aplikace **46**

Seznam obrázků

2.1 Vztah ekonomické stránky vůči nalezeným testům 7
2.2 Příklad dekompozičního stromu 10
2.3 Příklad grafu volání 11
2.4 Příklad MM-cesty pomocí entitních uzlů 12

4.1 Struktura adresáře projektu 25

5.1 Ukázka vyhodnocení testů ve frameworku Karma 42

Kapitola 1

Úvod

Při vývoji webových aplikací, lze využít mnoho strategií, jak postupovat. Avšak při každé z těchto strategií, by se nemělo zapomínat na testovací část. Mnoho firem nevěnuje dostatečnou pozornost této etapě při vývoji. Pokud se webová aplikace netestuje během vývoje, lze očekávat, že při nasazení do produkce bude obsahovat spoustu chyb. Následná oprava je časově i finančně nákladná.

Každý programovací jazyk lze testovat. Programovací jazyk JavaScript a jeho framework Angular lze testovat různými typy testů, jako jsou jednotkové testy (unit tests), integrační testy nebo testy od začátku do konce (end-to-end tests). Tyto testy umožňují ověřit správnost funkcionality a chování aplikace z různých úhlů pohledu a přispívají k zajištění kvality a spolehlivosti výsledného produktu. Každá z těchto skupin testů je orientovaná na testování jiné části webové aplikace. Obecně je lepší zahrnout do testování všechny tyto skupiny.

Jednotkové testy jsou zaměřené na testovaní jedné určité části webové aplikace například komponent, tříd nebo servisních metod. Integrační testy se soustředí na testování větších celků. Vytváří modul, který obsahuje několik komponent, tříd a servisních souborů. Hlavní rozdíl mezi těmito testy je ten, že integrační testy testují i komunikaci mezi jednotlivými části aplikace. Pro psaní jednotkových testů a integračních testů pro aplikace psané v programovacím jazyce JavaScript, lze použít mnoho rozšiřujících knihoven a frameworků, které tuto práci značně usnadní. Framework Angular po jeho nainstalování poskytuje možnost vytváření jednotkových testů a integračních testů pomocí testovacího frameworku Jasmine a nástroje pro spouštění a běh testů Karma. Aby se usnadnilo psaní testů ve frameworku Angular, je poskytován ke každé komponentě soubor, který obsahuje základní testování a lze jej i dále upravovat pro přesnější testování. V novějších verzích frameworku Angular je i více funkcionalit spojených právě s jednotkovými testy a integračními testy. Testy od začátku do se zaměřují na testování celé aplikace. Testuje se interakce, kterou by dělal uživatel v reálném čase. Pro framework Angular existuje mnoho doplňujících knihoven a frameworků, žádná však není implementován v základním nastavení. Specifickým frameworkem pro psaní testů od začátku do konce je framework Protractor, který je navržen pro čekání na asynchronní události typické pro webové aplikace psané ve frameworku Angular.

Framework Angular obsahuje spoustu možností, jak psát webové aplikace. Práce má za cíl zaměřit se hlavně na ty základní testy jako jsou například testování komponent, služeb, reaktivních formulářů a asynchronní komunikace.

Práce je dělena do čtyř částí. První část je zaměřena na vysvětlení teoretických znalostí a postupů při testování ve frameworku Angular. Ve druhé části jsou vysvětleny metody a postupy, které jsou použity při řešení problému této práce. Tato řešení jsou implementována v části třetí. V poslední části je shrnutí dosavadního postupu a poznámky pro změny.

Kapitola 2

Testování webových aplikací

Tato kapitola se zabývá studiem a popisem webových aplikací. Informuje o různých typech testů webových aplikací a vysvětluje základní funkcionalitu použitých frameworků při testování.

2.1 Webové aplikace

Webová aplikace je software velice podobný stolním (desktop) aplikacím. Má však jednu velkou výhodu a tou je možnost přístupu uživatele přes webový prohlížeč. Není tedy nutná instalace různých nástrojů a doplňujících balíčků ke spuštění. Uživatelům je zároveň poskytnuta flexibilita a pohodlí, protože mohou přistupovat k aplikaci kdekoliv, kde mají přístup k internetu. Webový prohlížeč v tomto případě funguje jako tenký klient. Jelikož nelze přesně definovat rozdíl mezi webovými stránkami a webovou aplikací, jsou často zaměňovány.

- Webová stránka je dokument, který slouží k předávání informací uživateli přes webový prohlížeč. Obvykle bývají designované tak, aby na první pohled zaujali, upoutali uživatele a předali určité informace. Webovou stránku může vytvořit kdokoliv, kdo ovládá alespoň částečně jazyk HTML (Hypertext Markup Language) a CSS (Kaskádové styly). Mylně se lze setkat s definicí, že jsou webové stránky statické, ale nemusí tomu tak vždy být. Mohou být dynamické – měnit svůj obsah v čase a interagovat s uživatelem – ale jsou pasivní, což znamená, že poskytují informace, které pouze informují a nemají složitou funkcionalitu.

- Webové aplikace také zobrazují data, ale uživatel tyto data může upravovat, tvořit nová nebo mazat. Tyto operace jsou posílány na server, který je vykoná pro danou databází. Vytvoření webové aplikace není jednoduché a často je zapotřebí celý tým odborníků. Příkladem webových aplikací může být Gmail, což je webová emailová aplikace, která umožňuje spravovat kalendář, kontakty a používat další funkce, přímo ve webovém prohlížeči.

Webové aplikace se dělí na server neboli backend a na klienta neboli frontend.

Backend je část webové aplikace, která běží, na již zmiňovaném serveru. Obsahuje veškerou logiku a někdy i databázi webové aplikace. Hlavní funkcionalitou backendu je zpracovávat požadavky ze strany klienta a poskytovat mu odpovídající data. Také se zde provádí všechny složité operace a komunikace s databází.

Jak již bylo zmíněno existuje i klientská část webové aplikace, frontend, která zobrazuje uživateli data. Tato data získává pomocí dotazů na serverovou část. Poté data vykreslí do

grafické podoby, aby uživatel mohl s daty interagovat. Na frontendu se typicky neprovádí logické operace, protože ty mohou mít za následek zpomalení webové aplikace. Celá klientská část webové aplikace běží ve webovém prohlížeči uživatele na jeho zařízení, které nemusí mít dostatečné technické parametry pro tyto operace.

Pro grafické vykreslení webové aplikace uživateli se používají jazyky.

HTML (Hypertext Markup Language) je základním stavebním kamenem webových stránek i aplikací. Definuje strukturu a obsah dokumentu a určuje, jakým způsobem má být obsah strukturován a zobrazen ve webovém prohlížeči.

CSS (Cascading Style Sheets) je programovací jazyk, který umožňuje definovat vzhled stylu webových stránek.

JavaScript je programovací jazyk používající se hlavně pro vývoj interaktivního a dynamického obsahu. Je podporován všemi moderními webovými prohlížeči.

Dělení webové aplikace na klientskou a serverovou část je důležité pro efektivní fungování a její údržbu. [14]

2.2 Testování programového vybavení

2.2.1 Proč testovat webové aplikace

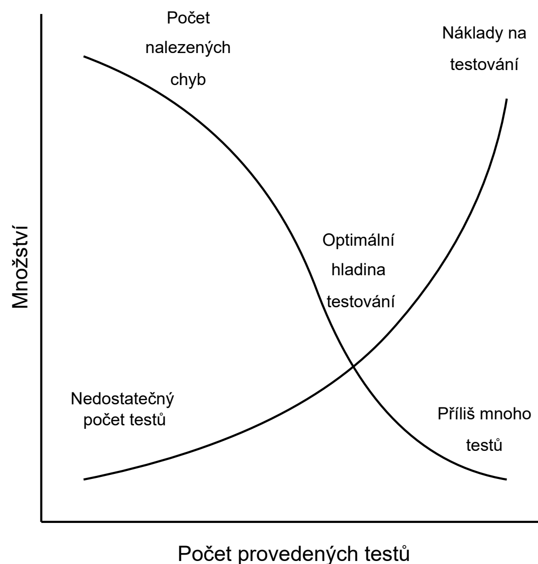
Udělat funkční webovou aplikaci, je velmi složité. Ve své podstatě záleží na spoustě faktorech, které výslednou webovou aplikaci ovlivňují. Patří mezi ně například rychlost internetového připojení, rychlost klientského zařízení, znalost programátora, čistota zdrojového kódu, i výběr správného programovacího jazyka nebo vhodné architektury. Nikdy nelze odstranit všechny faktory, ale lze je alespoň částečně eliminovat. A právě proto se webové aplikace testují, aby se nalezlo a odstranilo co nejvíce chyb, které ve webové aplikaci vzniknou.

Je mnoho způsobů, jak webovou aplikaci testovat. Všechny tyto způsoby lze rozdělit do metod statického testování nebo dynamického testování.

- Statické testování se zaměřuje na analýzu softwarových souborů, jako jsou zdrojové kódy, specifikace nebo dokumentace. Cílem této metody je nalézt chyby, které nejsou viditelné za běhu aplikace. Statické testování zahrnuje metody jako je statická analýza kódu, formální verifikace kódu anebo inspekce kódu.

- Dynamické testování na rozdíl od statického testování je vykonáváno za běhu webové aplikace. Cílem dynamického testování je zjistit, zda webová aplikace funguje správně a vyhovuje požadavkům. Do dynamického testování lze zahrnout automatické testování pomocí spouštění testovacích scénářů, testování výkonu nebo bezpečnosti.

Ve firmách se pro testování aplikací často vytváří týmy testerů, kteří provádí různé, již zmiňované testy. Nikdy nelze odhalit všechny chyby. I kdyby byl tým testerů zkušený, bude mít rozsáhlé testování za následek nalezení nákladnějších chyb. Jinak řečeno na to, aby bylo nalezeno velké množství chyb je potřeba velké množství testů. A pokud se vezme v potaz i ekonomická stránka testování webové aplikace, tak více testů stojí více peněz. Na obrázku je zobrazen zjednodušený model vztahu ekonomické stránky vůči nalezeným testům podle Pattona [6]:



Obrázek 2.1: Vztah ekonomické stránky vůči nalezeným testům

Aby ve firmě byla dosažena optimální hladina testování, tak s hledáním dalších chyb by se mělo přestat ve chvíli, kdy nalezení a oprava chyby stojí průměrně více peněz, než ponechání chyby ve webové aplikaci. Avšak ne vždy je možné tyto průměry vyčíslit.

2.2.2 Automatické testování

Testování webové aplikace provádí tester. Tester zná správné zadání webové aplikace, a tudíž dokáže posoudit, zda je chování chybné či nikoliv. Avšak u každé webové aplikace může nastat velké množství chybných scénářů, a některé se nemusí podařit odhalit. Jedním z důvodů může být časová náročnosti pro nalezení chyby nebo možnost přehlédnutí chyby. Z těchto důvodů se dá proces testování usnadnit pomocí automatických testů. Tyto testy jsou psané skripty v odpovídajícím jazyce, a jsou předloženy testovacímu nástroji, který testuje aplikaci, stejným, ale značně efektivnějším způsobem než ruční testování. Mezi výhody automatického testování se řadí:

- Rychlost – při využití automatických testů, je webová aplikace testována mnohem rychleji, než při manuálním testování.
- Efektivita – při automatickém testování se šetří čas testera, který webovou aplikaci testuje. Ušetřený čas může využít na vymýšlení nových testovacích scénářů a psaní dalších testů. V některých případech lze rozdělit testování webové aplikace do více testovacích balíků a spustit je na více zařízení. Tato metoda je využívána pouze při velmi komplikovaných testovacích scénářích.

- Správnost a přesnost – testy jsou prováděny přesně podle specifikace a vyhodnoceny testovacím nástrojem. Při práci testera může do testování zasáhnout lidský faktor, který může testy negativně ovlivnit.
- Neúnavnost – jelikož testování vykonává počítač nikdy se neunaví. Navíc může opětovně provést všechny již napsané testy, a to jenom při malé změně ve zdrojovém kódu. Tím lze zaručit, že provedená změna neměla vliv na jinou část webové aplikace.

V případě, že jsou při testování využity automatické testy bude vždy potřeba testera, aby tyto testy zhodnotil, popsal případné chyby pro možnou opravu nebo mohl psát další specifikace testů. Existuje spousta nástrojů pro vytváření automatických testů, například:

- Junit – pro testování aplikací v programovacím jazyku Java
- framework Jasmine – pro testování webových aplikací psaných v programovacím jazyce JavaScript
- Pytest – pro testování aplikací v programovacím jazyku Python

[16] [17]

2.2.3 Jednotkové testy (Unit Tests)

Jednotkové testy jsou procesy, které testují určitou funkcionalitu psaného kódu. Může se jednat o testování entit, jako jsou metody, funkce nebo třídy. Lze takto snadno odhalit chyby, které nastávají v jednotlivých entitách a přímo specifikovat jejich zdroj.

Vývojáři často využívají jednotkové testy pro metodu psaní testu před psaním kódu (test-driven development). Pokud se napíše první testovací sada a až poté se píše kód pro danou entitu, lze si při každé změně otestovat, zda je funkcionalita a chování entity správné.

Jednotkový test je blok kódu, který ověřuje část izolovaného kódu – entitu. Test je psán tak, aby odpovídal logickému chování dané entity v podobě, kterou zná programátor píšící jednotkový test. Jednotkový test je vždy vyhodnocen pouze jako true (pravda), nebo false (nepravda).

Jako jeden blok jednotkových testů lze definovat sadu těchto testů, která je známá pod pojmem testovací případ (test case). Pokud se splní sada testovacích případů bez chyby, lze říct, že je úspěšně otestováno správné chování části kódu. Avšak není nutné vždy vytvářet sadu testovacích případů.

Pro správné vyhodnocení jednotkového testu, musí tento test být vždy izolovaný, tedy nesmí využívat externí data z jiných částí webové aplikace. Pokud jsou potřebná data například z databáze, objektu nebo ze síťové komunikace, lze v takovém případě využít datové základy. Nejjednodušší je psát jednotkové testy pro logicky jednoduché bloky kódu.

Při vytváření jednotkových testů, lze zahrnout několik základních technik pro správné otestování všech testovacích případů.

- Logické kontroly – kontrolují, jestli entita provádí správné výpočty, sledují správný průchod kódem při očekáváném vstupu. A kontrolují, zda jsou definovány cesty pro všechny možné vstupy.
- Ohraničující kontroly – sledují reakci na zadané vstupy, reakci na typické vstupy, krajní vstupy anebo neplatné vstupy.

- Zpracování chyb – vyhodnocení reakce na vstupy s vyskytlou chybou a kontrola, zda při chybných vstupních datech nedochází k pádu celé webové apliakce.
- Objektově orientovaní kontroly – kontrola, zda je objekt správně aktualizován, pokud se změní jeho stav.

Jednotkové testy lze použít v několika případech při vývoji webové aplikace.

- Vývoj řízený testy (Test-driven development – TDD) – jedná se o metodu, kdy se nejprve napíší všechny testovací sady a až poté je psán kód pro dané entity. Při následném psaní kódu, lze ověřovat části webové aplikace.
- Po dokončení části kódu – pokud není použito TDD, lze napsat jednotkové testy pro právě dokončenou část kódu, která se tímto otestuje. Vytvořené testy nikdy nezanikají a je možné je spustit při každém dalším testování softwaru.
- Při nasazování softwaru vývojářem – v momentě kdy je část softwaru dokončena, je přidána k hlavní části softwaru, aby tvořila celek. Mezi klíčové činnosti tohoto postupuje patří kontinuální integrace a kontinuální dodávka (CI/CD). Během těchto činností je aplikována sada automatických testů, mezi které patří i jednotkové testy. Díky tomu, lze odhalit, že webová aplikace funguje i při aktualizaci.

Za velkou výhodu jednotkových testů se považuje implementace před nasazením webové aplikace do produkce a objevením tak chyb, které mohou nastat. Pokud se kód změní, lze tyto testy provést znovu, aniž by byla potřeba je upravit. Jednotkové testy mohou objevit chybu během několika sekund, zatím co tester by nad hledáním chyby strávil mnohem více času. V poslední řadě je také lze využít při psaní dokumentace kódu. Díky jednotkovým testům lze zjistit potřebné vstupy a výstupy i chování daného bloku kódu.

Testování pomocí jednotkových testů není vhodné ve chvíli, kdy je vyvíjen časový tlak na dokončení webové aplikace. Za pomoci určitých rozšíření je jednoduché vygenerovat testy na základě různých vstupů a požadovaných výstupů, ale je složité vytvářet testy pro logickou část kódu. Nehledě na to, že psaní jednotkových testů zabere vývojáři spoustu času, tak si zároveň může všimnou případného přepsání kódu a usnadnění jeho funkcionality. V tom případě může testování pomocí jednotkových testů zabrat ještě více času i peněz. [15]

2.2.4 Integrační testy (Integration tests)

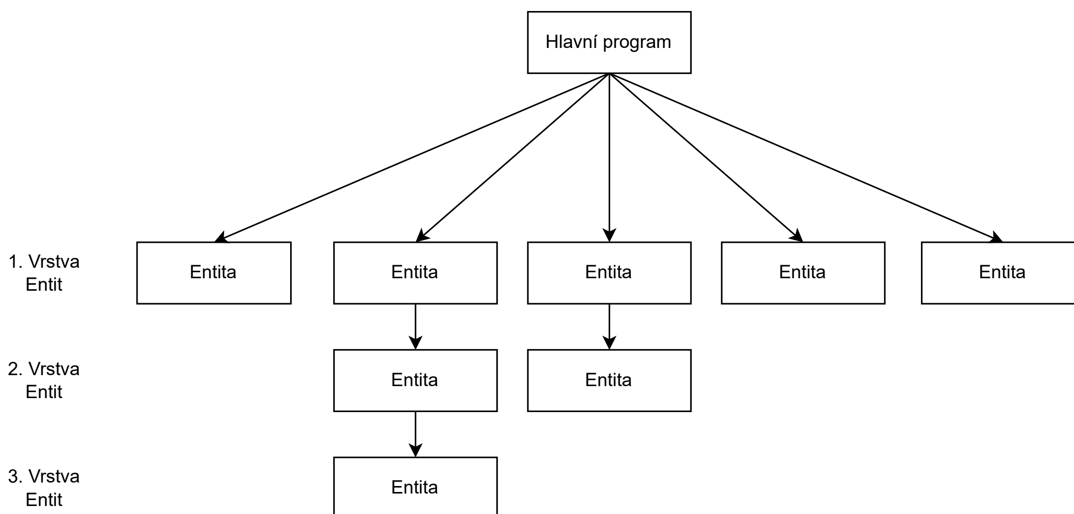
Integrační testování jsou procesy, při kterých je testována sada komponent kódu a jejich interakcí mezi sebou. Jedná se například o testování přenosu dat mezi dvěma objekty. Při integračních testech může nastat velké množství testovacích sad. Z toho důvodu se používají různé strategie pro vyhodnocení těchto testů.

Jednou z prvních možných strategii je integrace na základě dekompozice. V tomto případě jsou všechny entity sestaveny dohromady a testovány najednou. Základem této strategie je strom funkční dekompozice, který představuje základní zobrazením odvozené od konečného zdrojového kódu. Dále toto zobrazení ukazuje strukturální vztahy systému mezi entitami. Tato strategie předpokládá, že všechny entity byli testovány jednotlivě (například pomocí jednotkových testů), a tedy testuje pouze rozhraní mezi jednotlivými entitami. Strom funkční dekompozice je sestaven lexikálně správně vůči kompilování zdrojového kódu, aby nenastala chyba při rozsahu proměnných a u názvů entit. Při průchodu dekompozičním stromem lze využít tyto metody dekompozice:

- Integrace shora dolů (Top-down integration) – integrace začíná vždy hlavním programem (kořen dekompozičního stromu) a dál je rozvětven na entity, které hlavní program emulují. Jakmile je otestován hlavní program přechází se na nižší úroveň stromu. Při každém správném otestování je otestovaná část nahrazena a už se dál netestuje. Z toho vyplívá fakt, že je-li v dané části programu chyba, nachází se až v dalších emulačních entitách. Nevýhodou integrace shora dolů je vytváření dekompozičního stromu podle lexikálních pravidel kódu. Dochází zde k takzvanému vyhozenému kódu (k testování situace, která nikdy nenastane).

- Integrace zdola nahoru (Bottom-Up integration) – tato integrace je přesným opakem integrace shora dolů, až na jeden rozdíl. Entity, které se nachází v nižší úrovni stromu jsou nahrazeny moduly a dále se dělí. Při testování se pak začíná z nejnižších listů stromu a pro poskytnutí testovacího případu je použit modul rodičovské entity. U tohoto postupu nenastává tak často situace vyhozeného kódu.

- Sendvičová Integrace (Sandwitch integration) – sendvičová integrace je kombinací integrace shora dolů a integrace zdola nahoru. Díky tomu není potřeba nahrazovat entity, moduly, a lze použít celou větev stromu k otestování. Také lze u tohoto postupu lépe detekovat oblasti chyb.



Obrázek 2.2: Příklad dekompozičního stromu

Integrace na základě dekompozice má jednu nevýhodu, tou je nemožnost přesného určení chyby. To je zapříčiněno stylem sestavení testů a testováním pouze rozhraní mezi entitami.

Další strategií pro integrační testování je integrace založená na grafu volání. Na rozdíl od integrace na základě dekompozice nepoužívá dekompoziční strom, ale graf volání. Ten řeší problém s nemožností volání jiných entit. Graf volání se skládá z hlavního uzlu, který je představován hlavním programem. Dále je členěn na další uzly (entity emulující hlavní program), spojené hranami, které představují volání uzlů. Pokud nastane situace, při které je uzel A pod hlavním uzlem a uzel B je pod hlavním uzlem a současně uzel A volá uzel B, tak integrace založená na grafu volání může tuto situaci správně otestovat. Tento přístup lze testovat těmito metodami:

- Párové testování integrace (Pairwise integration) – při párovém testování integrace se testují dva uzly, respektive hrana mezi nimi. V tomto případě nastává více testovacích scénářů, ale je mnohem jednodušší přímá detekce chyb, v momentě kdy test bude vyhodnocen jako chybný, se problém nachází v jednom ze dvou uzlů. Velkou nevýhodou této metody je možnost výskytu již opravené chyby v jiném testovacím scénáři.
- Integrace sousedství (Neighborhood integration) – integrace sousedství využívá pro určení testovací sady informace z teorií grafů. Množina uzlů, které jsou testovány, jsou uzly vzdálené o jednu hranu od daného uzlu. Pro větší systémy má smysl tento rozsah rozšířit. V orientovaném grafu jsou zahrnuti všichni bezprostřední předchůdci a všichni bezprostřední následovníci. Výhodou této integrace je velké snížení relací integračních testů.

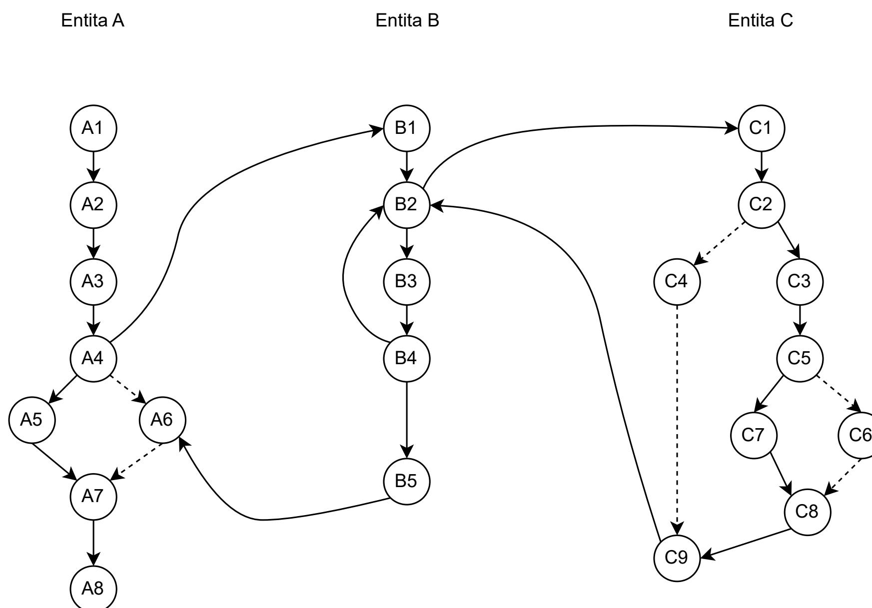
Obrázek 2.3: Příklad grafu volání

Při používání grafu volání při integračních testech, lze u větších projektů – zejména u metody integrace sousedství – vytvářet různé komplexní testy nazývané vesnice grafu. Testování těchto vesnic grafu přispívá správnému průchodu testem, protože ke správném výsledku se dá dojít více cestami. Nevýhodou při používání velkých shluků uzlů pro testování může být těžká detekce chybného místa. Také zde nastává problém opakovaného testování, kdy je při opravě uzlu potřeba znovu otestovat všechny relace integračních testů pro daný uzel.

Poslední strategií je integrace podle cesty. Jedná se o testování cesty mezi jednotlivými uzly. Je-li dán uzel, který má sadu zdrojových příkazů, jdoucí syntakticky za sebou, jedná se o cestu. Jeden z příkazů může být volání jiného uzlu, který má také svou cestu. Cesty se na sebe navazují až nám nastane jedna souvislá. V tomto případě nastat jeden zásadní problém, kdy dochází k opakovanému volání nadřazeného uzlu a podřízeného uzlu. K eliminaci tohoto problému existují dvě řešení. První možné řešení je opustit zásadu vstupu i výstupu a považovat toto volání za výstup, který opět začíná vstupem. A druhá možnost je potlačit opětovné volání nadřazeného uzlu. V takovém případě postrádají smysl integrační testy.

- MM-cesty (MM-paths) – jsou speciální případy cest, které kombinují funkční i strukturální testování. Je-li dána entita, tak je napsán její vyhodnocovací postup do uzlů,

které jsou spojené hranami závislosti. Tyto uzly mohou volat uzly z jiných entit nebo mohou volat uzly v dané entitě. To umožňuje kontrolu funkcionality i struktury uzlu.



Obrázek 2.4: Příklad MM-cesty pomocí entitních uzlů

Integrační testování podle cest je úzce spjato se skutečným chováním systému. Díky rozvržení entit do uzlů, lze podle volání detekovat přesné místo chyby. Avšak je daleko časově složitější správné definování MM-cest, než definování dekompozičního stromu. [11]

2.2.5 Funkční testy (Functional tests)

Funkční testování je zaměřeno na testování určité funkcionality. Spojením několika testovacích scénářů se testuje předem definovaná funkcionalita. Může se jednat například o přístup do webové aplikace nebo o platbu pomocí platební brány. Testovací sady jsou tvořeny převážně integračními testy, avšak neplní přesně jejich funkcionalitu, právě kvůli testování pouze určité části entity. [7]

2.2.6 Regresní testy (Regression tests)

Regresní testy jsou typem testování, který se provádí při každé velké změně ve webové aplikaci, aby se zaručilo, že změna nezpůsobí nechtěný problém. Toto testování je vhodné provádět pravidelně, jelikož při nepravidelném testování je složité nalézt změnu, která chybu způsobila. Při vyvíjení nových částí webové aplikace je potřeba, aby vývojáři často měnili již napsaný a funkční kód. Z toho důvodu jsou regresní testy volány iterativně. To znamená, že pokud se vyvíjí nová část webové aplikace a je otestována regresními testy, tak při další změně kódu v této části, jsou spuštěny stejné regresní testy a současně i nové regresní testy, které testují novou funkcionalitu. Při využívání regresního testování lze použít různé techniky postupu:

- Regresní testování jednotek (Unit regression testing) – jedná se o přístup, kdy se pohlíží na webovou aplikaci, jako na soubor entit a každá tato entita je testována zvlášť při nové změně.
- Částečné regresní testování (Partial regression testing) – tento přístup rozděluje webovou aplikaci na logické celky. Z těchto logických celků vybere ty nejvíce kritické a otestuje na nich různé scénáře testů. Pro ostatní celky použije jednotkové testování.
- Kompletní regresní testování (Complete regression testing) – tato technika je nejvíce komplexní možností regresního testování. Identifikuje všechny funkce, které by mohli rozbít funkcionalitu webové aplikace a napíše podrobné testovací scénáře pro všechny tyto funkcionality. Jedná se o nejdéle trvající techniku, ale také nejvíce účinnou. Aby tato technika měla největší úspěšnost, je dobré ji používat od začátku vývoje aplikace.

Regresní testování může přinést mnoho výhod do vývoje softwaru. Avšak před jeho nasazením je potřeba promyslet několik nevýhod, které obnáší:

- Časová a nákladová náročnost – při vytváření nových vylepšení do webové aplikace je potřeba rozšiřovat i regresní testy. Každé další rozšíření sebou nese čas, který je potřeba pro vyhodnocení testu. Tento čas bude narůstat a povede ke zvýšení nákladů na serverové zdroje.
- Složitost – nové funkcionality budou zpravidla budou složitější, než funkciolnality stávající a stejně složité budou i regresní testy.
- Pravidelná údržba – pro správný chod regresního testování je potřeba udržovat testy aktuální a aktualizované na nejnovější verzi webové aplikace. Tato údržba je velice nákladná i časově náročná.

[8]

2.2.7 Testy zátěže (Load tests)

Testy zátěže pomáhají zjistit maximální provozní kapacitu webové aplikace a také případná kritická místa, které by mohli bránit jejímu plnému provozu. Pro správné použití zátěžových testů je vhodné dodržovat tento scénář:

1. Identifikace scénářů, které jsou kritické z hlediska výkonu.
2. Určení profilů testování a stav webové aplikace vzhledem k identifikovaným scénářům.
3. Určení metrik, které se budou testovat s ohledem na výkonnostní cíle.
4. Návrh testů pro simulaci zátěže.
5. Implementace testů zátěže a zachycení metrik.
6. Analýza metrik.

Pro používání zátěžových testů existuje mnoho důvodů. Jedním z hlavních důvodů je potřeba zjistit, jak se webová aplikace chová nejen za normálních podmínek, ale i při nejvyšším vytížení. Doporučuje se zahájit testování s malým počtem virtuálních uživatelů

a postupně jejich počet navyšovat. Lze pak sledovat, jak se aplikace chová při zvyšování zátěže. V tomto případě je možnost měřit chování webové aplikace například při výkonu procesoru na 75 %. Pro úplné zátěžové testování je vhodné otestovat i situaci, kdy se přesáhne špičkový výkon. Tím lze zmapovat stav před pádem webové aplikace. [9]

2.2.8 Testy od začátku do konce (End to end tests)

Testování od začátku do konce, někdy značené jako E2E testování, testuje výkonnost a funkčnost celé softwarové aplikace, a to od začátku do konce pomocí reálných uživatelských scénářů. Toto testování pracuje s replikami reálných dat, aby byla zaručena realističnost testování. Hlavním cílem tohoto testování je odhalit chyby, které vznikají při interakci všech komponent.

Je možnost testovat jednotlivé entity i integrace mezi entitami jednotlivě, ale může nastat situace, kdy výstupní data z webové aplikace nejsou srovnatelná s očekávanými výstupními daty. Tato chyba může nastat po vykonání celé aplikační logiky a současně se testy mezi entitami mohou chovat bezchybně. Testováním od začátku do konce je možné tyto chyby nalézt a následně odstranit. Cílem těchto testů je dosáhnout co nejvyšší úrovně pokrytí. Tyto testy testují, jak samotnou aplikaci, tak i využívané subsystémy, databázi nebo aplikační server.

Testování od začátku do konce je spolehlivé a rozšířené díky těmto výhodám:

- Řízení kvality na více úrovních webové aplikace – moderní webové aplikace jsou skládány z více vrstev a tím tvoří komplexní architekturu. I když jednotlivé vrstvy mohou samy o sobě fungovat bezchybně, tak při propojení mezi sebou mohou nastávat chyby. Testování od začátku do konce může zaznamenat chyby mezi jednotlivými vrstvami webové aplikace.

- Testování kvality logiky aplikace – testování od začátku do konce, prvně testuje pozadí webové aplikace, jako je databáze a aplikační server, protože tyto části poskytují kritická data ostatním vrstvám.

- Zajistí konzistentní kvalitu webové aplikace v různých prostředí – při testování je funkcionalita vizuální části webové aplikace testována na větším množství prohlížečů, zařízení a platforem, tak aby nenastávala u uživatele nečekaná chyba.

- Testování aplikace třetích stran – pokud jsou do webové aplikace integrovány aplikace třetích stran, je možnost je otestovat pomocí testů od začátku do konce.

Životní cyklus testování od začátku do konce se skládá za čtyř částí:

1. Plánování testů – tato fáze nastává po dokončení integračního testování aplikace. Testy se plánují zejména na hlavní aplikační požadavky uživatele.

2. Návrh testů – na základě požadavků se vytváří vhodné testovací prostředí. Provádí se také analýza rizik a analýza využití, aby bylo možné efektivněji alokovat vhodné zdroje. Poté se vytváří testovací případy na daném prostředí.

3. Provedení testů – testovací případy se provádí na lokálním počítači i na serveru, kde se zaznamenává průběh testů.

4. Analýza výsledků – provádí se vyhodnocení výsledků testu, pokud nastala chyba, tak je tato chyba nahlášena vývojovému týmu. V poslední řadě se provádí retrospektiva projektu, aby se vyhodnotili procesy a prodiskutovali oblasti zlepšení.

Typy testování od začátku do konce:

- Horizontální testování od začátku do konce – jedná se o typ testování, kdy je procházena celá webová aplikace od logické části po vizuální část. Je potřeba se co nejvíce vžít do role uživatele a otestovat vše, co uživatel ve webové aplikaci může provést.
- Vertikální testování od začátku do konce – tento typ nejen že testuje logickou a vizuální část, ale také se zabývá správnou kontrolou procesů, které běží na pozadí webové aplikace. Například testuje token (znakový symbol) při přihlášení do webové aplikace, který je platný pouze určitý časový úsek.

Pro správné testování je vhodné zvolit i správné metriky testů. Jednou z hlavních metrik u testování od začátku do konce je stav přípravy testovacího případu. Tato metrika slouží k určení konkrétní pozice testovacích případů, které jsou připravovány, ve srovnání s těmi naplánovanými. Další vhodnou metrikou je sledování postupu testování. Tuto metriku je důležité sledovat pravidelně, jelikož poskytuje podrobné informace o procentuálním dokončení testů, tudíž informuje o tom, které testy jsou splněné a které nikoliv. Nebo také poskytuje informace o provedených a neprovedených testech. Pro detekci a kontrolu počtu závad lze využít metriku stav a podrobnosti o závadách. Tato metrika uvádí procento otevřených a uzavřených defektů za určité časové období. Dokáže také defekty rozdělit na základě závažnosti. Poslední metrikou, která je využívána, je metrika dostupnost prostředí. Udává počet provozních hodin a hodin naplánovaných pro testování v určitém časovém úseku.

Testování od začátku do konce má dvě zásadní nevýhody. První nevýhodou je časová náročnost. Při provádění testů na určitém prostředí je potřeba otestovat všechny vybrané testovací sady a jelikož jsou testy prováděné napříč celou webovou aplikaci, tak to zabere spoustu času. Druhou nevýhodou je potřeba nastavení správného testovacího prostředí. To často obnáší instalaci různých knihoven, agentů nebo získání přístup k určitým webům. Tuto nevýhodu částečně řeší testování na serverovém cloudu (vzdálené aplikaci na serveru). [13]

2.3 Přehled souvisejících technologií

V této kapitole je podrobně probrána dosavadní problematika testování ve frameworku Angular. Je zde vysvětleno, jak fungují jednotlivá rozšíření, která se používá při psaní testů. Také je zde vysvětleno, jakým způsobem lze použít samotný framework Angular pro usnadnění psaní testů.

2.3.1 Framework Angular pro vývoj aplikací

Framework Angular je vývojová platforma postavená na programovacím jazyce JavaScript. Přesněji řečeno na jeho rozšíření TypeScript. TypeScript je typovaný skriptovací jazyk. Právě typovost tohoto programovací jazyku je velice využívaná ve frameworku Angular. Framework Angular dále zahrnuje komponentový framework pro vytváření webových aplikací, sbírku dobře integrovaných knihoven, které pokrývají širokou škálu funkcí, a velkou

sadu nástrojů pro vývojáře, které pomáhají s vývojem, sestavováním a testováním kódu. Jelikož se jedná o OpenSource framework (volně dostupný), tak má širokou základnu programátorů, kteří se starají o vývoj nových funkcionalit frameworku a zároveň lze využít i knihovny vytvořené běžnými programátory.

Základním stavebním prvkem ve frameworku Angular je komponenta. Framework Angular poskytuje snadnou údržbu, přehlednost a organizovanost kódu, právě díky architektuře komponent. Každá komponenta obsahuje dekorátor, který definuje konfigurační možnosti pomocí selektoru, HTML šablony a kaskádových stylů. Pomocí selektoru lze danou komponentu přesně definovat a vykreslit v HTML stromu. HTML šablona obsahuje elementy, které jsou vykresleny do HTML stromu a kaskádové styly obsahují stylování tohoto stromu. Šablona i její styly mohou být definovány inline v rámci třídy v programovacím jazyku TypeScript nebo pomocí vlastností odkazu na HTML soubor a CSS soubor. Komponenta dále obsahuje třídu v programovacím jazyku TypeScript, která udává chování komponenty. Komponentu ve frameworku Angular je možné definovat pomocí přípony komponenty hello.component.ts.

```
1    @Component({
2        selector: 'hello-world',           // selektor
3        template: `<p>Hello World<\p>`,    // inline šablona HTML
4        styles: ['p {font-size: 20px;}'],  // inline kaskádové styly
5    })
6    export class HelloWorld {...}
```

Chování komponenty je definováno pomocí stavů a funkcí. Stavy se deklarují do pole třídy, kde se deklarují bez inicializátoru, a tím se stávají ihned veřejnými (public) pro celou webovou aplikaci. Funkce pro komponentu se deklarují jako metody třídy.

```
1    @Component({...})
2    export class HelloWorld {
3        text = 'Hello World!';                 // deklarace stavu
4        isItalic = false;
5
6        changeFont() {                         // deklarace funkce
7            this.isItalic = !this.isItalic;
8        }
9    }
```

Pokud je v šabloně HTML dynamicky se měnící hodnota, je potřeba tuto hodnotu vložit do dvojitých složených závorek. Tato syntaxe deklaruje interpolaci mezi vlastnostmi dynamických dat uvnitř programovacího jazyka HTML. Výsledkem je překreslení DOMu (dokumentového objektového modelu) a zobrazení nové hodnoty.

```
1    @Component({
2        template: '<p>Title: {{ text }}</p>',   // výpis stavu do HTML
             elementu
3    })
4    export class HelloWorld {
5        text = 'Hello World!';                 // deklarace stavu
6    }
```

Je-li potřeba dynamicky měnit hodnoty atributů v prvku HTML stromu, cílová vlastnost se zabalí do hranatých závorek. Atribut se tímto spojí s dynamickými daty a framework Angular dostává informaci, že tento atribut využívá tuto hodnotu. Není nutné deklarovat pro tuto funkcionalitu funkci v programovacím jazyku JavaScript.

```
1    @Component({
2        // propojení stavu isDisabled s atributem disabled
```

```
3         template: '<button [disabled]="isDisabled"></button>',
4     })
5     export class HelloWorld {
6         // deklarace stavu isDisabled v komponentě
7         isDisabled = true;
8     }
```

Další důležitou částí frameworku Angular je naslouchání událostem. Posluchač v HTML elementu je specifikován pomocí kulatých závorek a spojen s funkcí, která se má provést po vyvolání události. Je možné předat i objekt události pomocí implicitní proměnné $event.

```
1     @Component({
2         // propojení události klik s funkcí save
3         template: '<button (click)="save()">Save</button>',
4     })
5     export class HelloWorld {
6         // vykonání funkce save po kliknutí na tlačítko
7         save() {
8             console.log("Save changes!");
9         }
10    }
```

Při vytváření aplikace programátoři často potřebují upravit nebo rozšířit chování určité části HTML stromu. Například se může jednat o reakci na určitá data nebo vykreslení prvků na základě dat webové aplikace. Ve frameworku Angular pro takovou úpravu lze použít směrnice (directive). Tento koncept směrnic umožňuje přidávat k HTML elementu nová chování deklarativním a opakovaně použitelným způsobem. Jednou ze základních a zároveň vestavěných směrnic ve frameworku Angular, je směrnice vyhodnocovací podmínky *ngIf. Pomocí této podmínky lze rozhodnout, zda bude daný HTML element vykreslen či nikoliv.

```
1     // využití směrnice *ngIf pro vykreslení hlášky
2     <section class="hello-world" *ngIf="hasAdminScreen">
3         Hello from other world.
4     </section>
```

Ne vždy jsou vestavěné směrnice dostatečné pro řešení problémů. Proto je možné ve frameworku Angular vytvářet vlastní. Lze je identifikovat pomocí přípony directive, tedy například hello-world.directive.ts. Podobně jako komponenty obsahují směrnice dekorátor, který definuje konfigurační možnosti. Směrnice mají vlastní selektor, který udává název směrnice, když je volána. Také obsahují třídu v programovacím jazyku TypeScript, která definuje rozšířené chování, které daná směrnice přináší do HTML elementu. Směrnice mohou využívat objekty volané události nebo přijímat vstupní hodnoty.

```
1     @Directive({
2         // definování názvu selektoru
3         selector: '[appRedColor]',
4     })
5     export class RedColorDirective {
6         // deklarace elementu, který se bude měnit
7         private el = inject(ElementRef);
8
9         // změna stylu color na red
10        constructor() {
11            this.el.nativeElement.style.color = 'red';
12        }
13    }
```

```
// použití směrnice na element
<p appRedColor>Give me red!</p>
```

Ve větších projektech psaných ve frameworku Angular je mnoho komponent a některé z nich mohou obsahovat stejnou logiku. V této situaci lze využít služby (service), které poskytují stejnou logiku pro více komponent. Službu lze poznat od komponenty pomocí přípony service ve názvu souboru, například calc.service.ts. Služby také obsahují dekorátor, který obsahuje informace o jejich poskytnutí (providedIn). Toto poskytnutí udává, které soubory mohou danou službu využívat a zároveň definuje čas, kdy bude daná služba přeložena ze zdrojového kódu. Služba tak může být poskytnuta pro celý dokument (root) nebo jen pro jednotlivé komponenty. Služba dále obsahuje také třídu v programovacím jazyku TypeScript, která definuje její logiku. Příklad služby pro sčítání kladných čísel:

```
@Injectable({
    // definování místa, kde bude služba dostupná
    providedIn: 'root',
})
export class AddService {
    // sečtení dvou čísel
    add(x: number, y: number) {
        return x + y;
    }
}
```

Pokud se služba využije v komponentě, tak je potřeba ji vyvolat. Volání lze provést přes konstruktor a tak mít dostupnou službu pro celou komponentu a nebo se dá volat pomocí funkce `inject()` a tím je docíleno toho, že bude služba dostupná pouze pro daný rozsah.

```
@Component({
    // tlačítko pro součet
    template: '<button (click)="add()">add</button>',
})
export class HelloWorld {
    constructor(
        // vložení služby do konstruktoru, aby ji bylo možné použít
        // v celé komponentě
        private addService: AddService;
    ) {}

    save() {
        // volání funkce ze služby
        const final = addService.add(10, 10);
    }
}
```

Všechny komponenty, směrnice nebo služby je nutné deklarovat v Angular modulu. Jedná se o soubor, který umožňuje organizovat a shlukovat části aplikace. Webová aplikace může mít jeden modul, který bude definovat všechny entity, nebo může být definováno více modulů, pak je potřeba tyto moduly vzájemně propojit pomocí závislostí. [5]

2.3.2 Testovací rozšíření Jasmine

Jasmine je volně dostupný framework, který je hojně používaný pro testování kódu psaného v programovacím jazyku JavaScript.

Tento framework používá popisný styl psaní testů. Testy jsou psány jako specifikace, u kterých se popisuje očekávané chování jednotlivých částí kódu. Ve specifikaci lze definovat vstup, a podle toho vyhodnotit očekávaný výstup. Testovací skript obsahuje funkci

`describe()`, která seskupuje testy. Typicky bývá jedna funkce pro jeden testovací soubor. Funkce `describe()` obsahuje dva parametry. První parametr je typu řetězec, ten složí k popisu skupiny specifických testů. Díky tomu lze zvolit správný název skupiny a tím usnadnit čtení celého testovacího souboru i lidem, kteří budou soubor číst později. Druhým parametrem je anonymní funkce, ve která obsahuje jednotlivé testy.

Specifikace jsou definovány voláním globální funkce `it()`. Funkce `it()` taktéž obsahuje dva parametry. Prvním parametrem je řetězec, který slouží k popisu testu, a druhým parametrem je anonymní funkce, která obsahuje logiku testu. Anonymní funkce, volaná jako druhý parametr, musí obsahovat alespoň jedno očekávané chování. Tohle chování je vyznačeno asertivní funkcí `expect()`. Této funkci se předá proměnná, ve které je uložena testovaná hodnota a ta se později vyhodnocuje pomocí porovnávacích funkcí.

Porovnávače implementují srovnání mezi hodnotou v proměnné a očekávanou hodnotou. Na základě pravdivostních hodnot se rozhodne, zda je test pravdivý či nikoliv. Lze navázat více porovnávačů za sebe, a tak vytvořit řetězovou reakci testů, nebo vytvářet vlastní porovnávače. Aby bylo docíleno validního testu musí být všechna jeho očekávání pravdivá. Pokud je byť jedno očekávání nepravdivé, přestože je očekávaných hodnot více, považuje se celý test za nepravdivý.

Testovací framework Jasmine poskytuje metody `beforeEach()`, `beforeAll()`, které slouží k nastavení počátečních hodnot, ještě před provedením specifikace. Z názvu `beforeEach()` předem vyplývá, že se provádí před každou specifikací. Naopak `beforeAll()` se provede pouze jednou před všemi specifikacemi. Protože by mohlo docházelo k unikům alokované paměti, existují funkce `afterEach()`, která se provádí po každé specifikaci a funkce `afterAll()`, která se volá až po provedení všech specifikací. Při využití těchto metod se provádí potřebný úklid po testování. Všechny tyto metody obsahují parametry. Prvním parametrem je funkce, která obsahuje logiku pro nastavení nebo úklid po testu. Druhým parametrem je časový limit, který určuje, jak dlouho bude metoda naslouchat v případě asynchronní komunikace.

Testovací framework Jasmine podporuje běh specifikací pro asynchronní operace. Pokud jsou tyto operace v metodách zařazeny před specifikacemi, neprovedou se testy dříve, než metody vrátí odpověď typu Promise (slibu). Základní časový limit je nastaven na 5 sekund, ale lze jej přenastavit pro jednotlivé asynchronní volání nebo také změnit globálně. Asynchronní operace lze provádět přímo ve specifikaci. V tomto případě je pomocí funkce `done()` nebo `async`, `await` možnost pozastavení testu do doby, dokud nedostaneme požadovaný výsledek. [10]

2.3.3 Spouštěč testů Karma

Framework Karma je nástroj používaný na spouštění testů ve webovém prohlížeči při vývoji webových aplikací v programovacím jazyku JavaScript. Často se požívá současně s jinými testovacími nástroji jako jsou například Jasmine, Mocha nebo QUnit.

Při spuštěn frameworku Karma se vytvoří webový server. Na tomto webovém serveru se spustí zdrojový kód proti testovacímu kódu na webovém prohlížeči, který je připojen. Může být připojeno i několik internetových prohlížečů a nástroj Karma dokáže testy spustit na každém z nich současně. Potom co se testy provedou, tak se na každém prohlížeči zobrazí informace, které testy byly vyhodnoceny jako úspěšné a které nikoliv. Informace o výsledcích testů jsou podrobné a není problém identifikovat chyby v kódu.

Pokud provedeme změny v kódu, framework Karma je dokáže zachytit, odešle signál na webový server a automaticky pustí testy znovu. Tato funkce dokáže odhalit chyby v kódu ihned po jeho změně. [12]

2.3.4 Reaktivní formuláře

Ve frameworku Angular je možné využít dva typy formulářů podle [4]. Prvním typem jsou formuláře řízené šablonami a druhým typem jsou reaktivní formuláře.

Formuláře řízené šablonou umožňují přímý přístup k úpravě dat v šabloně, avšak jsou méně explicitní než reaktivní formuláře, protože se opírají o směrnice vložené do šablony spolu s proměnnými daty pro asynchronní sledování změn.

Reaktivní formuláře poskytují neměnnost pomocí pozorovatelných operátorů, sledování změn pomocí pozorovatelného toku dat a synchronní přístup k datovému modelu. Každá změna stavu reaktivního formuláře vrací nový stav, který zachová integritu modulu. Vstupy a hodnoty reaktivního formuláře jsou poskytovány jako pozorovatelné toky vstupních hodnot, ke kterým lze přistupovat synchronně.

Pro možnost použití reaktivních formulářů je nutné do komponenty nebo modulu importovat modul `ReactiveFormModule`. Importovaný modul umožní vytvářet nové instance objektu `FormControl()`, kterým definuje počáteční hodnotu reaktivního formuláře pomocí konstruktoru. Vytvořený reaktivní formulář se registruje v šabloně komponenty k ovládacímu prvku formuláře.

```
1     @Component({
2         standalone: true,
3         selector: 'app-name',
4         templateUrl:'
5             <label for="name">Name: </label>
6             <input id="name" type="text" [formControl]="name">'
7         styleUrls: ['./name-editor.component.css'],
8         imports: [ReactiveFormsModule],
9     })
10    export class NameComponent {
11        name = new FormControl('');
12    }
```

Sledovat změny hodnot reaktivního formuláře, lze pomocí pozorovatelného toku dat `valueChanges`. Aktuální stav reaktivního formuláře se získá dotazem na hodnotu vlastnosti `value`. Reaktivní formuláře umožňují programové aktualizování hodnoty vstupu pomocí metody `setValue()`. Tato metoda se používá při načítání dat z backendového rozhraní, kdy lze hodnoty reaktivního formuláře aktualizovat na tato data.

Reaktivní formuláře poskytují dva typy seskupení do několika souvisejících prvků. U prvního typu se vytváří skupina formulářů, která definuje přesný počet ovládacích prvků spravovaných společně. U druhého typu seskupení reaktivního formuláře se definuje pole formulářů, do kterých se dynamicky přidávají ovládací prvky. Pro získávání hodnot ovládacích prvků, se prochází skupina i pole formulářů pomocí identifikačního jména. Identifikační jméno je definováno u každého ovládacího prvku seskupení.

Každý reaktivní formulář nese určitá data o vstupu v šabloně, například, zda byla vstupní hodnota změněna, je-li validní nebo prázdná. Tyto hodnoty po seskupení platí pro celou skupinu nebo pole ovládacích prvků. Není tedy nutné při dotazování, zda je reaktivní formulář validní, vyhledávat ovládací prvek, ale lze se dotázat celého seskupení.

Pro generování seskupení existuje služba `FormBuilder`, která zjednodušuje definování složitých celků reaktivních formulářů.

Kapitola 3

Optimalizace postupu při psaní testů ve frameworku Angular

Tato kapitola popisuje postupy, jak provádět jednotkové a integrační testy ve frameworku Angular, a také vysvětluje jejich výhody a nevýhody.

3.1 Popis pomocné aplikace pro testování

V případě, že není použit vývoj řízenými testy (TDD), tak při testování webových aplikací je vhodné mít testovací subjekt. Testovacím subjektem může být funkční webová aplikace, ale i prototyp obsahující pouze určité části webové aplikace, které jsou testovány. Tyto části nemusí být plnohodnotné, může se jednat jen o částečnou implementaci. V každém případě tato částečná implementace musí obsahovat dostatečnou komplexnost pro testování dané funkcionality.

Části, které jsou primárně testovány neobsahují velké množství dat, ale obsahují data pro určitou množinu možných výsledků testů. Pokud se nejedná o testování vizualizace, není potřebné, aby všechny prvky testované části byli vykresleny správně podle vizuálního návrhu.

3.2 Metody pro psaní testů ve frameworku Angular

K psaní jednotkových testů a integračních testů, lze ve frameworku Angular [1] přistupovat dvěma způsoby:

1. Testování pomocí frameworku Jasmine bez využití syntaxe frameworku Angular – jedná se o přístup, kdy se při testování nevyužívají přidané výhody frameworku Angular a testuje se pouze kód programovacího jazyku TypeScript. Výhodou této metody je nutnost znalosti testovacího frameworku Jasmine. Může ji tedy využívat i programátor, který nikdy nestestoval kód ve frameworku Angular, ale testoval jiné kódy psané ve frameworcích postavených na programovacím jazyku JavaScript. Nevýhodami mohou být delší testovací specifikace, možnost volání ještě nevytvořené komponenty nebo delší vyhodnocovací čas testů.

```
1        // definování testu pro službu kalkulačky
2        describe('CalculatorService', () => {
3            let calculator: CalculatorService;
4
```

```
5           it('add two numbers', () => {
6               const result = calculator.add(2, 2);
7               expect(result).toBe(4);
8           })
9       }
```

Z kódu je patrné, že pro testování služby pro výpočty, je použit základní syntax frameworku Jasmine. Tou je testovací funkce `describe()` a v ní vyhodnocovací funkce `it()`. V testovací funkci je deklarována nová proměnná jako CalculatorService a z ní se volá funkce pro součet dvou čísel `add()`. Pomocí vyhodnocovací funkce je otestováno, zda se jedná o správný výsledek.

2. Testování pomocí frameworku Jasmine s využitím syntaxe frameworku Angular – naopak u postupu, který nevyužívá syntaxi frameworku Angular, je pro správné vykreslení testované komponenty v rámci testovací sady, potřebné komponentu deklarovat v modulu a navázat všechny vnitřní závislosti.

   V testovacím bloku `beforeEach` je použit testovací objekt `TestBed`, který správně nastaví modul frameworku Angular, do kterého se při konfiguraci modulu deklaruje testovaná komponenta. Pro jednotkové testy mohou být deklarovány například komponenty, služby nebo směrnice. Při integračních testech je deklarována testovací komponenta a jsou importovány moduly, směrnice nebo služby, na kterých je komponenta závislá. Než bude vykreslen deklarovaný obsah, je zkompilován metodou `compileComponents()`. Jakmile se komponenta vykreslí, obsahuje pouze statickou část HTML šablony. Všechny dynamické záznamy v HTML šabloně jsou spouštěny metodou `detectChanges()`.

   Vytvořená komponenta je uložena do proměnné `fixture` typu `ComponentFixture`. Tato proměnná drží instanci testované komponenty v takzvaném testovacím rámu. Testovací rám poskytuje pohodlné rozhraní k instanci testované komponenty a její HTML šabloně.

```
1       beforeEach(() => {
2
3       // konfigurace testovacího modulu
4       TestBed.configureTestingModule({
5
6           // deklarace testované komponenty
7           declarations: [RegistrationScheduleComponent],
8
9           // poskytnutí závislostí
10          providers: [HttpClient, HttpHandler],
11
12      // kompilace komponenty
13      }).compileComponents();
14
15      // vytvoření komponenty
16      fixture = TestBed.createComponent(
            RegistrationScheduleComponent);
17
18      // detekování dynamických změn HTML šablony
19      fixture.detectChanges();
20
21      // získání HTML šablony
22      debugElement = fixture.debugElement;
23    })
```

Výhodami v tomto případě mohou být rychlejší doba vyhodnocení testu, správné volání a konstruování komponent využívaných v jiných komponentách, zaručení izolace testu od reálného kódu nebo menší velikost specifikace. Hlavní nevýhodou tohoto způsobu testování je ta, že je zapotřebí programátor, který má potřebné dovednosti ke správnému napsání testu. Jelikož se jedná o specifickou dovednost s určitým zaměřením, není těchto programátorů mnoho.

3.3 Metody testování Angular komponenty s dětskou komponentou

Psaní testů pro rodičovské komponenty je obtížné z důvodu odkazů na dětské komponenty. Testovací soubor ve frameworku Angular při deklaraci testované komponenty zná HTML model, ve kterém jsou dětské komponenty použity, ale jelikož se snaží testy izolovat, tak je nepřidá do deklarace. Následkem je odkazování se na neznámý prvek v HTML modelu a špatné vyhodnocení testů.

Pokud se jedná o jednotkové testování (unit testing), je nutností při konfiguraci testovacího modulu přidat schéma.

```
schema: [ NO_ERROR_SCHEMA ]
```

Pomocí tohoto schéma se testovacímu souboru předá informace, která říká, že může ignorovat elementy v HTML modelu, které nezná a označit je za tak zvaný mrtvý kód. S tímto kódem se již na dále nepracuje. Jedná se o mělké vykreslování, kdy se nevykreslují dětské komponenty. [3]

Mezi jednotkovými testy a integračními testy existuje střední cesta a tou je práce s prázdnými vlastními potomky. Jedná se o falešnou komponentu, která má stejný selektor, vstupy i výstupy, ale nemá žádné závislosti a nemusí nic vykreslovat. Pro testování s dětskými komponentami se děti nahradí falešnými komponentami.

Do specifikace pro testování komponenty se deklaruje falešná komponenta, která požaduje, aby `FakeSubjectComponent` implementovala podmnožinu `SubjectComponents`.

```
1     // definování stejného selektoru
2     @Component({
3         selector: 'app-subject',
4         template: '',
5     })
6     // falešná třída s implicitním odkazem
7     class FakeSubjectComponent implements Partial<SubjectComponent> {
8         @Input()
9         public start = 0;
10
11        @Output()
12        public subjectChange = new EventEmitter<boolean>();
13    }
```

V testovací části je při konfiguraci modulu vložena falešná komponenta. Pokud tedy framework Angular narazí při testu na pravou komponentu, nahradí ji falešnou. Prvek zůstane prázdný, jelikož falešná šablona je také prázdná. Nastaví se pouze vstupy a výstupy falešné komponenty, které jsou identické se vstupy a výstupy pravé komponenty.

```
1     TestBed.configureTestingModule({
2         declarations: [TasksComponent, FakeSubjectComponent],
3         schemas: [NO_ERRORS_SCHEMA],
```

```
4        }).compileComponents();
```

Nalezení elementu v testovacím HTML modelu se neprovádí podle kaskádového selektoru, protože daná komponenta neexistuje v HTML modelu. K nalezení se používá směrnicový selektor (directive), který vrací falešnou komponentu. Pro přistup k falešné proměnné lze použít `componentInstance`, ale při použití je nutné přidat explicitní anotaci typu, protože neznáme typ falešné komponenty.

```
1     const subjectEl = fixture.debugElement.query(
2         By.directive(FakeSubjectComponent)
3     );
4     const subject: SubjectComponent = subjectEl.componentInstance;
```

Výhodou tohoto postupu je možnost nalezení vykreslené, i když prázdné komponenty v HTML modelu. Práce s instancí komponenty je intuitivnější, než práce s `debugElement`. Díky instanci komponenty, lze lépe číst vstupy a výstupy komponenty. Nevýhodou ručního falšování je manuální práce a čas, který testování stojí. Nejen že testování je zdlouhavé, ale také náchylné na případné chyby ze strany programátora.

Pokročilejším způsobem jak zfalšovat komponenty, je využití rozsáhlé knihovny ng-mock. Ng-mock dokáže vytvářet falešné komponenty a pokud je při volání očekávána původní komponenta vrací falešnou komponentu, která se podobá originálu. Místo vytvoření falešné komponenty na začátku specifikace, se vloží do deklarace komponenty v konfiguraci obalená podoba komponenty ve tvaru `MockComponent(SubjectComponent)`. Pokud bude v testu požadována pravá komponenta `SubjectComponent`, test dostane falešnou komponentu, se kterou může libovolně pracovat.

Vše, co platí pro pravou komponentu platí i pro podvrh, není tedy nutné při přepisování komponenty upravovat i falešnou komponentu, knihovna ng-mock to upraví. Ng-mock je rozsáhlá knihovna, která umožňuje zjednodušit kód pro testování, a to nejen u komponent, ale i u služeb, směrnic nebo modulů.

3.4 Metody testování Angular služby (service)

Služba ve frameworku Angular je návrhový vzor jedináček (singleton). Při definici nemá přesné označení, jako například komponenta `@Component` nebo směrnice `@Directive`. Služba značená jako nahraditelná `@Injectable` a není přesně definováno, zda to musí být třída nebo funkce.

Pokud je testována služba, která neodkazuje na jiné služby nebo objekty, lze tuto službu testovat bez definice testovacího objektu `TestBed`. Pro každou testovací sadu je v bloku `beforeEach()` vytvořená nová instance dané služby, kterou testovací sada testuje. Tato možnost je vhodná pro služby, které předávají vnitřní stav různým komponentám webové aplikace.

Obsahuje-li služba závislost na jinou službu nebo třídu, je zdlouhavé ji definovat pomocí nové instance. Jednodušší možností je definice v testovacím objektu `TestBed` pro nastavení testovacího modulu. Do testovacího modulu se importují třídy, na kterých je služba závislá a samotná služba je poskytována testovacím modulem. [2]

```
1     TestBed.configureTestingModule({
2         imports: [HttpClientTestingModule],
3         providers: [WorldClockService],
4       });
```

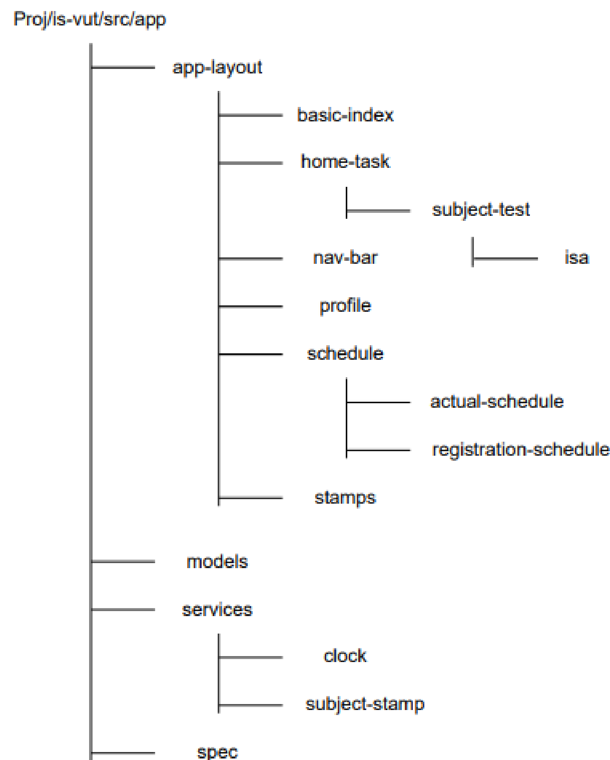
Kapitola 4

Implementace testů

Tato kapitola je věnována pomocné aplikaci, je zde vysvětlena pomocná testovací sada, a nakonec jsou zde popsány implementace jednotlivých testů pro určité testovací scénáře.

4.1 Pomocná testovací aplikace

Pro možnost testování ve frameworku Angular vznikla nová webová aplikace simulující určité funkcionality. Tato webová aplikace se nazývá is-vut a částečně simuluje informační systém pro studenty. Na obrázku níže, je vyobrazena struktura adresáře webové aplikace:



Obrázek 4.1: Struktura adresáře projektu

Všechny komponenty, které webovou aplikaci tvoří jsou uloženy v adresáři app-layout. Každá komponenta je obalena do svého adresáře, který obsahuje čtyři soubory:

- component.html – je soubor definující HTML šablonu komponenty
- component.scss – je soubor kaskádových stylů pro HTML šablonu komponenty
- component.ts – je soubor obsahující základní logiku komponenty a její definici
- component.spec.ts – je testovací specifikace pro komponentu

Pro správné navázání grafické vizualizace webové aplikace, jsou komponenty volány v HTML šablonách jiných komponent. Tato kaskáda volání tvoří celou webovou aplikaci.

Základem webové aplikace je hlavní stránka basic-index, která studenta informuje o novinkách. Na hlavní stránce se nachází navigace uložená v adresáři nav-bar. Navigace odkazuje na ostatní záložky webové aplikace.

První záložkou webové aplikace je Rozvrh, na které jsou vyobrazena dvě tlačítka. První tlačítko slouží k zobrazení aktuálního rozvrhu a druhé tlačítko umožňuje studentům zaregistrovat se k rozvrhu. Tato záložka je zařazena v adresáři schedule. Záložka Úkoly je uložena ve složce home-task a dává studentovi možnost plnit úkoly na určitý počet pokusů. Záložka Známky umožňuje studentovi shlédnout svůj dosavadní prospěch. Tato záložka je uložena v adresáři stamps. Poslední záložka webové aplikace is-vut, ke které je umožněn přístup je záložka Profil. Ta simuluje správu a nastavení změn přihlašovacích údajů. Je zařazena v adresáři profile a současně tento adresář obsahuje pomocný soubor pro testování, profile.spec-helper.ts.

Ostatní adresáře ve složce app, nejsou již tak rozsáhlé, nicméně je vhodné je zmínit, jelikož mají důležitý vliv na chod webové aplikace. Ve složce models lze nalézt modely dat pro komunikaci mezi komponentami. Ve složce services jsou dva podadresáře clock a subject-stamps, které obsahují služby poskytované komponentám webové aplikace. Oba tyto adresáře obsahují dva soubory:

- service.ts – jedná se o soubor definující logiku služby
- service.spec.ts – je testovací specifikace pro službu

V posledním adresáři se nachází soubor spec-help.ts. Jedná se o skript, ve kterém jsou definovány pomocné funkce pro testování komponent.

Ve složce app je uložena základní komponenta, na které je celá webová aplikace postavena, a zároveň jsou zde uloženy moduly pro deklaraci komponent a pro možnost přesměrování se na určitou stránku ve webové aplikaci.

4.2 Pomocná testovací sada

V pomocné testovací sadě spec-helper.ts jsou definovány funkce pro zjednodušení testování komponent.

První funkce `findEl()` musí být volána se dvěma parametry. První parametr je testovací komponenta `fixture` a druhý parametr je řetězec `testId`. Funkce vrací nalezený element v testovací šabloně `fixture.debugElement`, kde vyhledává element s atributem `testId` pomocí kaskádového selektoru.

```
1   export function findEl<T>(fixture: ComponentFixture<T>, testId:
        string): DebugElement {
2       // vrácení hledaného elementu
3       return fixture.debugElement.query(
4           By.css(`[test-id=${testId}]`)
5       );
6   }
```

Další podobná funkce je `findComponent()`, která také požaduje při volání dva parametry. Prvním parametrem je opět testovací komponenta `fixture` a druhým parametrem je `selektor` nesoucí název hledané komponenty. Tato funkce vrací nalezenou komponentu v testovací HTML šabloně.

```
1   export function findComponent<T>(fixture: ComponentFixture<T>,
        selector: string) {
2       // vrácení hledané komponenty
3       return fixture.debugElement.query(By.css(selector));
4   }
```

Funkce `click()` s parametry `fixture` a `testId` vyhledá v testované HTML šabloně element a následně vyvolá na tomto elementu falešnou událost kliknutí.

```
1   export function click<T>(fixture: ComponentFixture<T>, testId:
        string) {
2       const element = findEl(fixture, testId);
3
4       // zfalšování události kliknutí
5       const event = makeClickEvent(element.nativeElement);
6
7       // vyvolání události kliknutí
8       element.triggerEventHandler('click', event);
9   }
```

Aby bylo možné vyvolat falešné kliknutí, je potřeba zfalšovat událost `MouseEvent`. Tohoto zfalšování lze docílit funkcí `makeClickEvent()` s parametrem `target`, který nese odkaz na element v testovací HTML šabloně. Pro tento element vrací falešnou událost kliknutí.

```
1   export function makeClickEvent(target: EventTarget): Partial<
        MouseEvent> {
2
3       // vrácení falešné události kliknutí
4       return {
5           preventDefault(): void {},
6           stopPropagation(): void {},
7           stopImmediatePropagation(): void {},
8           type: 'click',
9           target,
10          currentTarget: target,
11          bubbles: true,
12          cancelable: true,
13          button: 0
14      };
15  }
```

4.3 Testování jednoduché Angular komponenty

V testovací aplikaci is-vut, je definována komponenta v adresáři registration-schedule. Jedná se o jednoduchou komponentu, která znázorňuje registraci předmětů, u kterých jsou použity

dvě funkcionality. První funkcionalitou je počítání celkového počtu přihlášených předmětů daného uživatele. A v případě druhé funkcionality se jedná o přičítání a odčítání celkového počtu přihlášených studentů k danému předmětu.

Testovací soubor s názvem registration-schedule.spec.component.ts obsahuje základní implementaci testování ve funkci `describe()`. V této funkci je využita další funkce, s názvem `beforeEach()`, která napomáhá ke správné definici komponenty před proběhnutím každého testu.

Testy jsou psány pomocí funkcí `it()`, které definují daný test a provádí jeho vyhodnocení. Je zde definován základní test, který testuje, zda je daná komponenta správně vložena do modulu a správně vykreslena.

```
1     it('Can be create', () => {
2         expect(component).toBeTruthy();
3       });
```

Pro počítání celkového počtu předmětů, jsou v ukázce použity dva testy. První pro testování správného přičítání k počtu předmětů při volbě předmětu a druhý pro testování správného odečítání od počtu předmětů při opuštění předmětu.

```
1     it('Increments the counter for subjects', () => {
2
3         // výběr tlačítka přihlásit v HTML modelu komponenty
4         const subscribeButton = debugElement.query(
5           By.css('[test-id="subscribe-button"]')
6         );
7
8         // vyvolání události kliknutí na tlačítko
9         subscribeButton.triggerEventHandler('click', null);
10
11        // přegenerování provedených změn v HTML modelu komponenty
12        fixture.detectChanges();
13
14        // výběr počítadla přihlášených předmětů
15        const counterOutput = debugElement.query(
16          By.css('[test-id="counter"]')
17        );
18
19        // vyhodnocení, zda bylo počítadlo navýšeno
20        expect(counterOutput.nativeElement.textContent).toBe('1');
21      });
```

Při testování je vybráno tlačítko z `debugElement`, což je v tomto případě HTML model testované komponenty, pomocí selektoru kaskádových stylů, který nalezne element s atributem `test-id="subscribe-button"`. Poté je na tomto tlačítku vyvolána událost simulující kliknutí a volána funkce `detectChanges()`, která odchytí změny, které nastali v HTML modelu. Stejným způsobem, jako bylo hledáno tlačítko pro přihlášení, je nalezeno počítadlo aktuálního počtu přihlášených předmětů. Na tomto elementu, je provedeno finální vyhodnocení, zda se jeho hodnota zvedla o jedna. Slovní logikou je možné tento postup popsat, jako:

> Testování, zda po kliknutí na tlačítko přihlásit, se navýší hodnota počítadla přihlášených předmětů právě o jedna.

Druhý test provádí velmi podobnou operaci. Testuje se, zda se při zmáčknutí tlačítka odhlásit odečte od hodnoty počítadla předmětů právě jedna. Změna nastává u použití

tlačítek, protože tlačítko odhlásit se zobrazí až v momentě po kliknutí na tlačítko přihlásit. Proto je v tomto testu nalezeno tlačítko přihlásit, následně vyvolaná událost kliknutí na něj a odchycení změn. Poté se provede vyhodnocení, zda byla hodnota přičtena a celý proces se opakuje. Nicméně tentokrát se hledá tlačítko odhlásit pod atributem `test-id="unsubscribe-button"`. Po zavolání události kliknutí, se opět volá funkce pro detekci změn a vyhodnocuje se, zda počet přihlášených předmětů klesl o jedna.

```
1      it('Decrements the counter for subjects', () => {
2
3          // nalezení tlačítka přihlásit a vyvolaná událost kliknutí
4          const subscribeButton = findEl(fixture, 'subscribe-button');
5          subscribeButton.triggerEventHandler('click', null);
6          fixture.detectChanges();
7
8          // nalezení počítadla a vyhodnocení, zda byla hodnota navýšena
9          const counterOutput = findEl(fixture, 'counter')
10         expect(counterOutput.nativeElement.textContent).toBe('1');
11
12         // nalezení tlačítka odhlásit a vyvolaná událost kliknutí
13         const unsubscribeButton = findEl(fixture, 'unsubscribe-button');
14         unsubscribeButton?.triggerEventHandler('click', null);
15         fixture.detectChanges();
16
17         // vyhodnocení, zda byla hodnota počítadla předmětů snížena
18         expect(counterOutput.nativeElement.textContent).toBe('0');
19       });
```

Pro nalezení elementu z HTML modelu komponenty je zde použita funkce `findEl()`. Tato funkce je definována v pomocném souboru pro testování spec-helper.ts.

Aby byla správně otestována celá funkcionalita, je definovaný další test, který vyhodnocuje, zda se tlačítko přihlásit změní na tlačítko odhlásit. V HTML modelu se nalezne tlačítko přihlásit pomocí `test-id="subscribe-button"`, vyhodnotí se, zda je opravdu vykresleno a vyvolá se událost kliknutí. Po kliknutí se odchytí změny v HTML modelu a hledá tlačítko odhlásit pod atributem `test-id="unsubscribe-button"`. Nakonec se vyhodnotí, zda je toto tlačítko vykresleno v HTML modelu komponenty.

```
1      it('Changing subscribe button for subjects', () => {
2
3          // nalezení tlačítka přihlásit a vyhodnocení, zda se vykreslí
4          const subscribeButton = findEl(fixture, 'subscribe-button');
5          expect(subscribeButton).toBeTruthy();
6
7          // vyvolání události kliknutí na tlačítko přihlásit
8          subscribeButton.triggerEventHandler('click', null);
9          fixture.detectChanges();
10
11         // nalezení tlačítka odhlásit a vyhodnocení, zda se vykreslí
12         const unsubscribeButton = findEl(fixture, 'unsubscribe-button');
13         expect(unsubscribeButton).toBeTruthy();
14       })
```

Komponenta obsahuje funkcionalitu přičítání a odčítání celkového počtu studentů přihlášených pro daný předmět, reprezentována jako výstup komponenty v podobě modelu v programovacím jazyce JavaScript. Tento model obsahuje hodnotu `numberOfStudents$`. Jedná se o pozorovaný objekt s počáteční hodnotou (BehaviorSubject), který sleduje změny hodnot a reaguje na ně. Jelikož se jedná o výstup, který je potenciálně sdílen s jinými komponentami, je nutné ho testovat.

Test obsahuje počáteční hodnotu, číslo, které je definováno jako počet již přihlášených studentů. Jde o pozorovatelnou hodnotu, tudíž musí test poslouchat na této hodnotě. Odposlech je proveden, pomocí metody `subscribe()`, která je vykonána pokaždé, odchytí-li změnu v pozorované hodnotě. Pokud tedy nastane změna, zapíše se změněná hodnota do proměnné `newNumber`. Test pokračuje nalezením tlačítka přihlásit a vyvoláním události kliknutí. U tohoto postupu není potřeba zaznamenávat změny v HTML modelu, jelikož test dále nepracuje s HTML modelem, ale testuje zmíněnou hodnotu pozorovaného objektu. Vyhodnotí tedy, zda-li se do pozorované hodnoty opravdu zapsala hodnota o jedno větší.

```
1     it('Output testing subscribe', () => {
2         // počáteční hodnota a deklarace nové hodnoty
3         let initialNumber = 3;
4         let newNumber: number = 0;
5
6         // odposlech pozorovaného objektu a zápis do nové hodnoty při
             odchytu změn
7         component.subjects[0].numberOfStudents$.subscribe((counter) => {
8             newNumber = counter;
9         });
10
11        // nalezení tlačítka přihlásit a vyvolání události kliknutí
12        const subscribeButton = findEl(fixture, 'subscribe-button');
13        subscribeButton.triggerEventHandler('click', null);
14
15        // vyhodnocení, zda byla hodnota navýšena
16        expect(newNumber).toBe(initialNumber + 1);
17    })
```

Podobný test je proveden pro správné odečítání pozorované hodnoty, kde jsou provedeny stejné operace, až na jednu změnu. Tou změnou je, po vyvolání události kliknutí na tlačítku přihlásit, zaznamenání změny do HTML modelu, z důvodu následného hledání tlačítka odhlásit, které do teď nebylo vyobrazeno. Jakmile je nalezeno, vyvolá se událost kliknutí na tomto tlačítku a vyhodnotí, zda se nová hodnota rovná počáteční hodnotě. V tomto testu není potřeba znovu vyhodnocovat navýšení hodnoty po vykonání události kliknutí na tlačítko přihlásit, jelikož tato funkcionalita je již otestována v jiném testu.

```
1     it('Output testing unsubscribe', () => {
2         /* ...Stejný kód jako při předchozím testu... */
3
4         // detekce změn v HTML modelu
5         fixture.detectChanges();
6
7         // nalezení tlačítka odhlásit a vyvolání události kliknutí
8         const unsubscribeButton = findEl(fixture, 'unsubscribe-button');
9         unsubscribeButton.triggerEventHandler('click', null);
10
11        // vyhodnocení, zda je hodnota rovna počáteční hodnotě
12        expect(newNumber).toBe(initialNumber);
13      })
```

4.4 Testování Angular komponenty s dětskou komponentou

V adresáři subject-test se nachází komponenta subject-test.component. Tato komponenta odkazuje v HTML modelu na dětskou komponentu isa.component, která simuluje vyplnění testu na určitý počet pokusů.

Dětská komponenta má jeden vstup, který definuje kolik možných pokusů má student pro vyplnění testu. A jeden výstup, který udává, zda byl daný test již vyplněn. Jelikož se jedná o simulaci funkcionality, test je automaticky vyplněn po kliknutí na tlačítko spustit test. Počet potřebných kliknutí je vždy rozdílný, jelikož je definovaný jako náhodné číslo od nuly do počtu možných pokusů. Po splnění testu se tlačítko stává neplatné a nelze na něj kliknout. Současně je rodičovské komponentě posláno potvrzení, že test byl splněn. Rodičovská komponenta na to reaguje podbarvením na zelenou barvou z dosud červené barvy k demonstraci správnosti.

Aby byl test validní, musí být daná komponenta otestována, zda-li je vyobrazena v HTML modelu. Nicméně, jelikož se jedná o rodičovskou komponentu, test musí obsahovat i vyhodnocení, jestli je vykreslena dětská komponenta. Proto se musí pomocí kaskádového selektoru nalézt komponenta s názvem app-isa a poté ověřit její vyobrazení.

```
1     it('Render isa component', () => {
2
3         // nalezení app-isa komponenty
4         const isaTest = fixture.debugElement.query(By.css('app-isa'));
5         expect(isaTest).toBeTruthy();
6       })
```

Vstupní data do dětské komponenty jsou testována opětovným nalezením komponenty app-isa. Tentokrát je použita funkce `findComponent()`, která podobně jako `findEl()` je vložena z pomocného testovacího scriptu spec-help.ts, a která nalezne komponentu v testovacím HTML modelu. Při vyhodnocení je proveden dotaz, zda nalezená komponenta obsahuje proměnnou `startTest` a zda tato proměnná nabývá hodnoty posílané z rodičovské komponenty.

```
1     it('Input binding on isa component', () => {
2
3         // nalezení app-isa komponenty pomocí findComponent()
4         const isaTest = findComponent(fixture, 'app-isa');
5
6         // vyhodnocení, zda komponenta obsahuje počáteční hodnotu
7         expect(isaTest.properties['startTest']).toBe(8);
8       })
```

V poslední řadě je potřeba testovat výstupní data z dětské komponenty a správné reakce na ně. Do testu je vložen prvek `ElementRef`, který odkazuje na HTML model dané komponenty. Tento objekt umožňuje úpravu HTML modelu bez dlouhého hledání elementu v celé aplikaci. Zde je využit, pro špehování na metodě `add()` v objektu `classList` s možností odchycení dat, se kterými je tato metoda volána. Test dále volá metodu `statusHandle` s hodnotou `true`. Tato pravdivostní hodnota je označením, že uživatel úspěšně splnil test. Vyhodnocení probíhá na základě, zda se špehovaná metoda volá s hodnotou done v reakci pravdivostní hodnotu `true`.

```
1     it('Output binding on isa component for event emiter is true', () =>
          {
2
3         // vložení element ref do testovací sady
4         const elementRef = fixture.debugElement.injector.get(ElementRef)
              ;
5
6         // špehování na metodě add v objektu classList
7         const statusHandleSpy = spyOn(elementRef.nativeElement.
              firstChild.classList, 'add');
8
```

```
9          // volání metody s hodnotu true
10         component.statusHandle(true);
11
12         // vyhodnocení, zda je metoda volána s parametrem done
13         expect(statusHandleSpy).toHaveBeenCalledWith('done');
14       })
```

Pokud by do komponenty přibyla další funkcionalita, musí se rozšířit testování o tuto funkcionalitu. Momentálně je však dostačující správné chování špehovaného objektu, pokud není výstupní hodnota z dětské komponenty jiná, než pravdivostní hodnota `true`.

4.5 Testování Angular komponenty se službou

Služby (services) jsou využívány pro přenesení logiky mimo komponentu, z důvodu možného kontrolování stavů pro více komponent. Tato funkcionalita je implementována ve skriptu stamps.component.ts v adresáři stamps. Této komponentě je poskytnuta služba subject-marks.service.ts uložená v adresáři services.

Hlavní účel komponenty je vykreslit předměty s bodovým ohodnocením prací v předmětu. Předměty jsou do komponenty posílány ze služby, která definuje simulační model, který lze získat v reálném projektu ze serverové části webové aplikace. Model předmětů je objekt s atributy `name` a `tasks`, kdy `name` popisuje název předmětu. `Tasks` je pole objektů s atributy `name`, tentokrát pro popis hodnocené části předmětu, a `marks`, které udávají dosažený počet bodů pro danou hodnocenou část.

```
1     {
2         name: string,       // název předmětu
3         tasks: [{
4             name: string,   // popis hodnocené části předmětu
5             marks: number   // hodnocení části předmětu
6         }]
7     }
```

Jednotkové testování komponenty se službou vyžaduje nahrazení služby falešnou službou, z důvodu dodržení izolace dat u jednotkových testů. Pro jednotkový test je nutné definovat celý model předmětů s testovacími hodnotami na začátek testu, a zároveň je zde deklarována nová proměnná `fakeSubjectMarksService`. Tato proměnná je ve funkci `beforeEach()` využita pro vytvoření špehovaného objektu. Pokud by zde nebyl vytvořen špehovaný objekt, tak by se museli špehovat každé metody falešné služby zvlášť. Framework Jasmine, to usnadňuje právě vytvořením špehovaného objektu, který má možnost kontrolovat vstupní a výstupní data na všech metodách.

Kontrola u této falešné služby bude probíhat na metodě `getSubject`, která vrací v podobě pozorovatelného datového toku (Observable) celé pole modelů předmětů. Při konfiguraci testování je při definici poskytování provedeno nahrazení reálné služby za falešnou službu. V poslední řadě je testováno volání metody falešné služby, která vrací pole modelů předmětů.

```
1     describe('StampsComponent: Unit test', () => {
2
3         // definování pole modelů předmětů pro testování
4         const subjects: { name: string, tasks: {name: string, marks:
              number}[] }[] = [
5             {   name: 'Tvorba webových stránek',
6                 tasks: [    {name: "Projekt 1", marks: 20  },
7                             {name: "Projekt 2", marks: 20  },
```

```
8                               {name: "Zkouška", marks: 48  }
9                           ]
10           },
11           {   name: 'Základy elektrotechniky',
12               tasks: [    {name: "Projekt 1", marks: 20  },
13                           {name: "Cvičení", marks: 30  },
14                           {name: "Zkouška", marks: 50  }
15                       ]
16           },
17           {   name: 'Vizualizace dat v pythonu',
18               tasks: [    {name: "Cvičení", marks: 30  },
19                           {name: "Zkouška", marks: 70  }
20                       ]
21           }
22       ];
23
24       let component: StampsComponent;
25       let fixture: ComponentFixture<StampsComponent>;
26
27       // deklarace falešné proměnné
28       let fakeSubjectMarksService: SubjectMarksService;
29
30       beforeEach(async () => {
31
32           // definování špehovaného objektu na službě
33           fakeSubjectMarksService = jasmine.createSpyObj<
                 SubjectMarksService>(
34               'SubjectMarksService',
35               {
36               getSubjects: of(subjects)
37               }
38           );
39
40         await TestBed.configureTestingModule({
41           declarations: [StampsComponent],
42
43           // poskytování falešné služby místo reálné
44           providers: [{   provide: SubjectMarksService,
45                           useValue: fakeSubjectMarksService
46                       }]
47         }).compileComponents();
48
49         fixture = TestBed.createComponent(StampsComponent);
50         component = fixture.componentInstance;
51         fixture.detectChanges();
52       });
53
54       // testovani, zda se komponenta vykresli
55       it('should create', () => {
56         expect(component).toBeTruthy();
57       });
58
59       // testování, zda je volána metoda falešné služby
60       it('Call getSubject', () => {
61         expect(fakeSubjectMarksService.getSubjects).toHaveBeenCalled()
             ;
62       })
63     });
```

Testování komponenty pomocí integračních testů je daleko jednodušší, jelikož není potřeba dodržet izolaci testu a je možné tedy poskytovat reálnou službu a testovat reálná data.

Na začátku testu je opět definována nová proměnná `subjectMarksService`, avšak nejedná se již o falešnou službu, nýbrž o reálnou službu. Pro správné fungování implementace je služba přidána do konfigurace, aby byla poskytnuta pro daný test a zároveň vložena do proměnné `subjectMarksService`.

Při testování, zda metoda `getSubjects()` opravdu poskytuje správná data, se v testu definuje dočasná proměnná `testSubjects` s prázdným polem modelů předmětů. Jelikož metoda `gatSubjects()` ze služby vrací pozorovatelný tok dat (Observable), test se na něj zaposlouchá a do pomocné proměnné `testSubjects` uloží vrácené pole modelů předmětů. Při vyhodnocování testu je toto pole porovnáváno s polem modelů předmětů v komponentě.

```
1  describe('StampsComponent: Integration test', () => {
2    let component: StampsComponent;
3    let subjectMarksService: SubjectMarksService
4    let fixture: ComponentFixture<StampsComponent>;
5
6    beforeEach(async () => {
7      await TestBed.configureTestingModule({
8        declarations: [StampsComponent],
9        providers: [SubjectMarksService]
10     }).compileComponents();
11     subjectMarksService = TestBed.inject(SubjectMarksService);
12     /* ...další deklarace... */
13   });
14
15   it('should load subjects on initialization', () => {
16     // definování prázdného pole modelu
17     let testSubjects: {name: string, tasks: { name: string, marks:
          number}[]}[] = [];
18
19     // poslouchání a zápis modelu předmětů
20     subjectMarksService.getSubjects().subscribe(subjects => {
21       if (subjects) {
22         testSubjects = subjects;
23       }
24     });
25
26     // vyhodnocení správnosti testu
27     expect(component.subjects).toBe(testSubjects);
28   })
29 });
```

4.6 Testování Angular služeb (service)

Komponenta stamps.component.ts využívá službu subject-marks.service.ts. Tato služba poskytuje pozorovatelný tok dat (Observable), jako pole modelů předmětů. Model se používá pro vykreslování předmětů a bodového hodnocení za jednotlivé úkoly v předmětu.

Při testování je definována proměnná s typem dané služby, do které se uloží instance třídy služby. Při jednotkovém testování není konfigurována testovací třída `TestBed`, tak jako u jiných testů. Konfigurace této třídy se používá u složitějších služeb, které potřebují deklarovat jiné služby nebo funkcionality.

Jelikož služba poskytuje pozorovatelný tok dat pomocí metody `getSubjects()`, testuje se, zda služba vrací pole správných modelů předmětů. Test prvně definuje pole testovacích modelů předmětů do proměnné `testSubjects` a pomocnou proměnnou pro uložení dat z pozorovatelného toku dat. Poté vyvolá metodu `getSubjects()` a zaznamená data do pomocné proměnné. Nakonec vyhodnotí, zda jsou data stejná, jako pole testovacích modelů předmětů.

```
1     describe('SubjectMarksService', () => {
2         // definování proměnné s typem služby
3         let subjectMarksService: SubjectMarksService;
4
5         beforeEach(() => {
6             // vytvoření nové instance služby
7             subjectMarksService = new SubjectMarksService()
8         });
9
10        it('return subject', () => {
11            // testovací model předmětů
12            const subjects: { name: string, tasks: {name: string, marks:
                  number}[] }[] = [
13                {   name: 'Tvorba webových stránek',
14                    tasks: [    {name: "Projekt 1", marks: 20  },
15                                {name: "Projekt 2", marks: 20  },
16                                {name: "Zkouška", marks: 48  }
17                            ]
18                },
19                {   name: 'Základy elektrotechniky',
20                    tasks: [    {name: "Projekt 1", marks: 20  },
21                                {name: "Cvičení", marks: 30  },
22                                {name: "Zkouška", marks: 50  }
23                            ]
24                },
25                {   name: 'Vizualizace dat v pythonu',
26                    tasks: [    {name: "Cvičení", marks: 30  },
27                                {name: "Zkouška", marks: 70  }
28                            ]
29                }
30            ];
31
32            // deklarace pomocné proměnné
33            let servicesSubjects: { name: string, tasks: {name: string,
                 marks: number}[] }[] = []
34
35            // zápis dat z pozorovaného objektu
36            subjectMarksService.getSubjects().subscribe((subjects) => {
37                if (subjects) {
38                servicesSubjects = subjects;
39                }
40            });
41
42            // vyhodnocení
43            expect(servicesSubjects).toEqual(testSubjects);
44        })
45    });
```

V adresáři services je služba world-clock.service.ts. Služba poskytuje informace o čase a datu daného dne. Tato data získává pomocí HTTP GET požadavku posílaným na URL

http://worldtimeapi.org/api/timezone/Europe/Prague. Jedná se o veřejné API, není proto nutné při posílání požadavku přikládat do hlavičky klíč pro ověření.

V tomto případě testovací soubor obsahuje rozšíření o proměnnou `httpFake` s typem `HttpTestingControler`, který umožňuje očekávat požadavky a specifikovat jaké odpovědi by měli být vráceny. V tomto testu je použit objekt `TestBed`, při jehož konfiguraci je importován testovací modul `HttpClientTestingModule`, který umožňuje testováni interakce se serverem.

```
1     describe('WorldClockService', () => {
2         let worldClockService: WorldClockService;
3 
4         // definice HTTP testovacího kontroleru
5         let httpFake: HttpTestingController;
6 
7         beforeEach(() => {
8 
9             // použití testovacího objektu
10            TestBed.configureTestingModule({
11
12                // importování testovacího modulu
13                imports: [HttpClientTestingModule],
14                providers: [WorldClockService]
15            });
16        });
17
18        /* ...pokračování testu... */
```

Před vyhodnocením testovacích sad je definována funkce `afterEach()`, která pomocí `htttpFake.verify()` kontroluje, zda nejsou volány další nepoužité HTTP požadavky.

První testovací sada obsahuje dvě vyhodnocení. První vyhodnocení slouží ke zjištění, zda se opravdu jedná o HTTP GET požadavek. Test pošle falešný HTTP požadavek na definovanou URL adresu. Po té vyhodnotí, zda se opravdu jedná o GET požadavek a vrátí odpověď s falešným datovým modelem. Falešný datový model je definován na začátku testu.

```
1     // poslání falešného HTTP požadavku
2     const req = httpFake.expectOne('http://worldtimeapi.org/api/timezone
          /Europe/Prague');
3 
4     // vyhodnocení, zda se jedná o HTTP GET požadavek
5     expect(req.request.method).toBe('GET');
6 
7     // vrácení falešného modelu
8     req.flush(fakeResponse);
```

Druhé vyhodnocení kontroluje, zda jsou data získána z metody služby identická s falešným datovým modelem. V testu je volána metoda služby `getUtcTime()`, která vrací pozorovatelný tok dat, ze kterého je získán datový model. A ten je porovná s falešným datovým modelem.

```
1     it('should retrieve UTC time', () => {
2         const fakeResponse = {
3             /* ... falešný datový model... */
4         };
5 
6         // pomocná proměnná
7         let testData: any;
8 
9         // volaná metoda služby a pozorovaný datový tok
```

```
10          worldClockService.getUtcTime().subscribe((time) => {
11            testData = time;
12          });
13
14          // vyhodnocení, zda je vrácen správný datový model
15          expect(testData).toEqual(fakeResponse);
16      });
```

Následující testovací sada vyhodnocuje, zda-li volání HTTP GET požadavku je chybné a odpověď serveru je `error`. Proto jsou definovány čtyři nové proměnné:

- `status` – udává číslo chyby vráceného HTTP GET požadavku
- `statusText` – udává název chyby vráceného HTTP GET požadavku
- `errorEvent` – definuje novou instanci třídy `ErrorEvent()`
- `actualError` – deklaruje proměnnou typu `HttpErrorResponse`, pro možnost zaznamenání chyby vzniklé při HTTP GET požadavku

Test dále volá metodu služby `getUtcTime()`, a současně na vráceném toku dat pozoruje, zda je vyvolána další hodnota toku chyba, anebo zda byl pozorovatelný tok úspěšně ukončen. Není-li vyvolána chyba během pozorování datového toku, je test vyhodnocen nesprávně, pokud ale chyba vyvolána je, zapíše se do proměnné `actualError`. Test dále provádí falešný HTTP GET požadavek a vrací falešnou chybovou hodnotu. Tato chybová hodnota by měla být zachycena v pozorovaném datovém toku. Pro úspěšnost testu musí být provedeny tři vyhodnocení. První vyhodnocení kontroluje, zda je chyba stejná jako falešná chyba, druhé vyhodnocení kontroluje, zda má stejné chybové číslo, a třetí kontroluje, zda má stejný chybový popisek.

```
1       it('get error from request', () => {
2
3           // deklarace proměnných
4           const status = 500;
5           const statusText = 'Server error';
6           const errorEvent = new ErrorEvent('API error');
7           let actualError: HttpErrorResponse | undefined;
8
9           // pozorování datového toku
10          worldClockService.getUtcTime().subscribe(
11            () => { fail('Do not call next value'); },
12            (error) => { actualError = error; },
13            () => { fail('Do not call complete value'); }
14          );
15
16          // volání falešného HTTP GET požadavku
17          httpFake.expectOne('http://worldtimeapi.org/api/timezone/Europe/
                Prague').error(
18            errorEvent,
19            {status, statusText}
20          );
21
22          // vyhození chyby, pokud není pozorován error
23          if (!actualError) {
24            throw new Error('Error needs to be defined');
25          }
26
```

```
27          // vyhodnocení
28          expect(actualError.error).toBe(errorEvent);
29          expect(actualError.status).toBe(status);
30          expect(actualError.statusText).toBe(statusText);
31        })
32      });
```

4.7 Testování reaktivních formulářů

V adresáři profile je definována komponenta profile.component.ts. Tato komponenta simuluje zobrazení profilu, ale především zobrazení reaktivních formulářů v profilu. Implementovány jsou tři vstupy formuláře. První vstup umožňuje změnit jméno profilu, druhý vstup slouží ke změně emailu a třetí pro změnu hesla. Reaktivní formulář slouží pouze pro simulaci, proto je veškerá část obsahující logiku kódu v komponentě profile.component.ts.

Každý z těchto vstupů reaktivního formuláře je vyžadován, není tedy možné uložit ho celý bez vyplnění. Zároveň jméno nesmí být delší než padesát znaků a email musí obsahovat zavináč. Pokud email neobsahuje tečku pro oddělení jména schránky od domény, není to chyba, jelikož může nastat situace, kdy tester využije webovou aplikaci lokálně a použije lokální email. Pokud je vstup reaktivního formuláře nevalidní je označen červeným rámečkem.

Pro testování je použit soubor profile.spec-helper.ts, ve kterém jsou definovány různé testovací hodnoty, které je možné vložit do reaktivního formuláře. Do specifikace, při konfiguraci testovacího modulu, je importován nový modul pro definici reaktivních formulářů `ReactiveFormModule`.

V testovacím dokumentu je definována pomocná funkce `fillForm()`, která má parametry `name`, `email`, `password`. Tyto parametry se vloží do vstupního pole reaktivního formuláře, který je nalezen pomocnou funkcí `findEl()`. Po vložení do každého vstupního pole reaktivního formuláře je vytvořena událost odeslání, která odešle data do komponenty.

```
1     const fillForm = (name: string, email: string, password: string) =>
         {
2
3         // nalezení vstupů a vložení dat
4         const nameField = findEl(fixture, 'name').nativeElement;
5         nameField.value = name;
6         const emailField = findEl(fixture, 'email').nativeElement;
7         emailField.value = email;
8         const passwordField = findEl(fixture, 'password').nativeElement;
9         passwordField.value = password;
10
11        // vytvoření události
12        const event = document.createEvent('Event');
13        event.initEvent('input', true, false);
14
15        // použití události pro každý vstup formuláře
16        nameField.dispatchEvent(event);
17        emailField.dispatchEvent(event);
18        passwordField.dispatchEvent(event);
19
20        // zachycení změn v HTML modelu
21        click(fixture, 'submit-form');
22        fixture.detectChanges();
23      }
```

První test vyhodnocuje, zda jsou vložené data do reaktivního formuláře rovny datům, která jsou definována v pomocné specifikaci. I když formulář při inicializaci obsahuje data z komponenty, jsou vložena nová validní data. Tím se testuje ukládání dat do reaktivního formuláře.

```
1     it('Insert new value into field', () => {
2
3         // vložení vstupních dat
4         fillForm(name, email, password);
5
6         // vyhodnocení, zda jsou data správná
7         expect(findEl(fixture, 'name').nativeElement.value).toBe(name);
8         expect(findEl(fixture, 'email').nativeElement.value).toBe(email)
              ;
9         expect(findEl(fixture, 'password').nativeElement.value).toBe(
              password);
10    });
```

Další testy se zaměřují na testování validity reaktivních formulářů. První test kontroluje, zda jsou data, vložená do reaktivního formuláře, validní. Druhý test vyhodnocuje, zda je reaktivní formulář nevalidní, pokud je do něj vložen prázdný řetězec. Validaci, po vložení do reaktivního formuláře jména, které je delší než padesát znaků, kontroluje třetí test. Poslední test vyhodnocuje, zda hodnota vložená do reaktivního formuláře emailu obsahuje zavináč.

```
1     it('Is form valid', () => {
2         const form = component.form;
3
4         fillForm(name, email, password);
5
6         // testování, zda je formulář validní
7         const nameControl = form?.get('name');
8         expect(nameControl?.valid).toBeTruthy();
9         const emailControl = form?.get('email');
10        expect(emailControl?.valid).toBeTruthy();
11        const passwordControl = form?.get('password');
12        expect(passwordControl?.valid).toBeTruthy();
13    });
14
15    it('Name is longer than 50 char', () => {
16        const form = component.form;
17
18        fillForm(longName, email, password);
19
20        // testování, zda je vstup formuláře jména nevalidní
21        const nameControl = form?.get('name');
22        expect(nameControl?.invalid).toBeTruthy();
23      })
```

Kapitola 5

Vyhodnocení testů a reakce na jejich výsledky

Kapitola popisuje a vyhodnocuje dosavadní postup práce, porovnává použité testovací frameworky. Kapitola také hodnotí postup jednotkových i integračních testů a poskytuje přehled o výsledcích.

5.1 Zhodnocení použitého testovacího frameworku

Pro testování frameworku Angular byl zvolen testovací framework Jasmine. Tento framework byl vybrán z důvodu přímé implementace ve frameworku Angular. Při přípravě byla provedena rešerše na možné využití i jiných testovacích frameworků, například Protractor, ale po porovnání, testovací framework Jasmine dominoval. Framework Jasmine dominuje oproti dalším frameworkům, díky možnosti vytváření testovacích instancí před testy, možností špehování entit při jednotkových testech a v poslední řadě díky spolupráci s vyhodnocovacím frameworkem Karma.

Framework Jasmine je vhodný pro psaní jednotkových a integračních testů, zejména proto, že je nápomocen falšováním objektů. Při testování od začátku do konce (E2E) dochází ke složité implementaci testů. Pro tento typ automatických testů je vhodné použít již zmiňovaný testovací framework Protractor, který je přívětivější pro implementaci.

5.2 Hodnocení průběhu testování

Protože se práce zaměřuje na testování ve frameworku Angular, jsou testovány určité funkcionality tohoto frameworku.

Testování Angular komponenty bylo úspěšné. Jednotkový test psaný pro komponentu byl vyhodnocen kladně, tedy všechny potřebné funkcionality dané komponenty byly otestovány. Otestovány byly také asynchronní operace, které byly správně vyhodnoceny.

Při testování Angular komponenty s dětskou komponentou je do konfigurace testovacího modulu přidáno `schemas: [NO_ERRRORS_SCHEMA]`. Toto schéma ignoruje vytváření dětských komponent při volání v HTML šabloně, aby mohl být proveden jednotkový test. Pokud by schéma definované nebylo, nastalo by chybné vyhodnocení testu, protože do testovacího modulu nejsou přidány všechny dětské komponenty, na kterých je testovaná komponenta závislá. Jedná se tedy o využívání části kódu, která není vytvořena. Při vlo-

žení dětských komponent do konfigurace, se z jednotkového testu stává integrační test a nemusí být správně otestována funkcionalita komponenty.

Provedený jednotkový test byl správně vyhodnocen. Přestože přidané schéma ignoruje nepoužitý kód, jsou otestovány vstupy i výstupy z dětské komponenty a současně je otestována i možná reakce na tato data.

Testování Angular komponenty se službou, obsahuje v jedné testovací specifikaci jednotkový i integrační test, a to z důvodu testování samotné komponenty, která nemůže mít při jednotkovém testu definovanou závislost na službě. Jelikož tento způsob testování využívá data služby, je potřeba testovat správnost těchto dat. Z tohoto důvodu se vytváří falešná služba s falešnými daty, použitými pro testování. Při integračním testu není potřebné dodržet izolaci testu, a proto je možné v konfiguraci testovacího modulu vkládat závislost na službě.

Testy byly vyhodnoceny kladně a byla správně otestována funkcionalita jak samotné komponenty, tak komunikace mezi službou a komponentou.

Při testování Angular služby se také řeší problémy se závislostmi jiných služeb. Dochází zde ke stejnému nedostatku, jako u testování Angular komponenty se službou, kdy je pro zaručení izolace jednotkového testu potřeba falšovat závislé služby. Pro služby HTTP požadavků je vytvořen ve frameworku Angular samostatný testovací modul, kterým lze tuto funkcionalitu nahradit.

Ve webové aplikaci jsou psány dvě testovací sady pro Angular služby. Jedna testovací sada testuje službu bez závislostí, je tedy možné vynechat celou definici testovacího modulu a použít čistou instanci služby. Druhá testovací sada testuje službu s HTTP požadavkem. U této testovací sady je vložena závislost do testovacího modulu, pro již zmíněný testovací modul HTTP požadavků.

Oba testy služeb byly hodnoceny kladně a byla otestována celá funkcionalita.

Testování reaktivních formulářů bylo hodnoceno kladně, nicméně jsou zde otestovány pouze hlavní testovací parametry reaktivního formuláře. Vedlejší parametry, například, zdali je možné do formuláře zapisovat, nebylo testováno.

5.3 Vyhodnocení testů

Testy jsou vyhodnoceny pomocí frameworku Karma. Tento framework vypíše vyhodnocení testů do terminálu a následně spustí webový prohlížeč chrome a vizuálně zobrazí vyhodnocení testů.

Pokud nastane chyba, jsou chybové hlášky ve webovém prohlížeči zobrazeny červenou barvou s nejvyšší prioritou. Nemůže se tedy stát, že by došlo k přehlédnutí chybného testu. Pokud jsou všechny testy vyhodnoceny správně, jsou zobrazeny zelenou barvou a uspořádány do bloku. Obsahuje-li test více vyhodnocení, tak po rozkliknutí tohoto testu, je možné si prohlédnout, které z vyhodnocení bylo chybné.

Jasmine 4.6.0

33 specs, 0 failures, randomized with seed 07128

IsaComponent
- should create

ScheduleComponent
- should create

HomeTaskComponent
- should create

ActualScheduleComponent
- should create

NavBarComponent
- should create

AppComponent
- should create the app

BasicIndexComponent
- should create

SubjectTestComponent
- Input binding on isa component
- Render isa component
- Output binding on isa component for event emiter is false
- Output binding on isa component for event emiter is true
- Should create

Obrázek 5.1: Ukázka vyhodnocení testů ve frameworku Karma

Kapitola 6

Závěr

Cílem práce bylo provést podrobnou rešerši problematiky možností testování frontendu v programovacím jazyce JavaScript a implementovat automatizované testování webové aplikace psané ve frameworku Angular.

Před samotným návrhem automatizovaných testů bylo nezbytné provést výběr dostupných typů testů vhodných pro webové aplikace. Dále bylo potřebné důkladně představit framework Angular, zejména z pohledu jeho architektury a funkčnosti komponent. Během této fáze byly identifikovány klíčové funkcionality, které je vhodné testovat.

Před implementací bylo třeba vybrat konkrétní testovacího frameworku, který bude nejvhodnější pro potřeby daného projektu.

Hlavní zaměření automatizovaných testů bylo na jednotkové a integrační testy, které testují specifickou funkcionalitu frameworku Angular. Zároveň byl kladen důraz na jednoduchou implementaci a vhodné vizuální zobrazení testů. Před začátkem testování byla vytvořena pomocná webové aplikace, na které byly testy implementovány.

Přínosem bakalářské práce jsou možnosti testování specifického prostředí frameworku Angular. Jedním z prvních uvedených testů je správné jednotkové testování komponenty. Tento test kontroluje chování dat, která jsou měněna interakcí uživatele. Testování komponenty také zahrnuje asynchronní testování, které vyhodnocuje stavy určitých elementů. Dále je proveden jednotkový test nad rodičovskou komponentou s vazbou komponentou dětskou, při kterém je nutno dětskou komponentu falšovat. Implementovaly se také testy komponenty se službou, které mají dva testovací scénáře, jeden pro integrační test a druhý pro jednotkový test. Poté byla testována služba pomocí jednotkových testů s využitím testovacího modulu i bez jeho použití. Nakonec byly testovány reaktivní formuláře a jejich validita. Všechny testy byly provedeny ve frameworku Jasmine a vyobrazeny ve frameworku Karma.

Během tvorby testů vzniklo mnoho námětů, jak lze testování dále zdokonalit a rozšířit. Významné zlepšení by mohlo přinést rozšíření testování na služby, zejména pokud mají tyto služby více závislostí. Dále by bylo užitečné zvážit testování směrnic nebo toků dat, což by umožnilo detailnější ověření chování aplikace. Další možností je využití jiného typu testování, například testování od začátku do konce pomocí frameworku Protractor, což by umožnilo simulaci reálných interakcí uživatele s aplikací.

Práce může být dobrým pomocníkem všem, kteří řeší způsoby testování webových aplikací ve frameworku Angular. Zároveň jsou tyto způsoby implementovány a demonstračně použity ve webové aplikaci.

Literatura

[1] Angular. *Basics of testing components* [online]. 2022 [cit. 2024-04-020]. Dostupné z: https://angular.io/guide/testing-components-basics.

[2] Angular. *Testing services* [online]. 2022 [cit. 2024-04-020]. Dostupné z: https://angular.io/guide/testing-services.

[3] Angular. *Component testing scenarios* [online]. 2023 [cit. 2024-04-020]. Dostupné z: https://angular.io/guide/testing-components-scenarios.

[4] Angular. *Reactive forms* [online]. 2023 [cit. 2024-04-020]. Dostupné z: https://angular.io/guide/reactive-forms.

[5] Angular. *Introduction to the Angular docs* [online]. 2024 [cit. 2024-04-020]. Dostupné z: https://angular.io/docs.

[6] Borovcová, A. *Testování webových aplikací* [online]. 2008 [cit. 2024-03-13]. Dostupné z: https://dspace.cuni.cz/bitstream/handle/20.500.11956/17225/DPTX_2006_1_11320_0_217475_0_46536.pdf.

[7] BrowserStack. *Functional Testing : A Detailed Guide* [online]. 2024 [cit. 2024-01-04]. Dostupné z: https://www.browserstack.com/guide/functional-testing.

[8] BrowserStack. *Regression Testing: A Detailed Guide* [online]. 2024 [cit. 2024-01-04]. Dostupné z: https://www.browserstack.com/guide/regression-testing.

[9] Hambling, B. *Software Testing An ISTQB-ISEB Foundation Guid.* 2. vyd. BCS The Chartered Institute for IT, 2010. ISBN 978-1-906124-76-2.

[10] Jasmine. *Jasmine* [online]. 2024 [cit. 2024-04-02]. Dostupné z: https://jasmine.github.io.

[11] Jorgensen, P. C. *Software Testing A Craftsman's Approach.* 4. vyd. CRC Press, 2014. ISBN 978-1-4665-6069-7.

[12] Karma. *Karma* [online]. 2013 [cit. 2024-04-020]. Dostupné z: https://karma-runner.github.io/latest/index.html.

[13] Katalon. *What is End-to-End Testing? Definition, Tools, Best Practices* [online]. 2024 [cit. 2024-01-04]. Dostupné z: https://katalon.com/resources-center/blog/end-to-end-e2e-testing.

[14] Moc, D. *Vývoj webových aplikací* [online]. 2019 [cit. 2024-03-13]. Dostupné z: https://is.ambis.cz/th/dfcat/Diplomova_prace_Dalibor_MOC.pdf.

[15] Services, A. W. *What is unit testing* [online]. 2024 [cit. 2024-03-03]. Dostupné z: https://aws.amazon.com/what-is/unit-testing/.

[16] Sokol, M. *Automatické testování webových aplikací* [online]. 2013 [cit. 2024-03-13]. Dostupné z: https://is.muni.cz/th/awomw/diplomova_praca.pdf.

[17] Veselovský, E. *Přehled nástrojů proautomatické testování aplikací* [online]. 2014 [cit. 2024-03-13]. Dostupné z: https://otik.uk.zcu.cz/bitstream/11025/13542/1/bp-Eduard%20Veselovsky.pdf.

Příloha A

Obsah přiloženého paměťového media a návod ke spuštění testované aplikace

- `Proj/is-vut/` – Adresářová struktura obsahující programový kód
- `doc/` – Text technické zprávy

Pro spuštění webové aplikace je nutné mít nainstalovaný `node.js` a framework Angular. Framework Angular je možné nainstalovat pomocí příkazu `npm install -g @angular/cli`. Při prvním otevření adresáře `Proj/is-vut/` je nutné v terminálu použít příkaz `npm install`, který stáhne potřebné moduly pro správnou funkci webové aplikace. Poté stačí použít příkaz `ng serve -port 4200`, který spustí webovou aplikaci na portu 4200. Pro spuštění testovacího frameworku Karma je nutné použít příkaz `ng test`, který spustí definovaný webový prohlížeč s přehledem testů.