Univerzita Hradec Králové

Fakulta informatiky a managementu

Katedra informačních technologií

Automatizované testování webové aplikace Onboardee.io

Bakalářská práce

Autor: Hai Yen Pham

Studijní obor: Aplikovaná informatika

Vedoucí práce: Ing. Martina Husáková, Ph.D.

Hradec Králové Duben 2023

Prohlášení

Prohlašuji, že jsem bakalářskou práci zpracovala samostatně a s použitím uvedené literatury.

V Hradci Králové dne 22.4.2023 Hai Yen Pham

Poděkování

Děkuji vedoucí bakalářské práce Ing. Martině Husákové, Ph.D. za odborné vedení práce, cenné připomínky a čas, který mi věnovala. Dále bych chtěla poděkovat Ing. Štěpánovi Bartyzalovi, Markovi Feuermannovi a celému týmu Recruitis.io za tuto příležitost, odborné vedení a podporu při vypracování práce. V neposlední řadě patří veliké díky také mé rodině a přátelům za obrovskou podporu v průběhu celého studia.

Anotace

Tato bakalářská práce se zaměřuje na automatizované testování webových aplikací, konkrétně na testovací nástroj Cypress a jeho využití při testování webových aplikací. Cílem bakalářské práce je návrh automatizovaných testů pro nově vyvíjenou webovou aplikaci Onboardee.io ve společnosti Recruitis.io s.r.o.

Teoretická část práce přináší obecný vhled do problematiky testování, následně se zaměřuje hlouběji na automatizované testování a představuje vybrané nástroje, které lze k tomu využít.

V praktické části práce jsou definovány požadavky a kritéria pro testování konkrétní aplikace. Následně je provedena analýza a výběr testovacího nástroje vhodného pro testování. Vybrané metody jsou poté aplikovány a popsány při testování samotné webové aplikace Onboardee.io. Výsledky jsou dále analyzovány pro zlepšení kvality produktu a také samotných testů.

Klíčová slova

testování, automatizované testování, Cypress, webové aplikace

Annotation

Title: Automated testing of the web application Onboardee.io

The topic of this bachelor thesis is automated testing of web applications with the aim of creating automated tests for the web application Onboardee.io using a testing framework called Cypress.

The theoretical part of the thesis describes the fundamentals of software testing, gives further insight into the automated testing of web applications, and presents various frameworks used for testing web applications.

The practical part introduces the web application Onboardee.io and its requirements for testing. Created scenarios are then implemented using the chosen testing framework. The results of these tests are further analyzed to improve the web application and the testing itself.

Keywords

testing, automated testing, Cypress, web application

Obsah

1 Úvod ... 1

2 Testování softwaru ... 2

2.1 Základní pojmy ... 2

2.2 Metody testování ... 3

2.2.1 Černá a bílá skříňka ... 3

2.2.2 Statické a dynamické testování ... 4

2.2.3 Testy splněním a selháním ... 5

2.2.4 Automatické a manuální testování ... 5

2.3 Fáze testování ... 6

2.3.1 Jednotkové testování ... 6

2.3.2 Integrační testování ... 6

2.3.3 Systémové testování ... 6

2.3.4 Akceptační testování ... 8

2.4 Problémy spojené s testováním ... 8

3 Automatizace ... 10

3.1 Výhody a nevýhody automatizace ... 10

3.2 Zásady automatizace ... 10

3.3 Automatizace testů ... 10

3.3.1 Automatizace jednotkových testů ... 11

3.3.2 Automatizace integračních testů ... 11

3.3.3 Automatizace testování výkonu ... 11

4 Nástroje pro automatizaci testování ... 12

4.1 Selenium ... 12

4.1.1 Selenium RC ... 12

4.1.2 Selenium WebDriver ... 12

4.1.3 Selenium IDE ... 13
4.1.4 Selenium Grid ... 13
4.2 Cypress ... 13
4.3 Playwright ... 13
4.4 Screenster ... 14
5 Cypress ... 15
5.1 Výhody a nevýhody ... 15
5.2 Cypress vs. Selenium ... 16
5.3 Práce se Cypressem ... 17
5.3.1 Získání Cypressu ... 17
5.3.2 Vytváření Cypress testů ... 18
5.3.3 Spuštění testů ... 23
5.3.4 Debuggování testů ... 23
6 Webová aplikace Onboardee.io ... 25
6.1 Technické specifikace ... 26
6.2 Požadavky na testování ... 26
7 Návrh a implementace automatizovaných testů ... 28
7.1 Testovací scénáře ... 28
7.1.1 Analýza aplikace ... 28
7.1.2 Návrh testovacích scénářů ... 29
7.2 Implementace testů ... 32
7.2.1 Struktura projektu ... 32
7.2.2 Implementace testovacího scénáře TS_A_04 ... 33
7.2.3 Zajímavé části kódu ... 37
7.3 Spuštění vytvořených testů ... 39
7.4 Problémy v průběhu psaní testů ... 40

8 Shrnutí výsledků ... 42

9 Závěr ... 44

10 Seznam použité literatury ... 45

11 Seznam použitých obrázků ... 47

11.1 Seznam obrázků ... 47

11.2 Seznam zdrojových kódů ... 48

11.3 Seznam tabulek ... 50

12 Přílohy ... 51

1 Úvod

V dnešní době je internet nedílnou součástí života většiny populace. Každou minutu vznikají nové a nové stránky a aplikace, které se přetahují o pozornost uživatelů. Z tohoto důvodu je návrh a implementace funkční a uživatelsky přívětivé webové aplikace či stránky jedním z hlavních požadavků. Jak dosáhnout tohoto cíle? Předtím než je aplikace spuštěna a zpřístupněna celému světu, je třeba zajistit, že funguje tak, jak má. K tomuto účelu je určené testování, které kontroluje funkcionalitu, bezpečnost, kompatibilitu a v neposlední řadě výkonnost webové aplikace či stránky. A jelikož každá minuta něco stojí, často se opakující a rutinní aktivity je vhodné ponechat na strojích, tedy testy automatizovat. A právě tímto se tato bakalářská práce zabývá.

V úvodní části bakalářské práce je popsán obecně pojem testování, jeho zákonitosti a dále obsahuje různá rozdělení testování. Další část této práce se zaměřuje hlouběji problematikou automatizace testovacího procesu, jejími výhodami a nevýhodami či pojednává o druzích testů, které lze automatizovat. Poslední část teoretické části představuje vybrané nástroje, které se pro automatizaci testování webových aplikací využívají. A představuje podrobněji jeden z novějších nástrojů, Cypress.

Praktická část práce se zaměřuje konkrétně na webovou aplikaci Onboardee.io společnosti Recruitis.io. Požadavky na testování aplikace jsou analyzovány a za pomoci nástroje Cypress je implementována sada automatizovaných testů. Výsledky jsou dále analyzovány pro zlepšení kvality aplikace Onboardee.io a také samotných testů.

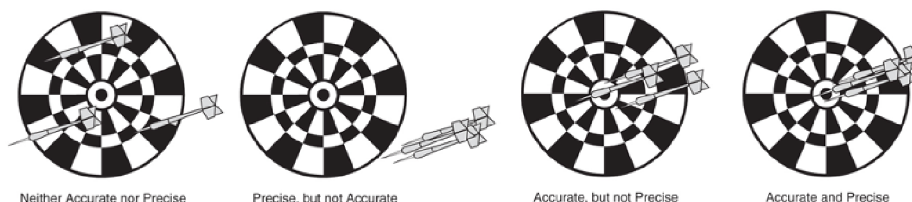
2 Testování softwaru

Testování je nedílnou součástí životního cyklu softwaru, jehož cílem je nalezení chyb a zajištění jejich nápravy, a to co nejdříve. Pozdější odhalení chyby totiž znamená vyšší náklady na její opravení. Účelem testování ale není jen nalezení chyb, ale také zajištění kvality produktu, tedy že je použitelný, spolehlivý a bezpečný. To vše je prováděno se snahou minimalizovat celkové náklady na vývoj.

2.1 Základní pojmy

Při testování se můžeme setkat s následujícími pojmy, mezi kterými jsou jisté rozdíly, které je důležité mít na paměti.

Jedním takovým důležitým rozlišením je rozdíl mezi přesností a precizností (z anglického „precision and accuracy"), což znázorňuje Obrázek 1, na němž lze graficky pozorovat rozdíl mezi těmito pojmy.



Obrázek 1: Znázornění rozdílu mezi přesností a správností na šipkách [1] (upraveno autorkou práce)

Cílem této hry je zasáhnout střed terče. Hody na první terč nejsou přesné ani precizní, protože se šipky nevyskytují ve středu terče, ani poblíž sebe. Na druhém terči je znázorněna situace, kdy byl hráč přesný, což je znázorněno shlukem šipek, ale bohužel mu chyběla preciznost. Třetí terč znázorňuje preciznost, šipky jsou tedy blízko středu terče, hráč ale nebyl přesný se svými hody. Poslední obrázek znázorňuje jak přesnost, tak i preciznost [1].

Cílem testování je mít přesnou i precizní aplikaci, není to ale zcela možné, protože takové testovaní vyžaduje spoustu prostředků, a proto se musí na začátku testování rozhodnout, zda se bude klást větší důraz na správnost či preciznost aplikace.

Dalšími důležitými pojmy jsou verifikace a validace, které jsou často zaměňovány mezi sebou. Je mezi nimi ale jistý rozdíl, který je potřeba rozlišovat. Verifikace je

proces, při kterém se potvrzuje, že software splňuje požadované specifikace. Zatímco validace potvrzuje, že finální produkt vyhovuje požadavkům uživatele. Při testování by se měl software verifikovat i validovat.

Také je potřeba si vymezit pojmy kvalita a spolehlivost. Často dochází k mylnému úsudku, že se kvalitou a spolehlivostí rozumí to samé. Pokud je software spolehlivý, neznamená to nutně, že je také kvalitní, protože je spolehlivost pouze jedním z aspektů kvality. Proto, aby byl software jak kvalitní, tak spolehlivý, je potřeba v průběhu vývoje verifikovat i validovat zároveň.

V případě ohodnocování závažnosti chyb se můžeme setkat s pojmy priorita a závažnost. Opět se tyto pojmy často zaměňují. V případě testování softwaru je mezi nimi mírný rozdíl. Priorita určuje důležitost dané věci na základě různých kritérií, typicky na důležitosti pro zákazníka. Zatímco závažnost definuje míru dopadu chyby, která je určena počtem postihnutých uživatelů a frekvencí výskytu problému [2].

2.2 Metody testování

Existuje několik způsobů jak testovat. Obecně můžeme testovaní rozdělit do několika kategorií podle různých ukazatelů jako jsou znalosti implementace webové aplikace, způsob provádění testů, rozměry kvality, které testy ověřují či fáze vývoje, ve kterém se testování vykonává.

Pro konkrétní produkt se mohou hodit různé metody testování, proto je důležité zvolit ten správný. V opačném případě může dojít k neodhalení velkého množství závažných chyb, či neefektivnímu testovacímu procesu.

2.2.1 Černá a bílá skříňka

Černá a bílá skříňka je jednou z možností rozdělení testů, a to na základě znalosti testovaného kódu.

2.2.1.1 Černá skříňka

Testování černé skříňky (z anglického „black box testing“), neboli také funkcionální testování, je jednou ze základních a nejčastěji využívaných technik, které se provádí

na finálním produktu. Vztahuje se na testování, při kterém je na testovaný systém pohlíženo jako na černou schránku. Zde tedy nepotřebujeme k testování znalosti specifického programovacího jazyka ani samotné implementace testované aplikace, což je hlavní výhodou tohoto způsobu. Testování je prováděno čistě z pohledu uživatele a pouze tester zná sadu vstupních dat a předpokládaných výstupů [3].

2.2.1.2 Bílá skříňka

Testování bílé skříňky (z anglického „white box testing"), či také strukturální testování, je způsob, při kterém musí tester porozumět vnitřním datovým a programovým strukturám a také jak je systém implementován. To dělá tento způsob náročnější a také mnohem nákladnější. Jak zmiňují autoři v [4], tato technika se využívá především na detekování logických chyb v kódu, pro debuggování kódu, nalezení typografických chyb a odhalení chybných předpokladů programu. Napomáhá tedy k vyšší kvalitě kódu výsledného produktu.

2.2.1.3 Šedá skříňka

Občas je možné se setkat s testováním typu šedá skříňka, který je jakýmsi kompromisem mezi výše zmíněnými typy testování a u kterého má tester určité znalosti o implementaci, ale nejsou na takové úrovni, jako je tomu při testování typu bílá skříňka. Příkladem je testování webové aplikace, kde tester nemá k dispozici celý zdrojový kód, ale pouze HTML kód výsledné stránky.

2.2.2 Statické a dynamické testování

Toto rozdělení vychází z toho, zda je k provedení testu potřeba software pustit.

2.2.2.1 Statické testování

Při statickém testování není vyžadován běh softwaru. Využívá se zejména v raných fázích životního cyklu softwaru, Je totiž možné s ním začít již před vytvořením první spustitelné verze daného programu, a to třeba na kontrolu specifikace požadavků a analýzu zdrojového kódu [5].

2.2.2.2 Dynamické testování

Dynamické testy jsou pravým opakem statického testování, resp. je při nich třeba spuštění aplikace. Potřebují alespoň spustitelný prototyp softwaru, používají se proto v pozdějších fázích vývoje a jsou zaměřeny na provoz softwaru [5].

2.2.3 Testy splněním a selháním

Dalším typem testů jsou testy splněním a selháním, které jsou velmi často využívané, zejména v kombinaci s testováním černé skříňky.

2.2.3.1 Testy splněním

Na začátku vývoje produktu je zřejmé, že funkcionality nejsou zcela kompletní, přesto je potřeba mít představu o aktuálním stavu. Při testech splněním jsou vstupními hodnotami množina dat, kterou musí aplikace vždy akceptovat. Úspěšný průchod testů je považován za milník pro přípravu k procesu integrace.

2.2.3.2 Testy selháním

Naopak ke konci vývoje se funkcionality považují za stabilizované a průchod testů splněním se považuje za samozřejmé. Tehdy se hledají případy, kdy testovaní selže. Do aplikace se tedy zadávají nestandardní data jako např. dělení nulou, extrémní hodnoty apod. [5] a během testů ověřujeme, že výstupní data neobsahují nežádoucí hodnoty.

2.2.4 Automatické a manuální testování

Testy lze dále rozdělit podle toho, zda jsou prováděny člověkem nebo softwarem, tj. na manuální a automatické.

2.2.4.1 Manuální testování

Manuální testování je proces, při kterém tester provádí testování individuálně krok po kroku bez využití nástrojů či skriptů. Používá se, pokud test vyžaduje lidské vyhodnocení a úsudek či přístupy, které není třeba zaznamenat a pravidelně opakovat.

2.2.4.2 Automatické testování

Při automatickém testování jsou testy provedeny pomocí nástrojů, skriptů a softwaru. Jelikož jsou testy automatizované prováděny mnohem rychleji než manuální testy, proto se používají pro spouštění velkého množství testů, testů s velkým množstvím generovaných dat nebo také pro zátěžové testování.

2.3 Fáze testování

Testování lze rozdělit z časového hlediska na následující fáze podle toho, kdy se od napsání kódu provádí [5]:

- jednotkové testování (Unit testing),
- integrační testování (Integration testing),
- systémové testování (System testing),
- akceptační testování (Acceptance testing).

2.3.1 Jednotkové testování

Jednotkové testy (z anglického „unit tests") jsou první fází testování, která se zaměřuje na nejmenší testovatelné části tzv. jednotky aplikace [6]. Je prováděna samotnými vývojáři, kdy po dokončení části systému, ale i v průběhu vývoje, vývojář svou práci otestuje v izolaci od zbytku aplikace. Ověřuje si tím, že nová či změněná část kódu funguje a že její funkce odpovídá očekávání.

2.3.2 Integrační testování

Cílem integračních testů je ověřit, zda nově přidané funkcionality spolu nekolidují a všechny jednotlivé podsystémy pracují správně i po zapojení s ostatními částmi systému. Někdy bývají označovány jako „testy vnitřní integrace" a jsou připravovány především testovacím týmem [5].

2.3.3 Systémové testování

Systémové testy přicházejí na řadu až v pozdějších fázích vývoje. Cílem testování je zjištění, zda celý systém pracuje podle požadavků. Systém bývá většinou procházen podle testovacích scénářů, které simulují běžnou práci uživatele. Systém je obvykle procházen ve více kolech, přičemž jsou nalezené chyby opraveny a znovu

otestovány v dalším kole. Je možné rozlišovat následující druhy systémových testů podle [7]:

- **Testování funkčnosti** (functional testing) ukazuje funkčnost systému, kde každá požadovaná vlastnost systému musí být dosažitelná. Testy kladou důraz na korektnost výstupu funkcí po zpracování zadaných vstupů.
- **Testování výkonnosti** (performance testing) slouží k ověření, zda vyvíjená aplikace naplní předpokládané požadavky uživatelů, jako je přijatelnost reakční doby, cíle propustnosti, počet souběžně zpracovaných uživatelů a požadavky na budoucí růst výkonnosti.
- **Zátěžové testy** (load testing) testují, zda je systém schopný ustát zatížení při zabrání maxima prostředků. Cílem je zjistit, za jakých okolností dojde ke zhroucení systému. Takto se dají odhalit chyby v real-time systémech, kde se mohou během ostrého nasazení stát nepředvídatelné události překračující specifikace softwaru z důsledku zátěže. Zátěžové testy tak mohou odhalit chyby, které se těžko odhalují za normálních testovacích podmínek.
- **Testy hardwarové konfigurace** (configuration testing) ověřují, jestli systém běží stejně na různých hardwarových sestavách, nezávisle na změně konfigurace.
- **Testování bezpečnosti** (security testing) kontroluje, zda systém chrání svá data a funguje, jak bylo zamýšleno. Na tyto testy je kladen stále větší důraz s rozmachem celosvětové sítě, je ale jeden z nejnáročnějších, protože u každého softwaru mohou být různé postupy ke zneužití.
- **Testování schopnosti zotavení** (recovery testing) je proces, při kterém se systém vystaví ztrátě prostředků a poté ověřuje, zda se systém řádně zotaví. Toto testování se nasazuje zpravidla při testech bankovních systémů, kdy např. hrozí ztráta síťového spojení během transakce, nebo jiných kritických systémů.
- **Testování spolehlivosti** (reliability testing) se provádí za účelem změřit spolehlivost softwaru. Je definován jako pravděpodobnost, že systém bude fungovat bezchybně po daný časový interval.

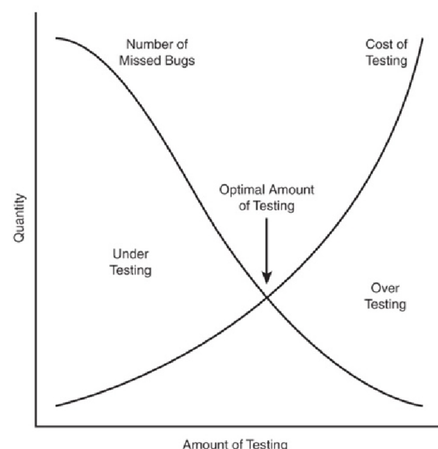
- **Testování použitelnosti** (usability testing) je součástí kontroly kvality softwaru. Zaměřuje se na přehlednost, zapamatovatelnost, rychlost použití – jaké úsilí je třeba vynaložit k pochopení ovládání systému. Z toho důvodu nelze tyto testy automatizovat, protože lidské rozhodování a reagováni na různé uživatelské prostředí se nedá plnohodnotně programově podchytit.

2.3.4 Akceptační testování

Pokud všechny předchozí etapy testů proběhly bez větších nedostatků, přichází na řadu uživatelské akceptační testy, které provádí samotný koncový uživatel ve svém vlastním testovacím prostředí. Slouží k ověření, zda software plní všechny funkční a nefunkční požadavky a je tedy připraven k nasazení do ostrého provozu.

2.4 Problémy spojené s testováním

Žádný software nelze kompletně otestovat. I ty na pohled nejjednodušší aplikace obsahují velké množství kombinací stavů, které mohou nastat, a proto je potřeba rozhodnout, jaké hodnoty se budou vůbec testovat a které se z testování vynechají. Při velkém množství testů vzrůstá také cena testování, a to zejména na udržování samotných testů, proto je potřeba najít hranici, kdy je testování stále dostatečně efektivní a vynaložené prostředky jsou k tomu přiměřené, jako je to znázorněno na Obrázku 2.



Obrázek 2: Každý software má vlastní optimální testování [1]

Testování tedy není důkazem, že software neobsahuje žádné chyby. Může pouze poukázat na chyby, které se v systému nachází [1].

Dalším problémem je tzv. pesticidový paradox (z anglického „pesticide paradox"), který popisuje fenomén, kdy čím víc software testujeme, tím víc je proti testům imunní [1]. Vztahuje se to zejména k automatizovaným testům, které při spuštění opakují jeden a ten samý scénář. Proto je důležité, aby testeři neustále psali nové testy, které testují jiné části softwaru.

Občas dochází k situacím, kdy se nalezené chyby úmyslně neopravují. To může nastat třeba z časových důvodů, kdy není dostatek prostředků k nápravě, či je oprava problému příliš riskantní a mohla by způsobit velké množství dalších. Dochází ale také k situacím, kdy oprava chyby prostě nestojí za to, pokud se jedná například o chybu, která se vyskytuje pouze vzácně.

V neposlední řadě je problémem fakt, že požadavky na produkt se s časem mění. Software se v dnešní době neustále obměňuje podle poptávky trhu a různých trendů, a tím mohou vznikat situace, kdy již otestované prvky projdou změnou, či dojde k jejich úplnému odstranění. Také mohou přibývat nové prvky, které je potřeba otestovat nad rámec původního plánu.

3 Automatizace

Automatizace je proces, při kterém jsou strojům přiřazovány aktivity dříve vykonávané lidmi. Nahrazení manuálních metod automatizovanými procesy umožňuje dosáhnout zvýšení kvality výrobku a také snížení nákladů.

3.1 Výhody a nevýhody automatizace

Jako vše v životě má automatizované testování jak pozitivní, tak i negativní stránky. Mezi výhody automatizace lze zařadit častější provádění testů za kratší čas, než by tomu bylo, kdyby byly testy prováděny manuálně. Napomáhá to zvýšit pokrytí testované oblasti. Dále testy mohou běžet neúnavně po dlouhou dobu, např. během noci, bez dozoru. Také přináší konzistenci opakování, které nelze zcela dosáhnout, pokud je testování prováděno člověkem, jelikož může být ovlivněn řadou faktorů, které poté ovlivňují samotné testování.

Přesto se bez lidského faktoru automatizované testy neobejdou, jelikož musí skripta někdo napsat. V tom může spočívat nevýhoda automatizace. Pokud má tvůrce slabou testovací praxi, může docházet k tvorbě neefektivních testů. Mnohdy se můžeme také setkat s příliš nereálnými očekáváními od vývojářů, či jiných členů týmu, které může vést ke tvoření příliš složitých testů. Ty se později těžce udržují, což má za následek zvýšení nákladů. Naprosto se tím ztrácí hlavní myšlenka tohoto typu testování, tedy snížení nákladů a zjednodušení práce.

3.2 Zásady automatizace

Při automatizaci testů je potřeba dodržet jisté zásady, které lze rozdělit na následující: testy by měly zvýšit kvalitu produktu, snížit riziko dalších chyb, napomáhat pochopit kód, musí být jednoduché na psaní a na běh, a neměly by být obtížné na údržbu [8].

3.3 Automatizace testů

Některé druhy testů, jako jsou např. testování použitelnosti, není možné automatizovat, u některých se jedná pouze o jednorázovou činnost, kterou není třeba automatizovat. Existují také testy, které by byly příliš složité tedy neefektivní,

pokud by se automatizovaly. Níže jsou zmíněné některé druhy testů [9], které se mohou automatizovat a také často automatizují.

3.3.1 Automatizace jednotkových testů

Jak již bylo dříve zmíněno, jednotkové testy jsou prováděny samotnými vývojáři a jsou zaměřené na nejmenší testovatelné části. Tyto testy je vhodné automatizovat, aby mohly být testy spouštěny neustále během vývoje, nejčastěji během procesu kompilace, díky čemuž vývojář zjistí vadu ihned a může ji co nejdříve opravit.

3.3.2 Automatizace integračních testů

Další testy, které se často automatizují jsou testy integrační. Ty mají za úkol zajistit, že interakce mezi jednotlivými jednotkami nevykazují žádné chyby. Automatizace těchto testů pomáhá s projevy regresních chyb, které změnami vznikají.

3.3.3 Automatizace testování výkonu

Výkonnostní testování je prakticky nemožné nasimulovat manuálně pomocí týmu testerů. Testy výkonu, kde je třeba simulovat stovky až tisíce uživatelů najednou, což je při manuálním testování organizačně i finančně náročné, jsou tedy automatizovány [9].

4 Nástroje pro automatizaci testování

Nástrojů pro automatizaci testování je v současné době mnoho. Nejdůležitějšími vlastnostmi nástrojů pro automatizaci testů jsou podle Rona Pattona [1]:

- rychlost – testovací nástroje provádí testy mnohonásobně rychleji než člověk,
- efektivita – při vykonávání testů automatizovanými nástroji, máme prostor plánovat další testovací případy, či manuálně testovat případy které nelze zautomatizovat,
- přesnost – testovací nástroj provádí práci vždy se stejnou přesností, čehož člověk nemůže dosáhnout,
- neúnavnost – automatické testy mohou neúnavně běžet bez přestání.

V této kapitole je uveden výběr nástrojů určených k testování webových aplikací se zaměřením na uživatelské rozhraní.

4.1 Selenium

Selenium je jednou z nejznámějších open-source sad nástrojů pro automatické testování webových aplikací. Jednou z největších výhod tohoto nástroje je podpora spouštění testů v mnoha webových prohlížečích a na mnoha platformách. Samotné testy lze psát v mnoha programovacích jazycích. Oficiálně jsou podporovány jazyky: Java, Python, C#, Ruby, JavaScript, Kotlin [6].

4.1.1 Selenium RC

Selenium Remote Control, neboli Selenium 1, je nástroj, jenž umožňuje vytváření automatizovaných testů ve velké škále programovacích jazyků. Stavebním kamenem tohoto nástroje je server, Selenium Server, který slouží jako proxy server, prostředník mezi prohlížečem a testovanou webovou aplikací.

4.1.2 Selenium WebDriver

Selenium WebDriver neboli Selenium 2, je následníkem Selenia RC a přináší jednodušší přístup vytváření testů. K jeho spouštění není třeba používat Selenium

Server, jelikož má ve svém API integrovanou podporu pro otevírání webového prohlížeče.

4.1.3 Selenium IDE

Selenium Integrated Development Environment je implementováno jako plugin do webového prohlížeče Chrome, Firefox či Edge, jehož výhodou je snadná a rychlá instalace. Další a také hlavní výhodou tohoto doplňku je možnost vytváření a spouštění testů přímo ve webovém prohlížeči.

4.1.4 Selenium Grid

Selenium Grid slouží k paralelnímu spouštění testů na více webových prohlížečích za využití virtuálních strojů. Pomocí tohoto nástroje je tedy možné testovat velké sestavy testů na více zařízeních ve stejnou dobu a tím navýšit jejich výkon.

4.2 Cypress

Cypress je open-source javascriptový testovací framework, nástroj pro testování webových aplikací v reálném prostředí prohlížeče, který zjednodušuje psaní testů. Umožňuje psaní tzv. end-to-end testů, které provádějí ověření kompletních procesů v aplikaci. Dále umožňuje psaní také integračních nebo jednotkových testů, ty ale nejsou primárním záměrem a mohou způsobovat pomalejší běh.

Tomuto testovacímu nástroji se bude podrobněji věnovat následující kapitola.

4.3 Playwright

Playwright je open source nástroj pro automatické testování od Microsoftu, který umožňuje psát end-to-end testy v TypeScriptu, Javascriptu, Pythonu, v .NET a v Javě. Testy pak mohou být spuštěny v klasických prohlížečích jako je Chromium, Firefox nebo Safari. Playwright disponuje také dalšími nástroji jako jsou Codegen, Playwright Inspector a Traceview, které ještě více ulehčují práci. Například Codegen dovoluje uživatelům nahrávat své akce, které je možné uložit v jakémkoliv jazyce [10].

4.4 Screenster

Existují také testovací nástroje, které nevyžadují vytváření skriptů. Jedním z nich je například Screenster, který nahrává jednotlivé kroky uživatele při používání testované aplikace, čímž vytváří komplexní testovací scénáře. Přičemž 95 % testů lze vytvořit pomocí nahrávání a zbylých 5 % dopsat v Javascriptu či Javě jako Seleniový test [11]. Vytvořené testy pak dovoluje spouštět v prohlížečích jako jsou Chrome, Firefox, Safari či Edge na cloudu.

5 Cypress

Cypress je open source testovací nástroj pro testování frontendu, který umožňuje psaní end-to-end testů, testů na testování komponent a dále integračních a jednotkových testů. Je však hlavně přizpůsobený pro testování prvních dvou zmíněných typů testování, testování zbylých dvou tedy může mít za následek pomalejší běh. Běh testů je pak možný realizovat v prohlížečích Chrome, Edge a nově i ve Firefox.

Nástroj je poměrně jednoduchý jak na psaní, tak na pochopení testů. Jeho nejatraktivnějším prvkem je však velice kvalitně zpracované UI testovací prostředí, díky kterému má uživatel přehled o průběhu testu, má také možnost se zpětně přepnout do jakéhokoliv okamžiku proběhlého testu a pomocí této funkcionality zjistit, kde došlo k chybě.

5.1 Výhody a nevýhody

Mezi přednosti tohoto testovacího nástroje je zcela jistě již zmíněné UI testovacího prostředí, který přináší uživateli přehled o průběhu testu. Po skončení celého testu má pak uživatel možnost procházet jednotlivé kroky a může se do jakéhokoliv okamžiku zpětně přepnout. Pokud jsou testy spouštěny v terminálu, jsou pořizovány snímky obrazovky částí, kde nastaly chyby a k tomu vzniká také video, zachycující celý test.

Cypress, jako vše na světě, není zcela dokonalý a obsahuje různé nedostatky. Mezi hlavní nevýhody je považovaný fakt, že lze psát testy pouze v Javascriptu, nenabízí tedy žádné alternativy. Dále obsahuje pouze limitovanou podporu iframů[1] a kompletně chybí podpora multi-tabu[2]. Také neumí spouštět více prohlížečů najednou. A jak již bylo zmíněno, momentálně mohou být testy spuštěny pouze v prohlížečích Chrome, Firefox a Edge, oproti konkurenci chybí například Safari.

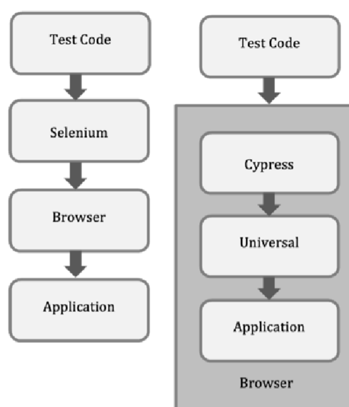
[1] Iframe (z anglického „inline frame“) je HTML a XHTML element, který umožňuje vložit do stránky plochu přesné velikosti a zobrazit v ní jinou stránku.

[2] Multi-tab (z anglického „multi-tabs“) znamená více záložek.

5.2 Cypress vs. Selenium

Cypress se svou rostoucí popularitou bývá často porovnáván právě se Seleniem, které je zcela jistě nejznámějším testovacím nástrojem v oblasti testování UI, oba testovací nástroje jsou ale ve své podstatě rozdílné [12].

Rozdíl je již v architektuře těchto nástrojů, což znázorňuje Obrázek 3. Cypress, který běží přímo v prohlížeči, je znatelně rychlejší než Selenium WebDriver, který interaguje s prohlížečem pomocí HTTP protokolu, což způsobuje jistá zpoždění [13].



Obrázek 3: Selenium versus Cypress z hlediska architektury provádění testů [13] (upraveno autorkou práce)

Pokud se zaměříme na podporované jazyky těchto nástrojů, zjistíme, že Selenium přináší mnohem větší výběr oproti Cypressu, který podporuje pouze Javascript. Selenium testy lze psát ve všech populárních jazycích jako Java, Python, Ruby, C#, PHP a ve spoustě dalších [14].

Co se týče podporovaných frameworků. Cypress, který podporuje pouze Mocha JS, v tomto ohledu opět zaostává. Selenium nabízí podporu několika frameworků specifických pro podporované jazyky zmíněné v předchozím odstavci. Podporuje například JUnit pro Javu nebo Cucumber pro Javascript a spoustu dalších [14].

Dalším rozdílem je ve složitosti při začínání s testováním. Zatímco Cypress není třeba složitě nastavovat, abychom spustili Selenium WebDriver, musíme nastavit a stáhnout další prvky jako například drivery pro specifické prohlížeče nezbytné pro běh testů.

Oba testovací nástroje mají své výhody a nevýhody, záleží tedy na požadavcích konkrétního projektu pro zvolení toho správného nástroje. Je však možné použít oba nástroje, Cypress a Selenium, najednou a tak limitace jednoho nástroje, jako například nepodporování multi-tabu v Cypressu, vyřešit nástrojem druhým.

5.3 Práce se Cypressem

Následující kapitola se zabývá samotnou prací se Cypressem. Nejprve je popsán způsob získání testovacího frameworku, poté je princip psaní testů znázorněn na jednoduchém příkladu a následně je popsán způsob spuštění a debuggování.

5.3.1 Získání Cypressu

Abychom mohli testy vytvářet a spouštět, je potřeba Cypress nejprve stáhnout. Jedna z cest jak nainstalovat Cypress je přes konzoli. Nejprve je potřeba vstoupit do složky s projektem, který by měl obsahovat složku `node_modules` nebo soubor `package.json`.

```
cd /your/project/path
```

Zdrojový kód 1: Přesun do složku projektu [12]

Instalace je pak možná za použití `npm`, které je doporučováno i samotným testovacím frameworkem, protože umožňuje verzování Cypressu jako všechny ostatní závislosti projektu a navíc zjednodušuje běh Cypressu v průběžné integraci neboli CI (z anglického „Continuous Integration“) a to za pomocí příkazu níže.

```
npm install cypress --save-dev
```

Zdrojový kód 2: Instalace Cypressu pomocí npm [12]

Není to jediný způsob. Framework lze nainstalovat za pomoci `yarn` a to následovně.

```
yarn add cypress --dev
```

Zdrojový kód 3: Instalace Cypressu pomocí yarn [12]

Cypress lze získat také z oficiálních stránek přímým stažením. To je ale doporučováno pouze pro vyzkoušení frameworku, protože tato verze Cypressu

nepodporuje například nahrávání testů na Cypress Cloud. Po stažení a rozbalení zip souboru stačí Cypress pouze spustit, není třeba žádné další instalace.

5.3.2 Vytváření Cypress testů

Po instalaci Cypressu přichází čas k samotnému vytváření testů. Při tvorbě testů je důležité mít jasně definované cíle a pečlivě navrhnout testovací scénáře, aby co nejvíce pokryly funkcionalitu aplikace. K usnadnění práce nabízí Cypress řadu užitečných funkcí, které umožňují například přístup k DOM elementům, či kontrolu stavu aplikace. Základním příkazům se bude věnovat následující podkapitola.

5.3.2.1 Základní příkazy

Následuje výpis základních, nejvíce používaných, Cypress příkazů používaných pro tvorbu testů.

`cy.visit()` | Jedná se o příkaz pro navštívení dané stránky. Jako základní parametr je url adresa. Pro testování je doporučeno si nastavit `baseUrl` v souboru `cypress.config.js` na kořenovou adresu projektu, díky čemuž pak není třeba udávat celou url adresu.

```
cy.visit(url)
cy.visit(url, options)
cy.visit(options)

cy.visit('http://google.com')
```

Zdrojový kód 4: Ukázka použití příkazu cy.visit() [12]

`cy.url()` | Návratová hodnota tohoto příkazu vrací svoji aktuální url adresu, která se pak může zkontrolovat pomocí příkazu `.should()`, navázaného přímo na tento příkaz.

```
cy.url()
cy.url(options)

cy.url().should('include', '/index.html')
```

Zdrojový kód 5: Ukázka použití příkazu cy.url() [12]

cy.get() | Tímto příkazem se odkazujeme na konkrétní DOM elementy na testované stránce. Jako parametr se udává CSS selektor, tedy buď selektor třídy (`.název_třídy`), id selektor (`#id`), selektor elementu (`element1 > element2`), pseudo selektory tříd (`:active`) či globální selektor (`*`).

```
cy.get(selector)
cy.get(alias)
cy.get(selector, options)
cy.get(alias, options)

cy.get('.dropdown-menu').click()
```

Zdrojový kód 6: Ukázka použití příkazu cy.get() [12]

.should() | Příkaz pro provádění kontroly se řetězí za předchozím příkazem, nejde jej použít jako samostatný příkaz. Pro kontrolu využívá asertovací knihovnu Chai a její rozšíření Sinon a jQuery.

```
.should(chainers)
.should(chainers, value)
.should(chainers, method, value)
.should(callbackFn)

cy.get(':checkbox').should('be.disabled')
```

Zdrojový kód 7: Ukázka použití příkazu .should() [12]

.click() | Tímto příkazem jsme schopni kliknout na určený DOM element, je tedy potřeba jej zřetězit za příkaz, který se odkazuje na DOM element.

```
.click()
.click(options)
.click(position)
.click(position, options)
.click(x, y)
.click(x, y, options)

cy.get('.action-btn').click()
```

Zdrojový kód 8: Ukázka použití příkazu .click() [12]

.type() | Tento příkaz zase dovoluje psát do DOM elementu, který je k zapisování určen. Jako u kliknutí je potřeba řetězit tento příkaz za takovým, který odkazuje na DOM element, tedy nejčastěji za příkaz `cy.get()`.

```
.type(text)
.type(text, options)

cy.get('input').type('Hello, World')
```

Zdrojový kód 9: Ukázka použití příkazu .type() [12]

5.3.2.1.1 Struktura testů

Cypress využívá testovací strukturu testovacího frameworku Mocha, přesněji BDD[3] rozhraní, které poskytuje `describe()` a `it()` pro strukturování testů a hooky jako například `before()`, `beforeEach()`, `after()`, `afterEach()`.

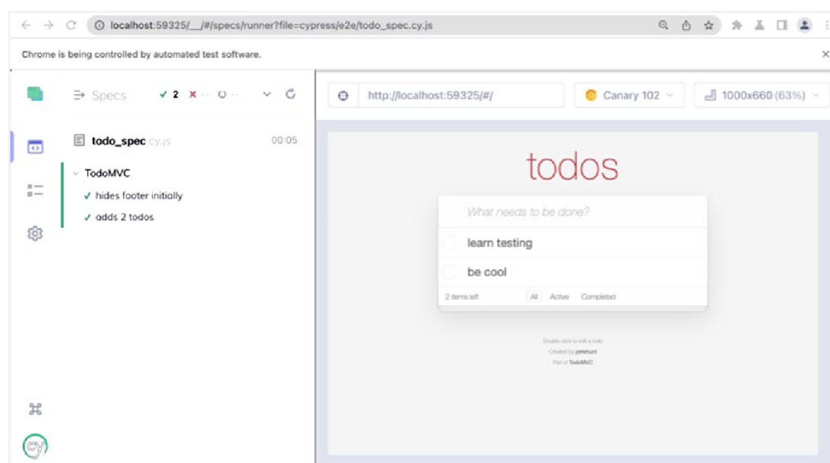
```
describe('TodoMVC', () => {
  beforeEach(() => {
    cy.visit('/')
  })

  it('hides footer initially', () => {
    cy.get('[data-testid="filters"]').should('not.exist')
  })

  it('adds 2 todos', () => {
    cy.get('[data-testid="new-todo"]').as('new').type('learn
      testing{enter}')
    cy.get('@new').type('be cool{enter}')
    cy.get('[data-testid="todo-list"] li').should('have.length', 2)
  })
})
```

Zdrojový kód 10: Ukázka struktury testu [12]

Na dané ukázce lze vidět `describe()`, který obaluje celý test, pro jehož přehlednost lze `describe()` do sebe i vnořovat. V něm se nachází hooky, v tomto případě je to `beforeEach()`, jehož příkazy se zavolají před každou `it()` částí testu. Tzn. před testováním částí „hides footer initially“ a „adds 2 todos“ Cypress navštíví adresu „/“, v tomto případě se předpokládá, že je nastavena `baseUrl`, tedy kmenová url adresa testovaného projektu, v souboru `cypress.config.js`. Následují jednotlivé testy jejichž příkazy jsou obalené v `it()`.

[3] BDD (z anglického „behaviour-driven development“) testy se používají na testování aplikace, zda se chová podle popisu.

Na následujícím Obrázku 4 se nachází výstup z testů u ukázky Zdrojového kódu 10. Lze na něm pozorovat, že všechny testy bez problému prošly, což je znázorněno zeleným pruhem. A konečný stav aplikace je takový, že obsahuje 2 úkoly, „learn testing“ a „be cool“, v TODO listu.



Obrázek 4: Ukázka výstupu testu vycházejícího z předchozího kódu [12] (upraveno autorkou práce)

Na tomto jednoduchém testu lze pozorovat, že psaní testů v Cypressu není příliš složité a hodí se tedy i pro méně zkušeného člověka.

5.3.2.2 Vytvoření vlastního jednoduchého testu

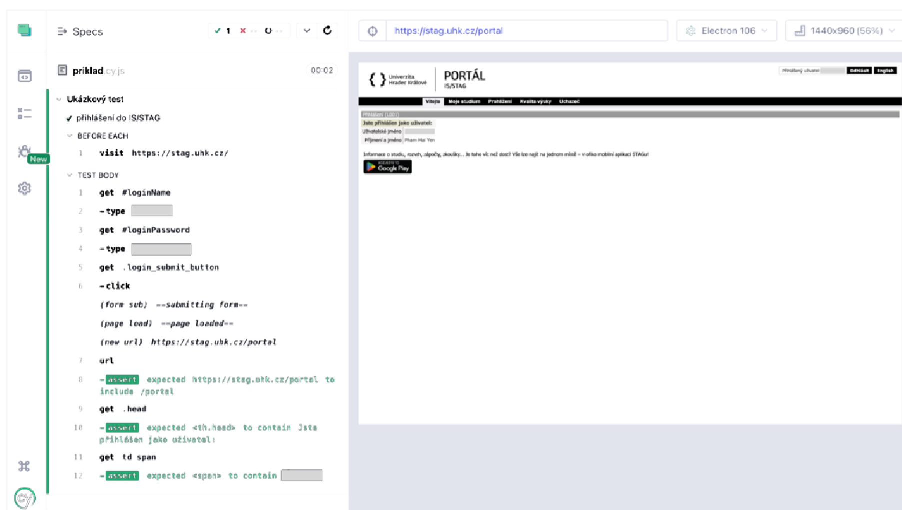
Se základní znalostí příkazů a testovací struktury je již možné se pustit do vytváření vlastního jednoduchého testovacího skriptu. V následující části práce bude ukázán krátký test, který otestuje scénář přihlašování do školního portálu IS/STAG.

Jde tedy o to navštívit stránku „`https://stag.uhk.cz/`“, zde vyplnit přihlašovací údaje a odeslat přihlašovací formulář. Po úspěšném přihlášení bychom měli být přesměrováni na „`/portal`“, kde je možné zkontrolovat, zda byl přihlášen správný uživatel například pomocí tabulky obsahující informace o přihlášeném uživateli.

```
describe('Ukázkový test', () => {
  beforeEach(() => {
    cy.visit(' https://stag.uhk.cz/')
  })

  it('přihlášení do IS/STAG', () => {
    // přihlášení
    cy.get('#loginName').type('*prihlasovaci_jmeno*')
    cy.get('#loginPassword').type('*heslo*')
    cy.get('.login_submit_button').click()

    // kontrola úspěšného přihlášení
    cy.url().should('contain', '/portal')
    cy.get('.head').should('contain', 'Jste přihlášen jako
      uživatel:')
    cy.get('td span').should('contain', '*prihlasovaci_jmeno*')
  })
})
```

Zdrojový kód 11: Ukázka testu pro přihlašování do IS/STAGu (autorka práce)

Zdrojový kód výše je jednou z možností, jak může takový test pro přihlášení vypadat. Samotná kontrola je zde provedena nejprve zjištěním, zda odpovídá url adresa, tedy jestli byla stránka přesměrována po přihlášení. Poté se zjišťuje, zda existuje tabulka, která obsahuje text „Jste přihlášen jako uživatel:“ a v neposlední řadě se kontroluje, zda odpovídá přihlašovací jméno. Výstup po spuštění testu je zachycen na následujícím obrázku, viz Obrázek 5.



Obrázek 5: Ukázka výstupu vycházející ze Zdrojového kódu 11 (autorka práce)

5.3.3 Spuštění testů

Když už jsou testy vytvořené, je na čase vytvořené testovací skripty spustit a přesvědčit se, že v testované webové aplikaci vše funguje tak, jak má. Opět existuje více možností jak na to. Testy lze spustit přímo z konzole za použití `npx`.

```
npx cypress run [options]
```

Zdrojový kód 12: Spuštění testů přímo v konzoli [12]

Je ale možné spouštět testy v samotné Cypress aplikaci s kvalitně zpracovaným uživatelským prostředím, která se z konzole spouští pomocí `npx` následujícím příkazem.

```
npx cypress open
```

Zdrojový kód 13: Otevření Cypress pomocí npx [12]

Pokud je používán místo `npm yarn`, pak se pro spuštění používá tento příkaz.

```
yarn run cypress open
```

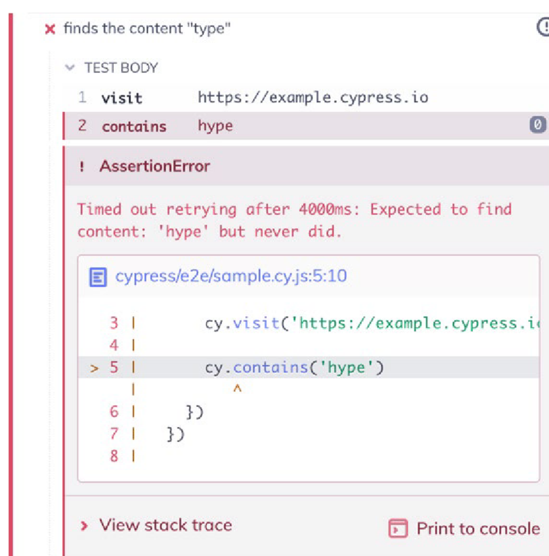
Zdrojový kód 14: Otevření Cypress pomocí yarn [12]

Po spuštění Cypressu je nejdříve na výběr mezi E2E (end-to-end) testováním a testováním komponent dále jsou na výběr prohlížeče, ve kterém chceme dané testy spustit. Momentálně jsou k dispozici pouze prohlížeče Chrome, Electron, Firefox a Edge.

5.3.4 Debuggování testů

Jak již bylo dříve zmíněno, Cypress disponuje funkcí krokování testů, které je zprostředkováno pomocí snapshotů (snímků obrazovky) pořizovaných během spuštění testů. Debuggování testů je tedy velice jednoduché.

Je-li spuštěno uživatelské rozhraní aplikace Cypress, je možné sledovat fungování testů v reálném čase a případné problémy zaznamenat ihned při testování. Pokud je ale potřeba důslednější analýza problému, je možné se prokliknout na daná problémová místa v levé části aplikace, kde je k naleznutí i krátký popis toho, co se stalo, viz Obrázek 6.



Obrázek 6: Příklad chybové hlášky spadlého testu [12] (upraveno autorkou práce)

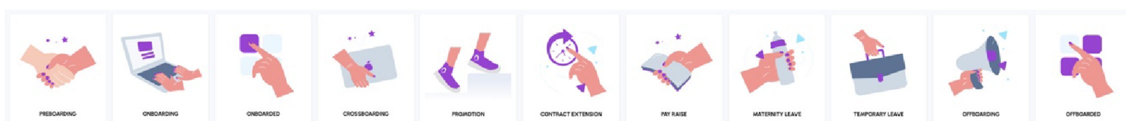
Na obrázku je znázorněna chyba testu, který měl zkontrolovat, že na stránce „`https://example.cypress.io/`" se nachází text „hype". Cypress ale po 4000ms, defaultní době čekání, toto slovo na stránce nenalezl. A proto vyhodnotil tento test jako neúspěšný, což je také znázorněno červeným křížkem. Při kliknutí na řádek „2 contains" bychom se pak pomocí snapshotu dostali na daný moment a tím si mohli ověřit, že slovo „hype" se na stránce doopravdy nevyskytuje.

Při spouštění testů v konzoli sice není k dispozici sledování průběhu v reálném čase, přesto je však jeho průběh možné později zhlédnout ve vygenerovaném videu, které se vytváří po dokončení jednotlivých testů. Pokud by test v průběhu běhu spadl, objevila se chyba, pak jsou také vygenerované snímky obrazovky z daného problémového místa.

6 Webová aplikace Onboardee.io

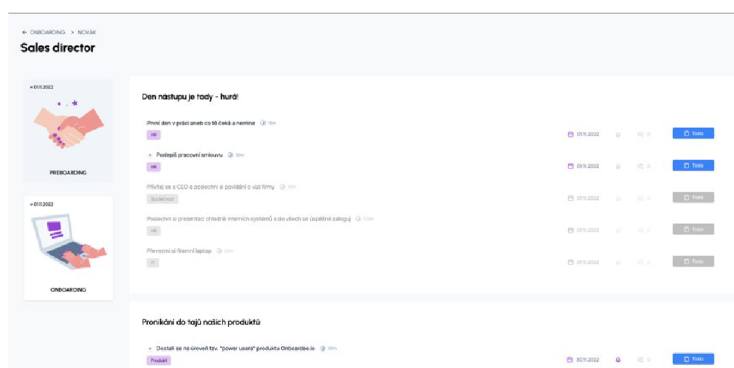
Onboardee.io je webová aplikace vyvíjená ve společnosti Recruitis.io s.r.o. na digitalizaci a modernizaci nejen nástupních procesů pro HR personalisty, ale hlavně pro samotné nastupující zaměstnance [15].

Jedná se o systém pro řízení procesů preboardingu[4], onboardingu[5] přes odchod na mateřskou až po proces offboardingu[6].



Obrázek 7: Ukázka dostupných kariérních milníků (autorka práce)

Nabízí řadu možností, jak pomoci firmám zjednodušit a vylepšit zmíněné procesy a zařídit tím zaměstnanci hladký průběh nástupu či odchodu. Pokud se zaměříme na procesy preboardingu a onboardingu, tedy na nástup nového člověka do firmy, zlepšení těchto procesů může značně přispět k rychlejší adaptaci člověka v novém prostředí, což může přispět ke spokojenosti nového zaměstnance v nové práci, a navíc urychlit jeho naskočení na ostré projekty.



Obrázek 8: Ukázka kariéry uchazeče (autorka práce)

---

[4] Preboardingem se v procesu nástupu rozumí období mezi přijetím pracovní nabídky kandidátem a jeho prvním dnem.

[5] Onboarding je proces nástupu, při kterém jsou nováčci integrováni do organizace a pracovní role.

[6] Offboarding zaměstnance je proces odchodu, kdy zaměstnanec ukončuje zaměstnanecký poměr, opouští organizaci.

6.1 Technické specifikace

Po technické stránce je webová aplikace Onboardee.io postavena na následujících technologiích. Základním kamenem celé aplikace je MeteorJS. Jedná se o full-stack[7] JavaScriptovou platformu využívající se pro vývoj moderního webu a mobilních aplikací. V rámci backendu je využíván Node.js, open-source serverové prostředí. Co se týče dat, ty jsou uloženy v NoSQL databázi MongoDB. Tato databáze běží v prostředí škálované cloudové služby MongoDB Atlas. Frontend komunikuje pomocí WebSockets, počítačovým komunikačním protokolem, s backendem a běží na React.js, JavaScriptové knihovně využívané pro vývoj UI.

Aplikace je nasazena na AWS, neboli webových službách Amazonu (z anglického Amazon Web Services), což je v současnosti nejrozvinutější cloudová platforma pro poskytování online služeb od společnosti Amazon.com. Z důvodu globálních ambicí tohoto projektu bylo zvoleno horizontální škálování projektu s ohledem na geolokaci klientů přistupujících do aplikace. To znamená, že při zvýšení kapacity stávajícího hardwaru či softwaru dochází k přidání dalších uzlů namísto přidávání nových zdrojů do původního uzlu, jako tomu je při vertikální škálovatelnosti [16]. Distribuci požadavků mezi uzly realizuje aplikace pomocí služby AWS Elastic Load Balancer.

Na monitorování aplikace je používán Monti APM, který je určen na monitorování aplikací založených právě na MeteorJS, a to pro zlepšení výkonnosti aplikací a lehčí opravu chyb [17].

6.2 Požadavky na testování

Jelikož je aplikace a její funkce velice obsáhlá, není v lidských silách celou aplikaci otestovat manuálně, navíc je to velice časově náročné. Proto se začaly řešit automatizované testy, které by kompletně otestovaly funkcionalitu aplikace

[7] Full-stack vývoj (z anglického „full-stack development“) je vývoj, který zahrnuje frontend a backend zároveň.

před vydáním na produkci, tedy zavedením do reálného provozu. Požadavky na testovací nástroj a samotné testy byly následující:

- jednoduchý testovací nástroj na pochopení, psaní a údržbu testů v budoucnosti,
- testy simulující uživatelské chování,
- schopnost integrace s Gitlab CI/CD[8],
- otestovat celou aplikaci od začátku až do konce,
- ale také mít možnost spouštět testy na konkrétní části systému.

První požadavky byly splněné výběrem testovacího nástroje Cypress, který je jak jednoduchý na pochopení, tak i na psaní testů a jejich údržbu. Navíc se specializuje na end-to-end testování, které ověřuje kompletní procesy aplikace a simuluje chování skutečného uživatele. A v neposlední řadě disponuje schopností se integrovat s Gitlab CI/CD. Poslední požadavky jsou pak otázkou implementace daných testů, kterou byl pověřen právě autor této bakalářské práce. Samotným vytvořením Cypress testů se bude zabývat následující kapitola.

[8] CI/CD (z anglického „continuous integration/continuous delivery") je proces, který umožňuje nepřetržité dodávání aplikací k testování. CI je filozofie kódování, či sada postupů a CD automatizuje doručování aplikací do vybraných infrastrukturních prostředí.

7 Návrh a implementace automatizovaných testů

Následující část bakalářské práce se věnuje vytvářením testů pro zmíněnou aplikaci Onboardee.io. Nejdříve se zaměří na analýzu aplikace a požadavků na testování a poté se bude věnovat konkrétně jejich návrhem a následně jejich implementací s cílem vytvořit automatizované testy ověřující funkčnost a spolehlivost aplikace.

7.1 Testovací scénáře

Jak již bylo zmíněno nejdříve bude práce zaměřena na analýzu aplikace a jejích požadavků na testování. Poté bude následovat návrh testovacích scénářů, které budou později implementovány.

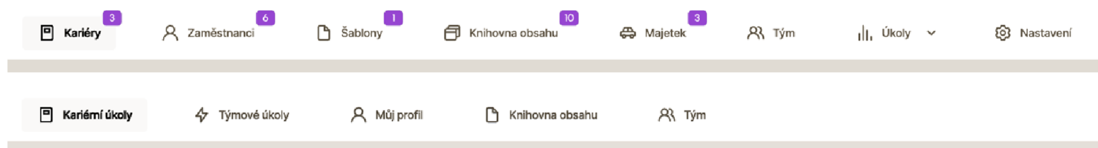
7.1.1 Analýza aplikace

Onboardee.io je cloudová webová aplikace, která je postavena na MeteorJS s využitím React.js na frontendu.

V současné době nabízí pouze 2 základní role: administrátor a zaměstnanec. Administrátorský účet je určen zejména HR personalistům, kteří mají nábor na starosti. Zatímco zaměstnanecký účet je určen pro všechny ostatní osoby, zejména pro nastupující nováčky, ale také i pro další zaměstnance společnosti. Až na možnost editace svých osobních údajů mají oba účty různá práva.

Administrátor má přístup k většině modulů a jejich náležitých funkcí. Ty zahrnují přehled o probíhajících kariérách zaměstnanců, přehled všech zaměstnanců a jejich správa, možnost vytváření šablon pro kariéry, vytváření obsahu do úkolů, vytváření kvízů, formulářů, či dokonce smluv. Disponuje také právem pro správu majetku, například firemních aut, počítačů a dále. Další funkcionalitou je přehled a správa úkolů, možnost se seznámit se svými kolegy z týmu a v neposlední řadě má přístup k nastavení firemního Onboardee.io.

Zaměstnanec má omezená práva a má přístup pouze ke své aktivní kariéře, pokud ji má přiřazenou. Dále má přístup k úkolům, které byly přiřazené jeho osobě, k obsahu, jenž byl administrátorem přidán do knihovny, a může si prohlídnout své kolegy z týmu.



Obrázek 9: Rozdíl mezi administrátorským a zaměstnaneckým účtem (autorka práce)

Na Obrázku 9, lze pozorovat rozdíl mezi administrátorským (vrchní část obrázku) a zaměstnaneckým účtem (spodní část obrázku) v počtu modulů.

Co se týče požadavků na testování, je potřeba vytvořit testy simulující uživatelské používání aplikace, které otestují celou aplikaci, ale přitom dovolí i pouze částečné otestování konkrétní části aplikace. A to z důvodu, že vývojář provedl pouze malou změnu v jednom z modulů a chce si jen narychlo ověřit, že jeho změna nezpůsobila chybu.

Testy musí být přitom dostatečně členěny a části náležitě označeny, aby z výstupu testů bylo jednoznačně jasné, na čem daný test selhal a bylo v případě nepoužitelného videa, či snímků obrazovky, jednoduše a rychle dohledatelná příčina pádu testu.

Posledním požadavkem vytvořených automatizovaných testů pro Onboardee.io je integrace s Gitlab CI/CD, díku kterému je možné zakomponovat testy přímo do vývoje a tím zautomatizovat a zjednodušit celý proces, čímž šetří nejen čas, ale také prostředky.

7.1.2 Návrh testovacích scénářů

Po důkladné analýze aplikace a požadavků na testování v minulé kapitole jsou testovací scénáře navrhnuty následovně.

Každý vytvořený testovací scénář je zaměřený na otestování právě jednoho modulu aplikace. Testová sada tedy obsahuje testové scénáře na moduly `Kariéry`, `Zaměstnanci`, `Šablony`, `Knihovna obsahu`, `Majetek`, `Tým`, `Úkoly`, `Nastavení` a `Profil` z pohledu administrátora. Z pohledu zaměstnance či nováčka pak obsahuje moduly `Kariérní úkoly`, `Týmové úkoly`, `Můj profil`, `Knihovna obsahu` a `Tým`. Seznam všech testů zachycuje následující tabulka níže.

Kód	Název	Testovací URL	Testovací případy
TS_A_01	profile	`/profile/*`	změna údajů/jazyka/hesla
TS_A_02	careers	`/`	plnění úkolů v kariéře
TS_A_03	employees	`/employees`	vytvoření a správa zaměstnanců
TS_A_04	templates	`/templates`	vytvoření šablony/milníků/…
TS_A_05	content_library	`/content`	vytvoření obsahu/smluv/…
TS_A_06	inventory	`/inventory`	vytvoření a správa inventáře
TS_A_07	team	`/team`	testování viditelnosti týmu
TS_A_08	task_center	`/dashboard/*`	filtrování úkolů
TS_A_09	settings	`/settings/*`	nastavení konfigurace/…
TS_E_01	career_tasks	`/`	plnění úkolů v kariéře
TS_E_02	team_tasks	`/my-tasks`	plnění ostatních úkolů
TS_E_03	my_profile	`/profile/*`	změna údajů/jazyka/hesla
TS_E_04	content_library	`/wiki`	procházení obsahů
TS_E_05	team	`/team`	prohlížení týmu

Tabulka 1: Seznam testovacích scénářů (autorka práce)

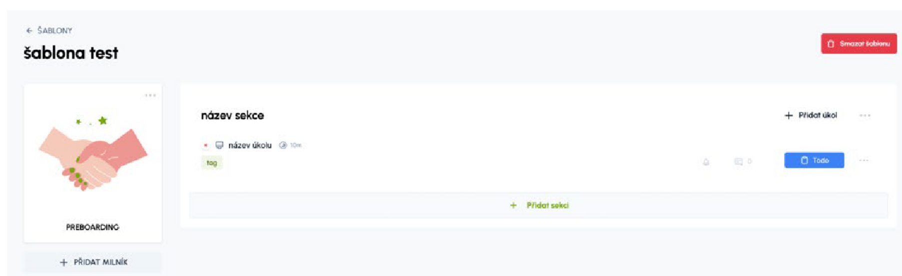
Z důvodu možnosti spouštět jednotlivé testy také samostatně, dochází na začátku každého testovacího scénáře k vytvoření a nastavení potřebných položek. Ty jsou pak na konci testů zase všechny smazány a nastaveny na defaultní nastavení. Testy tedy po sobě uklízí workspace (pracovní prostor), aby byl čistý a připraven pro další testování.

Hlavní kostra testovacích scénářů obsahuje následující části:

1. přihlášení do aplikace,
2. nastavení potřebných položek k testování,
3. přesměrování se na testovací modul,
4. samotné testování,
5. smazání všech nastavení.

7.1.2.1 Testovací scénář TS_A_04

Modul `Šablony` je určen pro vytvoření šablony kariéry a její správu. V tomto modulu aplikace je možné vytvářet šablony kariéry, do nich přidávat milníky, do těch přidávat sekce a do sekcí jednotlivé úkoly pro nastupující zaměstnance.



Obrázek 10: Ukázka šablony (autorka práce)

V tabulce níže, je ukázka testovacího scénáře na modul `Šablony` z pohledu administrátora, TS_A_04. Jsou v ní znázorněné podčásti kroku 2, kde se nastavuje workspace pro testování v kroku 4.

Krok	Akce	Očekávaný výsledek
1	přihlásit se	administrátor přihlášen do aplikace
2.1	nastavit konfiguraci	úspěšně nastavena konfigurace
2.2	nastavit struktury	úspěšně vytvořené projektové a geo struktury
2.3	vytvořit obsah	úspěšně vytvořený obsah
3	přesměrovat se na /templates	úspěšně přesměrováno do modulu šablon /templates
4	vytvořit novou šablonu	úspěšně vytvořená šablona
...	...	...
4.9	přidat úkol	úspěšně přidaný úkol
...	...	...
4.12	smazat milník	úspěšně smazaný milník
4.13	smazat šablonu	úspěšně smazaný šablona
5.1	smazat obsah	úspěšně smazaný obsah
5.2	smazat konfiguraci	úspěšně smazaná konfigurace

Tabulka 2: Seznam testovacích případů v TS_A_04_templates (autorka práce)

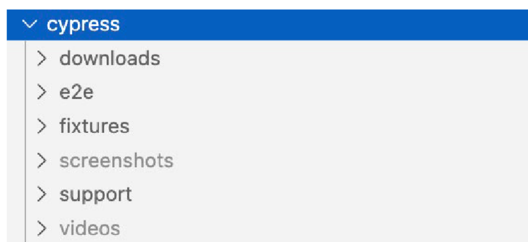
Ostatní testovací scénáře vypadají velice podobně, jako je znázorněno v příkladu v Tabulce 2 a liší se hlavně v kroku čtvrtém, tedy v samotných testech.

7.2 Implementace testů

Tato část práce nejdříve poskytuje náhled do projektu a přibližuje jeho členění, a poté je zaměřen na samotnou implementaci testů pro Onboardee.io. Přičemž se nejdříve zaměří na jeden konkrétní test a poté poukáže na zajímavé části kódu, které stojí za zmínku a je vhodné je dále přiblížit.

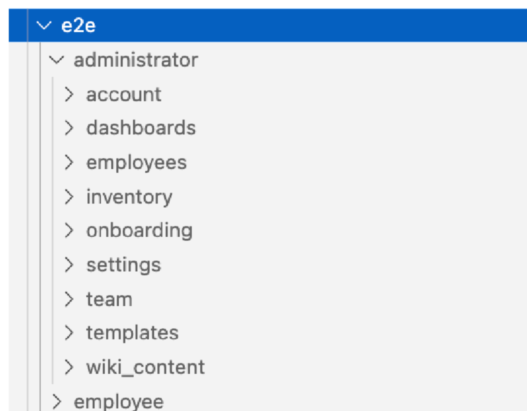
7.2.1 Struktura projektu

Hlavní část projektu, kde se nachází všechny testy, je k nalezení ve složce `cypress`, která se vytvoří společně s její vnitřní strukturou při instalaci Cypressu. Struktura obsahuje podsložky `downloads`, pro stažené soubory, `e2e`, kde se nachází samotné spustitelné testy, `fixtures` obsahující testovací data či různé soubory používané při testování či `support`, která obsahuje pomocné testovací funkce. Dále jsou zde k naleznutí složky `screenshots` a `videos`, kam se nahrávají snímky obrazovky a videa z testování.



Obrázek 11: Struktura projektu (autorka práce)

Jak již bylo zmíněno, vytvořené testy se nachází ve složce `e2e`, složka `e2e` tohoto konkrétního projektu vypadá následovně, viz Obrázek 12. Hlavní 2 podsložky rozdělují na testy z pohledu administrátora, `administrator`, a z pohledu zaměstnance, `employee`. Administrátor je poté rozdělen podle jednotlivých modulů aplikace na dalších 9 podsložek.



Obrázek 12: Struktura e2e složky s testy (autorka práce)

7.2.2 Implementace testovacího scénáře TS_A_04

Následující část je zaměřena na implementaci testovacího scénáře, konkrétně TS_A_04, tedy scénář ověřující funkčnost modulu `Šablony` testované webové aplikace.

Implementace se řídí kroky Tabulky 2. Nejprve je tedy potřeba se přihlásit do aplikace, přesněji do testovacího pracovního prostoru. Toho je docíleno pomocí session, Cypressové funkce, která je blíže přiblížena v následující kapitole 7.4. A jelikož je přihlašování potřeba provést před všemi testovacími scénáři, tento `beforeEach()` hook se nachází ve speciálním souboru `e2e.js`, který se spouští automaticky před všemi testy. Díky němu tedy nedochází k opakování kódu.

Není to potřeba, ale pro jistotu se před spuštěním každého testového scénáře vymažou také všechny uložené session pomocí `clearAllSavedSession()` v `before()` hooku. Výsledný kód v souboru e2e.js je možné pozorovat níže, viz Zdrojový kód 15.

```
before(() => {
  Cypress.session.clearAllSavedSessions()
})

beforeEach(() => {
  cy.login_sesh()
})
```

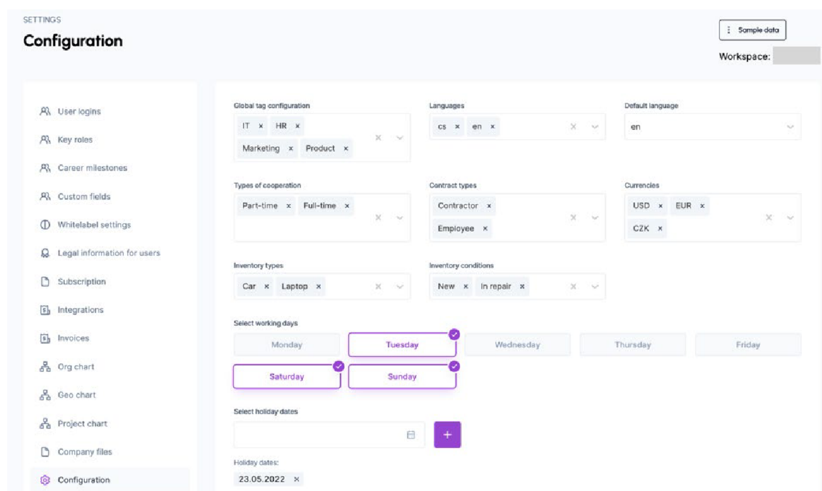
Zdrojový kód 15: Mazání předchozí session a přihlašování řešené v e2e.js (autorka práce)

Druhým krokem je nastavení pracovního prostoru (workspace) a vytvoření potřebných částí, které jsou nezbytné pro otestování Šablon. Jednou z hlavních částí je nastavení konfigurace, která je zajištěna následujícím kódem, viz Zdrojový kód 16.

```
it('Sets configuration', () => {
  // kontrola načtení stránky
  cy.visit('/settings/configurations')
  cy.get('#mainContent > div.login-spinner').should('not.exist')

  // konfigurace nastavení
  configuration.addItem(confInfo["global tags"], 'IT')
  ...

  configuration.addItem(confInfo["default language"], 'en')
  ...

  configuration.selectWorkingDays([1, 5, 6])

  configuration.selectHolidays('23.05.2022')
})
```

Zdrojový kód 16: Ukázka nastavení konfigurace (autorka práce)

V předchozí ukázce kódu výše lze spatřit funkce `addItem(info, item)`, `selectWorkingDays(workdays)` a `selectHolidays(date)`. Jedná se o funkce, které jsou naimportovány ze souboru `configuration.js` nacházející se ve složce `support`, kde lze nalézt další pomocné funkce. Cílem je totiž zamezit opakování častého kódu. Výsledný stav aplikace je následovný, viz Obrázek 13.



Obrázek 13: Výstup ze Zdrojového kódu 16 (autorka práce)

Po nastavení konfigurace je potřeba vytvořit projektové struktury, geo struktury a obsah, který se bude později vkládat do vytvořených úkolů, viz Zdrojový kód 17.

```
describe('Sets up workspace', () => {
  set.setConfiguration()
  set.setCharts()
  set.createContents()
})
```

Zdrojový kód 17: Nastavení pracovního prostoru pro TS_A_04 (autorka práce)

Funkce `setConfiguration()`, `setCharts()`, `createContents()` jsou opět pomocné funkce ze složky `support`.

Na řadu přichází samotné testování.

```
describe('Templates', () => {
  it('Creates a new template', () => {
    template.createTemplate('template#1', 'amazing temp')
  })

  it('Renames template', () => {...})
  it('Delete initial milestones', () => {...})
  it('Creates a milestone', () => {...})
  it('Creates a section', () => {...})
  it('Renames section', () => {...})
  it('Creates an empty task and internal task', () => {...})
  it('Renames task and internal task', () => {...})
  it('Creates a task', () => {...})
  it('Deletes internal task and task', () => {...})
  it('Deletes section', () => {...})
  it('Deletes milestone', () => {...})
  it('Deletes template', () => {...})
})
```

Zdrojový kód 18: Testovací případy pro testování šablon (autorka práce)

Jak lze pozorovat na ukázce kódu výše, viz Zdrojový kód 18, struktura testů je pro přehlednost rozčleněna do více `it()` bloků, čímž umožňuje rychlé vytyčení problému a jednoduchou orientaci v kódu při výskytu chyby. Bohužel je to na úkor rychlosti testů, protože každý `it()` blok je Cypressem považován jako samostatný test. Rychlost je samozřejmě důležitý aspekt, důraz je ale v těchto testech kladen více na přehlednost testů, proto byl zvolen tento přístup.

Detailní ukázka použité pomocné funkce, která je určena pro vytvoření šablony, `createTemplate(templateName, description)`, vypadá v kódu následovně, viz Zdrojový kód 19.

```
export function createTemplate(templateName, description) {
  cy.contains('Add').click()
  cy.get('div[id^="headlessui-dialog-panel"]').should('be.visible')

  // vyplnění formuláře
  cy.get('[name="name"]').type(templateName)
  cy.get('[name="description"]').type(description)
  cy.selectOption('Position in geo chart', 'PRAGUE')
  cy.selectOption('Template language', 'en')

  // odeslání formuláře
  cy.get('[type="submit"]').click()

  // kontrola
  cy.url().should('contain', '/templates')
  cy.get('body tr').should('be.visible').and('contain',templateName)
}
```

Zdrojový kód 19: Ukázka pomocné funkce createTemplate() (autorka práce)

Funkce klikne na tlačítko „`Add`" pro přidání nové šablony, přičemž by mělo vyskočit dialogové okno s formulářem. Ten je poté vyplněn údaji o jméně a popisu, geo struktuře, pro kterou je tvořena, a o jazyku šablony. Poté dochází k odeslání formuláře a kontrole zda se šablona vytvořila a má záznam v tabulce šablon.

Následuje poslední část testovacího scénáře pro testování modulu `Šablony`, kde dochází ke smazání obsahu, který byl při tvoření úkolů v šabloně využíván, a následně dochází ke smazání nastavení konfigurace, viz Zdrojový kód 20.

```
describe('Cleans up workspace after tests', () => {
  set.deleteContents()
  set.deleteConfiguration()
})
```

Zdrojový kód 20: Ukázka úklidu na konci testu (autorka práce)

Podle Tabulky 2 došel vyčištěním pracovního prostoru (workspace) testovací scénář do svého konce. Na obrázku níže je ukázka výstupu tohoto testovacího scénáře při spuštění v konzoli. Všechny části testu úspěšně prošly. Modul `Šablony` aplikace Onboardee neobsahuje v momentě spuštění testů žádné problémy, viz Obrázek 14.

```
====================================================================================================

  (Run Starting)

  ┌────────────────────────────────────────────────────────────────────────────────────────────────┐
  │ Cypress:        12.6.0                                                                         │
  │ Browser:        Electron 106 (headless)                                                        │
  │ Node Version:   v18.12.1 (/usr/local/bin/node)                                                 │
  │ Specs:          1 found (templates.cy.js)                                                      │
  │ Searched:       cypress/e2e/administrator/templates                                            │
  │ Experiments:    experimentalMemoryManagement=true                                              │
  └────────────────────────────────────────────────────────────────────────────────────────────────┘


────────────────────────────────────────────────────────────────────────────────────────────────────

  Running:  templates.cy.js                                                                 (1 of 1)


  Templates
    Sets up workspace
      ✓ Sets configuration (37552ms)
      ✓ Sets charts (6392ms)
      ✓ Creates contents: Creates contents (48715ms)
    Templates
      ✓ Creates a new template (3542ms)
      ✓ Renames template (7314ms)
      ✓ Delete initial milestones (5973ms)
      ✓ Creates a milestone (5728ms)
      ✓ Creates a section (5513ms)
      ✓ Renames section (7586ms)
      ✓ Creates an empty task and internal task (16800ms)
      ✓ Renames task and internal task (6756ms)
      ✓ Creates a task (8711ms)
      ✓ Deletes internal task and task (7051ms)
      ✓ Deletes section (5195ms)
      ✓ Deletes milestone (4961ms)
      ✓ Deletes template (5049ms)
    Cleans up workspace after tests
      ✓ Deletes created contents (13096ms)
      ✓ Deletes configuration (5914ms)


  18 passing (3m)


  (Results)

  ┌────────────────────────────────────────────────────────────────────────────────────────────────┐
  │ Tests:        18                                                                               │
  │ Passing:      18                                                                               │
  │ Failing:      0                                                                                │
  │ Pending:      0                                                                                │
  │ Skipped:      0                                                                                │
  │ Screenshots:  0                                                                                │
  │ Video:        false                                                                            │
  │ Duration:     3 minutes, 23 seconds                                                            │
  │ Spec Ran:     templates.cy.js                                                                  │
  └────────────────────────────────────────────────────────────────────────────────────────────────┘


====================================================================================================

  (Run Finished)


       Spec                                              Tests  Passing  Failing  Pending  Skipped
  ┌────────────────────────────────────────────────────────────────────────────────────────────────┐
  │ ✔  templates.cy.js                          03:23       18       18        -        -        - │
  └────────────────────────────────────────────────────────────────────────────────────────────────┘
    ✔  All specs passed!                        03:23       18       18        -        -        -
```

Obrázek 14: Ukázka úspěšného projití testu TS_A_04 (autorka práce)

7.2.3 Zajímavé části kódu

Následující kapitola je věnována některým zajímavým částem kódu vzniklým při vytváření testů, které stojí za zmínku.

Jedním z takových je funkce, která vybírá možnost z rozbalovacího seznamu. V průběhu psaní testů, byla využívána tak často, že byla přidána do vlastních Cypressových příkazů, do souboru `commands.js` nacházející se ve složce `support`, viz Zdrojový kód 21.

```
Cypress.Commands.add('selectOption', (selectName, optionName) => {
  cy.get('label:contains("' + selectName + '") + ').last().click()
  cy.get('.selectTw__menu-list').should('be.visible')
  cy.get('.selectTw__option:contains("' + optionName + '")')
    .should('exist').click()
  cy.get('label:contains("' + selectName + '"):last ~ div,
    .selectTW__single-value').should('contain', optionName)
})
```

Zdrojový kód 21: Ukázka vytvoření vlastního Cypress příkazu pro selectOption() (autorka práce)

Příkaz `selectOption` nejprve klikne na `select` element s daným popisem, to otevře seznam možností, z nějž vybere danou možnost a následně zkontroluje zda byla možnost doopravdy zvolena.

Další část kódu, který si zasloužil místo ve vlastních příkazech v `commands.js`, je příkaz na mazaní. Na různých místech aplikace dochází k mazání, které je svým průběhem identické jako na jiných místech. Po kliknutí na tlačítko pro smazání, které se může lišit obsahem, se objeví dialog, který se ujišťuje, že uživatel tuto akci opravdu chce provést. Kód níže, Zdrojový kód 22, tuto akci zachycuje.

```
Cypress.Commands.add('delete', (deleteButtonName) => {
  cy.get('button:contains("' + deleteButtonName + '")').click()
  cy.get('div[id^="headlessui-dialog-panel"]').should('be.visible')
  cy.get('button:contains("Confirm")').click()
})
```

Zdrojový kód 22: Ukázka vytvoření vlastního Cypress příkazu pro delete() (autorka práce)

Další zajímavý kousek kódu je funkce, která je určena k přejmenování položky v organizační/geo/projektové struktuře. Z dosud neznámých důvodů se část nového jména neukládala. Bylo tedy potřeba přidat prodlevu při psaní nového jména. To samotné však nestačilo a bylo třeba vložit příkaz `wait(2000)`, který čeká na aplikaci 2 sekundy, aby test nespadl.

```
export function renameItem(itemName, newName) {
  // kliknutí na editaci
  cy.get('div:contains("' + itemName + '"):parent + div > div >
    button:first').click({ force: true })

  // vymaže pole, vyplní ho novým jménem
  cy.get('input').should('have.value',itemName).clear()
    .type(newName, { delay: 500 }).then(() => {
    cy.get('input').should('have.value',newName).type('{enter}')
      .then(() => {
      cy.wait(2000)
      cy.get('ol').should('contain', newName)
    })
  })

  cy.reload()
  cy.wait(2000)
  cy.get('ol').should('contain', newName)
}
```

Zdrojový kód 23: Ukázka funkce renameItem() (autorka práce)

7.3 Spuštění vytvořených testů

Po vytvoření daných testů je potřeba je spustit, aby byly k užitku. Jak testy spustit?

Při spouštění testů v konzoli, či spuštění uživatelského prostředí Cypressu, je potřeba programu předat Cypressovou proměnnou prostředí (z anglického „environment variables“) s názvem `WORKSPACE_NAME`. Kde `testXXXXXX` značí jméno testovacího prostoru (workspace) daný náhodně vygenerovaným 6místným číslem.

Sada vytvořených testů je možné spustit z konzole pomocí příkazu níže.

```
CYPRESS_WORKSPACE_NAME=testXXXXXX npx cypress run
```

Zdrojový kód 24: Spuštění sady vytvořených testů v konzoli (autorka práce)

Pro spouštění určitého testu, ne celé sady, je potřeba přidat k příkazu výše specifikaci souboru pomocí `--spec „cypress/e2e/*/*/*.cy.js“`, viz příkaz níže.

```
CYPRESS_WORKSPACE_NAME=testXXXXXX npx cypress run --spec
„cypress/e2e/*/*/*.cy.js“
```

Zdrojový kód 25: Spuštění konkrétního vytvořeného testu (autorka práce)

Pro spuštění testů přes uživatelské prostředí Cypressu lze použít následující příkaz.

```
CYPRESS_WORKSPACE_NAME=testXXXXXX npx cypress open
```

Zdrojový kód 26: Spuštění uživatelského prostředí Cypressu (autorka práce)

7.4 Problémy v průběhu psaní testů

V průběhu psaní testů nastaly určité problémy, které vedly k nekonzistentnosti testů. Některé problémy je možné vyřešit pomocí funkcí, které nabízí samotný Cypress, jiné vyžadují zásah do samotného zdrojového kódu.

Problém, který se vyskytl na začátku testování, byl spojen s ukládáním dat do lokálního úložiště[9], respektive jeho neukládáním. Testy proto padaly, protože se v průběhu testů aplikace náhodně odhlašovala. Naštěstí Cypress nabízí příkaz `cy.session()`, který tento problém vyřešil.

```
cy.session(id, setup)
cy.session(id, setup, options)
```

Zdrojový kód 27: Ukázka použití příkazu .session() [12]

V souboru `commands.js` bylo potřeba vytvořit vlastní příkaz , který byl v projektu pojmenován jako „`login_sesh`", využívaný pro přihlašování pomocí `session()`. Níže je zjednodušená ukázka použití v projektu, viz Zdrojový kód 28.

[9] Lokální uložiště (z anglického „local storage") je uložiště v prohlížeči klienta přístupný JavaScriptem, který se využívá v případě, že je potřeba uložit návštěvníkovi data, která není potřeba přenášet na server.

```
Cypress.Commands.add('login_sesh', () => {
  cy.session('login_session', () => {
    ...

    // přesměrování na přihlašovací stránku
    cy.visit('/login')
    cy.url().should('equal', 'https://app.onbee.link/login/')

    // přihlášení
    cy.get('[name="username"]').type('*prihlasovaci_jmeno*')
    cy.get('[name="password"]').type('*heslo*')
    cy.get('form [type="submit"]').click()

    // kontrola přihlášení
    cy.url().should('contain', Cypress.config('baseUrl')).then(()=>{
      ...

      cy.get('svg.object-contain').should('exist').and('be.visible')
    })
  })
})
```

Zdrojový kód 28: Zjednodušená ukázka použitého cy.session() (autorka práce)

Další problém je takový, který provází tento projekt již od začátku a prozatím, není nijak řešený. Jedná se o chybějící testovací selektory. Momentálně jsou testy závislé na CSS selektorech, které se ale v průběhu vývoje mohou volně měnit a tím narušit průběh spouštěných testů.

```
export function deleteModule(milestoneName) {
  ...

  cy.get('li > div:contains("Delete")').click()
  ...
}
```

Zdrojový kód 29: Ukázka z funkce deleteModule(milestoneName) (autorka práce)

Na ukázce Zdrojový kód 29 je pozorovatelné, že je selektor závislý na struktuře kódu „`li > div`" a na samotným názvu daného elementu „`:contains("Delete")`". Pokud by tedy došlo ke změně struktury, či pouhé změně textu, pak by testy selhaly a způsobily problém.

Podle dokumentace Cypressu a jejich tipů, by se měl používat atribut `data-*`, například `data-cy`, který by nepodléhal změnám a díky kterému by se jednodušeji odkazoval na testovaný element.

8 Shrnutí výsledků

Jak již bylo zmíněno v průběhu bakalářské práce, testování má klíčový význam pro zajištění funkčnosti a spolehlivosti každého softwaru. Implementace automatizovaných testů navíc šetří čas a prostředky, které by byly potřeba při manuálním testování.

Tato bakalářská práce měla za cíl takové automatizované testy vytvořit a to konktrétně pro webovou aplikaci Onboardee.io ve společnosti Recruitis.io. K vytvoření testů byl využit testovací framework Cypress.

Testy jsou za pomocí integrace CI/CD zaintegrovány do Gitlabu a tímto způsobem spouštěny v Electronu při každém vydání nové změny. Jejich průběh trvá přibližně 1,5 hodiny při úspěšném projití. Neúspěšné testy celkový průběh testování značně prodlužují. Může se proto stát, že testy vůbec nedojedou do konce, protože vyprší limit 4 hodin nastavených pro běh testů. Níže je přiložený obrázek s výsledky testovací sady, viz. Obrázek 15.

(Run Finished)

	Spec		Tests	Passing	Failing	Pending	Skipped
✔	employee/TS_E_01_career_tasks.cy.js	03:10	18	18	-	-	-
✔	employee/TS_E_02_team_tasks.cy.js	03:10	17	17	-	-	-
✔	employee/TS_E_03_my_profile.cy.js	02:39	19	19	-	-	-
✔	employee/TS_E_04_content_library.cy.js	03:18	20	20	-	-	-
✔	employee/TS_E_05_team.cy.js	02:03	13	13	-	-	-
✔	administrator/account/TS_A_01_profile.cy.js	01:41	14	14	-	-	-
✔	administrator/dashboards/TS_A_08_task_center.cy.js	04:10	19	19	-	-	-
✔	administrator/employees/TS_A_03_employee.cy.js	10:05	45	45	-	-	-
✔	administrator/inventory/TS_A_06_inventory.cy.js	02:35	17	17	-	-	-
✔	administrator/onboarding/TS_A_02_careers.cy.js	03:58	18	18	-	-	-
✔	administrator/team/TS_A_07_team.cy.js	01:55	12	12	-	-	-
✔	administrator/templates/TS_A_04_templates.cy.js	03:51	18	18	-	-	-
✔	administrator/wiki_content/TS_A_05_01_content_form.cy.js	01:59	13	13	-	-	-
✔	administrator/wiki_content/TS_A_05_02_content_content.cy.js	03:33	31	31	-	-	-
✔	administrator/wiki_content/TS_A_05_03_content_signature.cy.js	01:11	9	9	-	-	-
✔	administrator/settings/TS_A_09_settings.cy.js	05:57	41	41	-	-	-
✔	All specs passed!	55:21	324	324	-	-	-

Obrázek 15: Výsledky vytvořené testovací sady (autorka práce)

Vytváření testů však provází problém nekonzistence z důvodu chybějících testovacích selektorů, které by nepodléhali změnám, ať už stylu, nebo popisků. Jelikož jako tester nemám přístup ke zdrojovému kódu projektu, není v mé kompetenci tento problém napravit. Je však nutné se na tento problém zaměřit v následujícím vývoji testů.

Testy přes veškeré problémy značně šetří čas a prostředky potřebné k otestování aplikace, která je velmi rozsáhlá. Ve společnosti Recruitis.io s.r.o. jsou proto Cypress testy rozšiřovány i na testování jejich staršího produktu Recruitis.io.

9 Závěr

Cílem této práce bylo seznámit čtenáře s tématem automatizovaného testování a vytvořit sadu automatizovaných testů pro webovou aplikaci Onboardee.io ve společnosti Recruitis.io s.r.o. Výstupem práce je tedy sada Cypress testů, které pokrývají dané scénáře používání zmíněné webové aplikace. Testy kontrolují základní operace jako přihlášení uživatele, ale hlavně testují funkcionalitu jednotlivých modulů aplikace a jejich provázanost.

Teoretická část práce shrnula poznatky o testování softwaru s důrazem na automatizované testování. Dále přiblížila samotný testovací framework Cypress, ve kterém jsou výsledné testy vytvořeny.

Vytvořené testy popsané v praktické části práce, jsou pomocí CI Gitlab zasazené do procesu vývoje a momentálně se spouští při každém nasazení nějaké změny. Vývojáři jsou tedy schopni si zkontrolovat, zda jejich úprava nezpůsobila zanesení chyb.

Automatizované testy, i přes menší problémy výrazně šetří čas, který by byl nutný pro manuální otestování funkčnosti aplikace. A budou se proto v budoucnosti dále vyvíjet a rozšiřovat podle potřeb a požadavků aktuální verze webové aplikace.

10 Seznam použité literatury

[1] PATTON, Ron. Software testing. Indianapolis: Sams Publishing, 2006. ISBN 0-672-32798-8.

[2] SOKOL, Martin. Automatické testování webových aplikací [online]. Brno, 2013 [cit. 25. 01. 2023]. Diplomová práce. Masarykova univerzita, Fakulta informatiky. Dostupné z: https://is.muni.cz/th/awomw/diplomova_praca.pdf

[3] STOTTLEMYER, Diane. Automated Web testing toolkit: expert methods for testing and managing Web applications. New York: John Wiley & Sons, Inc., 2001. ISBN 0-471-41435-2.

[4] KHAN, Mohd Ehmer and KHAN, Farmeena. A comparative study of white box, black box and grey box testing techniques. International Journal of Advanced Computer Science and Applications. 2012. ISSN 2156-5570.

[5] HLAVA, Tomáš. Testování softwaru: Portál zabývající se problematikou ověřování kvality software. Manuální i automatizované testování [online]. 2011 [cit. 26. 06. 2022]. Dostupné z: http://testovanisoftwaru.cz

[6] PIETRIK, Michal. Automatizované testování webových aplikací [online]. Brno, 2012 [cit. 27. 06. 2022]. Disertační práce. Masarykova univerzita, Fakulta informatiky. Dostupné z: https://is.muni.cz/th/yh8ir/_DP_Pietrik.pdf

[7] SUCHOMEL, Vít. Automatické testovací nástroje DIVINE [online]. Brno, 2007 [cit. 29. 06. 2022]. Bakalářská práce. Masarykova univerzita, Fakulta informatiky. Dostupné z: https://is.muni.cz/th/bqxcx/Automaticke_testovani_nastroje_DiVinE.pdf

[8] FERNANDEZ, Tomas. The 6 principles of Test Automation. [online]. 01 June 2022 [cit. 29. 07. 2022]. Dostupné z: https://semaphoreci.com/blog/test-automation

[9] BĚLOCH, Tomáš. Automatizované testování webových aplikací [online]. Brno, 2013 [cit. 29. 07. 2022]. Bakalářská práce. Vysoké učení technické, Fakulta informačních technologií. Dostupné z: https://dspace.vutbr.cz/bitstream/handle/11012/187471/final-thesis.pdf

[10] BACKER, Afsal. Playwright vs. Selenium: Key Differences: Muuk Test [online]. [cit. 24. 01. 2023]. Dostupné z: https://muuktest.com/blog/playwright-vs-selenium/

[11] Screenster [online]. [cit. 25. 01. 2023]. Dostupné z: https://www.screenster.io/

[12] Cypress: Documentation [online]. [cit. 27. 01. 2023]. Dostupné z: https://docs.cypress.io/

[13] MWAURA, Waweru. End-to-end Web Testing with Cypress. Birmingham: Packt Publishing, 2021. ISBN 978-1-83921-385-4.

[14] UNADKAT Jash. Cypress vs Selenium: Key Differences: BrowserStack [online]. 10 June 2022 [cit. 25. 01. 2023]. Dostupné z: https://www.browserstack.com/guide/cypress-vs-selenium

[15] Onboardee [online]. [cit. 24. 01. 2023]. Dostupné z: https://onboardee.io/

[16] VEBROVÁ, J. Škálovatelnost [online]. [cit. 13. 04. 2023]. Dostupné z: https://wiki.knihovna.cz/index.php/Škálovatelnost

[17] Monti APM [online]. [cit. 13. 04. 2023]. Dostupné z: https://montiapm.com/

11 Seznam použitých obrázků

11.1 Seznam obrázků

Obrázek 1: Znázornění rozdílu mezi přesností a správností na šipkách [1] (upraveno autorkou práce) 2

Obrázek 2: Každý software má vlastní optimální testování [1] 8

Obrázek 3: Selenium versus Cypress z hlediska architektury provádění testů [13] (upraveno autorkou práce) 16

Obrázek 4: Ukázka výstupu testu vycházejícího z předchozího kódu [12] (upraveno autorkou práce) 21

Obrázek 5: Ukázka výstupu vycházející ze Zdrojového kódu 11 (autorka práce) 22

Obrázek 6: Příklad chybové hlášky spadlého testu [12] (upraveno autorkou práce) 24

Obrázek 7: Ukázka dostupných kariérních milníků (autorka práce) 25

Obrázek 8: Ukázka kariéry uchazeče (autorka práce) 25

Obrázek 9: Rozdíl mezi administrátorským a zaměstnaneckým účtem (autorka práce) 29

Obrázek 10: Ukázka šablony (autorka práce) 31

Obrázek 11: Struktura projektu (autorka práce) 32

Obrázek 12: Struktura e2e složky s testy (autorka práce) 33

Obrázek 13: Výstup ze Zdrojového kódu 16 (autorka práce) 34

Obrázek 14: Ukázka úspěšného projití testu TS_A_04 (autorka práce) 37

Obrázek 15: Výsledky vytvořené testovací sady (autorka práce) 42

11.2 Seznam zdrojových kódů

Zdrojový kód 1: Přesun do složku projektu [12] ... 17

Zdrojový kód 2: Instalace Cypressu pomocí npm [12] ... 17

Zdrojový kód 3: Instalace Cypressu pomocí yarn [12] ... 17

Zdrojový kód 4: Ukázka použití příkazu cy.visit() [12] ... 18

Zdrojový kód 5: Ukázka použití příkazu cy.url() [12] ... 18

Zdrojový kód 6: Ukázka použití příkazu cy.get() [12] ... 19

Zdrojový kód 7: Ukázka použití příkazu .should() [12] ... 19

Zdrojový kód 8: Ukázka použití příkazu .click() [12] ... 19

Zdrojový kód 9: Ukázka použití příkazu .type() [12] ... 20

Zdrojový kód 10: Ukázka struktury testu [12] ... 20

Zdrojový kód 11: Ukázka testu pro přihlašování do IS/STAGu (autorka práce) ... 22

Zdrojový kód 12: Spuštění testů přímo v konzoli [12] ... 23

Zdrojový kód 13: Otevření Cypress pomocí npx [12] ... 23

Zdrojový kód 14: Otevření Cypress pomocí yarn [12] ... 23

Zdrojový kód 15: Mazání předchozí session a přihlašování řešené v e2e.js (autorka práce) ... 33

Zdrojový kód 16: Ukázka nastavení konfigurace (autorka práce) ... 34

Zdrojový kód 17: Nastavení pracovního prostoru pro TS_A_04 (autorka práce) ... 35

Zdrojový kód 18: Testovací případy pro testování šablon (autorka práce) ... 35

Zdrojový kód 19: Ukázka pomocné funkce createTemplate() (autorka práce) ... 36

Zdrojový kód 20: Ukázka úklidu na konci testu (autorka práce) ... 36

Zdrojový kód 21: Ukázka vytvoření vlastního Cypress příkazu pro selectOption() (autorka práce) ... 38

Zdrojový kód 22: Ukázka vytvoření vlastního Cypress příkazu pro delete() (autorka práce) 38

Zdrojový kód 23: Ukázka funkce renameItem() (autorka práce) 39

Zdrojový kód 24: Spuštění sady vytvořených testů v konzoli (autorka práce) 39

Zdrojový kód 25: Spuštění konkrétního vytvořeného testu (autorka práce) 39

Zdrojový kód 26: Spuštění uživatelského prostředí Cypressu (autorka práce) 40

Zdrojový kód 27: Ukázka použití příkazu .session() [12] 40

Zdrojový kód 28: Zjednodušená ukázka použitého cy.session() (autorka práce) .. 41

Zdrojový kód 29: Ukázka z funkce deleteModule(milestoneName) (autorka práce) 41

11.3 Seznam tabulek

Tabulka 1: Seznam testovacích scénářů (autorka práce) 30

Tabulka 2: Seznam testovacích případů v TS_A_04_templates (autorka práce) 31

12 Přílohy

1) CD/DVD obsahující vytvořené testy pro webovou aplikaci Onboardee.io.

   Pozn. Z právních důvodů, nejsou dodané testy kompletní. Nelze je tedy spustit bez vytvořeného testovacího prostředí aplikace a přihlašovacích údajů.

Oskenované zadání práce

UNIVERZITA HRADEC KRÁLOVÉ
Fakulta informatiky a managementu
Akademický rok: 2021/2022

Studijní program: Aplikovaná informatika
Forma studia: Prezenční
Obor/kombinace: Aplikovaná informatika (ai3-p)

Podklad pro zadání BAKALÁŘSKÉ práce studenta

Jméno a příjmení:	**Hai Yen Pham**
Osobní číslo:	**I2000450**
Adresa:	Stržanov 85, Žďár nad Sázavou – Stržanov, 59102 Žďár nad Sázavou 2, Česká republika
Téma práce:	Automatizované testování webové aplikace Onboardee.io
Téma práce anglicky:	Automated testing of the web application Onboardee.io
Jazyk práce:	Čeština
Vedoucí práce:	Ing. Martina Husáková, Ph.D. Katedra informačních technologií

Zásady pro vypracování:

Cíl: Cílem bakalářské práce je návrh automatizovaných testů pro nově vyvíjenou webovou aplikaci Onboardee.io.
Teoretická část práce přináší obecný vhled do problematiky testování, následně se zaměřuje hlouběji na automatizované testování a představuje vybrané nástroje, které lze k tomuto využít.
Praktická část je zaměřena na samotnou aplikaci Onboardee.io a její požadavky na testování, která vede k výběru testovacího frameworku a následnému návrhu a implementaci automatizovaných testů.
Osnova práce:
| Úvod
| Testování softwaru
: Kategorie testování
: Automatizované testování
| Cypress
| Webová aplikace Onboardee.io
| Návrh a implementace automatizovaných testů pro aplikaci
| Závěr

Seznam doporučené literatury:

1. BUREŠ, Miroslav, Miroslav RENDA, Michal DOLEŽEL, Peter SVOBODA, Zdeněk GRÖSSL, Martin KOMÁREK, Ondřej MACEK a Radoslav MLYNÁŘ. *Efektivní testování softwaru: klíčové otázky pro efektivitu testovacího procesu.* Praha: Grada, 2016. Profesionál. ISBN 978-80-247-5594-6.
2. KHAN, Mohd Ehmer and KHAN, Farmeena. A comparative study of white box, black box and grey box testing techniques. International Journal of Advanced Computer Science and Applications. 2012. ISSN 2156-5570.
3. UMAR, Mubarak Albarka and CHEN Zhanfang. *A Study of Automated Software Testing: Automation Tools and Frameworks.* International Journal of Computer Science Engineering. 2019. ISSN 2319-7323.

Podpis studenta: Datum:

Podpis vedoucího práce: Datum:

© IS/STAG, Portál – Podklad kvalifikační práce , phamha1, 11. dubna 2023 11:17