Česká zemědělská univerzita v Praze

Provozně ekonomická fakulta

Katedra informačních technologií

Diplomová práce

Automatické testování

Vasily Gusev

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Provozně ekonomická fakulta

ZADÁNÍ DIPLOMOVÉ PRÁCE

Bc. Vasily Gusev

Informatika

Název práce

Automatické testování

Název anglicky

Automatic testing

Cíle práce

Diplomová práce je tematicky zaměřena na různé přístupy k automatickému testování. Hlavním cílem práce je analýza možností existujících metod a technologii automatického testování. Dílčí cíle práce jsou:
- vypracovaní přehledu metod automatického testování
- vypracování přehledu vývoje strategií a metodologií automatického testování.

Metodika

Metodika řešené problematiky diplomové práce je založena na studiu a analýze odborných informačních zdrojů. Vlastní práce spočívá v analýze možností existujících metod a technologii automatického testování s vypracováním návrhu a provedení experimentů pomocí existujících softwarových řešení pro automatizaci testů na zvoleném příkladu. Pro vývoj strategií a metodologií automatického testování budou použité existující rekomendace ISTQB (International Software Testing Qualifications Board). Na základě syntézy teoretických poznatků a výsledků praktické části budou formulovány závěry diplomové práce.

Doporučený rozsah práce

60 – 80 stran textu.

Klíčová slova

Automatické testování, Testovací scénáře, Testovací strategie, Přístupy k automatizaci testování, Metody testování, ISTQB

Doporučené zdroje informací

BAKKER, Bryan, Graham BATH, Mark FEWSTER, Armin BORN, Judy MCKAY, Andrew POLLNER, Raluca POPESCU a Ina SCHIEFERDECKER. Certified Tester Advanced Level Syllabus. International Software Testing Qualifications Board [online]. 2016, 2016 [cit. 2023-06-07]. Dostupné z: https://istqb-main-web-prod.s3.amazonaws.com/media/documents/ISTQB-CT-TAE_Syllabus_v1.0_2016.pdf

BUREŠ, Miroslav, Miroslav RENDA, Michal DOLEŽEL, Peter SVOBODA, Zdeněk GRÖSSL, Martin KOMÁREK, Ondřej MACEK a Radoslav MLYNÁŘ. Efektivní testování softwaru. Grada, 2016. ISBN 9788024755946.

CODESIDO, Ivan. What is front-end development?. The Guardian [online]. 2009, 2023 [cit. 2023-06-07]. Dostupné z: https://www.theguardian.com/help/insideguardian/2009/sep/28/blogpost

KOĎOUSKOVÁ, Barbora. CO JE TO API A JAKÉ JSOU MOŽNOSTI JEHO VYUŽITÍ?. Rascasone [online]. 2023, 2023 [cit. 2023-06-07]. Dostupné z: https://www.theguardian.com/help/insideguardian/2009/sep/28/blogpost

ZAPTEST. Co je automatizace testování? Jednoduchý průvodce bez žargonu. Zaptest [online]. 2023, 2023 [cit. 2023-06-07]. Dostupné z: https://www.zaptest.com/cs/co-je-automatizace-testovani-jednoduchy-pruvodce-bez-zargonu

---

Předběžný termín obhajoby

2023/24 LS – PEF

Vedoucí práce

doc. Ing. Pavel Šimek, Ph.D.

Garantující pracoviště

Katedra informačních technologií

Elektronicky schváleno dne 4. 7. 2023

doc. Ing. Jiří Vaněk, Ph.D.

Vedoucí katedry

Elektronicky schváleno dne 3. 11. 2023

doc. Ing. Tomáš Šubrt, Ph.D.

Děkan

V Praze dne 30. 03. 2024

Čestné prohlášení

Prohlašuji, že svou diplomovou práci „Automatické testování“ jsem vypracoval samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou citovány v práci a uvedeny v seznamu použitých zdrojů na konci práce. Jako autor uvedené diplomové práce dále prohlašuji, že jsem v souvislosti s jejím vytvořením neporušil autorská práva třetích osob.

V Praze dne 30. 3. 2024 _____

Poděkování

Chtěl bych poděkovat vedoucímu bakalářské práce panu Ing. Pavlu Šimkovi, Ph.D., za odborné vedení práce, trpělivost, ochotu pomoci a cenné rady, které mi pomohly tuto práci zkompletovat.

Automatické testování

Abstrakt

Diplomová práce se zabývá problematikou automatického testování. Jejím cílem je analýza možností existujících metod a technologií automatického testování. Technologie automatického testování jsou frameworky nebo software, které se používají k automatizaci testovacích scénářů. V závislosti na jejich účelu je jejich pomocí testován front-end, back-end nebo API.

V diplomové práci je provedena analýza a studie informačních zdrojů tykajících se testování softwaru. Jsou definovány pojmy testování jako proces a jeho cíle, strategie, přístup a plán. Byly rozlišeny termíny „selhání“, „chyba“, „defekt“ a „incident“. Následně byly definovány termíny SDLC a STLC. Byly popsány modely V a W. Rovněž bylo rozepsáno použití jednotlivých typů prostředí. Byl vytvořen přehled metod a technik testování. Dále proběhla definice front-endu, back-endu a API. Závěrem byla zformulována doporučení ISTQB.

Pomocí analýzy klíčových slov byly vybrány softwary a frameworky pro následné provedení experimentu. U každého experimentu byl definován cíl a metodika. Frameworky a softwary byly rozděleny podle jejich účelu a schopností otestovat komponentu front-end, API a back-end. Pro každou komponentu byly na základě cíle a metodiky experimentu vytvořeny testovací scénáře, podle kterých byla provedena exekuce jednotlivými nástroji. Na základě výsledků dosažených jednotlivými experimenty byl vypracován přehled vybraných metod a technologií automatického testování.

Klíčová slova: Automatické testování, Testovací scénáře, Testovací strategie, Přístupy k automatizaci testování, Metody testování, ISTQB.

Automatic Testing

Abstract

This thesis addresses the issue of automated testing. Its goal is to analyze the possibilities of existing methods and technologies of automated testing. Automated testing technologies are frameworks or software used to automate testing scenarios. Depending on their purpose, they are used to test the front-end, back-end or API.

The thesis conducts an analysis and study of information sources related to software testing. It defines concepts of testing as a process and its goals, strategies, approaches, and plan. The terms "failure", "error", "defect", and "incident" are distinguished. Subsequently, the terms SDLC and STLC were defined. The V and W models were described. The use of various types of environments was also elaborated. An overview of testing methods and techniques was created. Furthermore, the definitions of front-end, back-end, and API were provided. In conclusion, ISTQB recommendations were formulated.

Through keyword analysis, software and frameworks were selected for subsequent experimentation. Each experiment defined a goal and methodology. Frameworks and software were divided according to their purpose and capabilities to test the front-end, API, and back-end components. For each component, test scenarios were created based on the goal and methodology of the experiment, according to which the execution was carried out by individual tools. Based on the results achieved by individual experiments, an overview of selected methods and technologies of automated testing was compiled.

Keywords: Automated Testing, Testing Scenarios, Testing Strategy, Approaches to Test Automation, Testing Methods, ISTQB.

Obsah

1 Úvod

V dnešní době se ve světě softwarového vývoje používají neměnné fáze, bez kterých už se tento vývoj nepředpokládá. První a poměrně důležitou fází je analýza stanoveného problému. Potom následuje sběr požadavků, plánování a návrh softwarového řešení, do kterého také patří analýza a design. Poté následuje samotný vývoj softwarového řešení a spolu s tím začíná testování a ladění. Po úspěšném vývoji se softwarový systém nasazuje. Dále začíná údržba softwaru, do které patří jak oprava chyb, tak i přidání dalších funkcí.

Testování je jednou z nejdůležitějších fází v procesu vývoje softwarového řešení, protože pomáhá identifikovat chyby a nedostatky ještě před nasazením do produkčního prostředí. S postupem doby a nárůstem rozsahu testovacích scénářů se tradiční manuální testování stává náročným a časově nákladným úkonem. Proto je lepší pro nějaké testovací scénáře použít automatické testování. Tímto způsobem je možné zvýšit účinnost testování, snížit náklady a zkrátit časovou náročnost testovacího cyklu.

V závislosti na tom, co je zapotřebí otestovat, je zvolen i software pro testování. Existuje sada různých aplikací, které pomáhají v testování front-endu nebo back-endu. U front-endu se většinou ověřuje například funkčnost, responzivita, grafický design, zobrazení. Když se testuje back-end, tak se zaměřuje na ověření operací a procesů, které probíhají na straně serveru. Jako například funkčnost API, správnost logiky, bezpečnost, výkonnost.

Autor diplomové práce si klade za cíl analýzu možností existujících metod a technologií automatického testování. Dílčími cíli jsou vypracování přehledu metod automatického testování a vypracování přehledu vývoje strategií a metodologií automatického testování.

2 Cíl práce a metodika

2.1 Cíl práce

Diplomová práce je tematicky zaměřena na různé přístupy k automatickému testování. Hlavním cílem práce je analýza možností existujících metod a technologií automatického testování. Dílčí cíle práce jsou:

- vypracování přehledu metod automatického testování;
- vypracování přehledu vývoje strategií a metodologií automatického testování.

2.2 Metodika

Metodika řešené problematiky diplomové práce je založena na studiu a analýze odborných informačních zdrojů. Vlastní práce spočívá v analýze možností existujících metod a technologií automatického testování s vypracováním návrhu a provedení experimentů pomocí existujících softwarových řešení pro automatizaci testů na zvoleném příkladu. Volba technologií automatického testování proběhne pomocí analýzy trendu a klíčových slov. Pro vývoj strategií a metodologií automatického testování budou použité existující rekomendace ISTQB (International Software Testing Qualifications Board). Na základě syntézy teoretických poznatků a výsledků praktické části budou formulovány závěry diplomové práce.

3 Teoretická východiska

3.1 Testování softwaru

Tato kapitola se bude zabývat definicí testování a jeho klíčovými aspekty. Testování představuje nezbytný proces v softwarovém vývoji, který umožňuje ověřit kvalitu a funkčnost softwaru před jeho nasazením. Bude zde rozebráno, co zahrnuje samotná testovací činnost, jaké jsou cíle testování a způsoby.

3.1.1 Definice testování

Testování je proces, který slouží k ověření správného fungování a kvality softwarového produktu. Pomocí testování dochází odhalení chyb, nedostatků a nepřesností ve všech částech softwaru před jeho uvedením do provozu. Během testování se provádějí různé typy testů, aby se zjistilo, zda software pracuje podle očekávání, splňuje stanovené požadavky a je stabilní i za různých podmínek. Zajišťuje spolehlivost a funkčnost softwaru, a tím minimalizuje riziko chyb, které by mohly negativně ovlivnit zkušenost uživatele, čímž potom sníží celkovou kvalitu produktu. Rozlišuje se manuální a automatické testování. (7)

3.1.2 Chyba

Chyba je nedostatek způsobený lidským faktorem, nesrovnalost nebo nepřesnost v softwarovém kódu. Chyba může být způsobena nepozorností osoby, chybou v návrhu, chybou vývojáře anebo chybou v kódu softwaru. Vzniknutí takové chyby může způsobit vznik defektu nebo takzvaného Bugu. (2 str. 149)

3.1.3 Defekt

Defekt, anglicky označovaný jako *Bug*, je konkrétním případem chyby v softwarovém produktu. Defekt je odchylka od specifikace nebo očekávání, která může způsobit nesprávné chování softwaru. Defekty se vyskytují během testování a jsou reportovány vývojovému týmu. Jsou také detekovány v případě, když je software uveden do produkčního provozu. Defekty z produkčního prostředí, jsou ve většině případů reportovány uživateli softwaru. (2 str. 149)

3.1.4 Selhání

Selhání, angl. *Failure,* je případ, kdy software funguje nesprávně ve skutečném provozním prostředí. Selhání je ve většině případů způsobeno defektem a může vést k havárii buď celého softwaru, anebo jenom jeho části. (2 str. 149)

3.1.5 Incident

Incident je širší termín (z definice ISTQB), který zahrnuje jakýkoli problém spojený se softwarem v produkci. Tento problém vyžaduje větší pozornost, investigaci a následné řešení. Incident může být způsoben defekty i selhání. A vývojový tým by měl provést investigaci problému, řešení a zajištění, aby k podobným problémům nedocházelo v budoucnu. Incidenty jsou vždy více sledování. (2 str. 149)

3.1.6 Cíle testování

Cíle testování představují základní směrnice, jež testovací tým sleduje během provádění testování softwaru nebo systému. Tato sada cílů slouží ke zlepšení kvality, spolehlivosti a funkčnosti konečného produktu. Zde jsou uvedeny některé z hlavních:

- **ověření funkčnosti:** ověření, zda se software provozuje podle očekávání a naplňuje stanovené funkční požadavky;
- **ověření kvality:** cílem je zajistit, že software splňuje stanovené kvalitativní standardy;
- **minimalizace rizika:** minimalizace výskytů takových rizik, jako jsou výpadky, chyby, chybné chování nebo ztráta důležitých dat uživatelů;
- **odhalení chyb:** identifikování nedostatků, chyb a defektů testovaného softwaru proto, aby bylo možné tyto nedostatky opravit a zajistit kvalitu výsledného produktu;
- **ověření použitelnosti**: ověření, že software je intuitivní a snadno použitelný pro koncového uživatele;
- **ověření výkonu:** hodnocení výkonnostních charakteristik softwaru, jako jsou rychlost odezvy, chování za ztížených podmínek a kapacitní parametry;
- **zajištění spolehlivosti:** ověření, že softwaru je uživatelům k dispozici bez přerušení a že funguje bez nějakých závažnějších vad. (2 str. 44–49, 7)

Cíle testování se mohou lišit v závislosti na konkrétním projektu, jeho charakteristikách, prioritách a produktových požadavcích. Nicméně obecně jsou cíle testování zaměřeny na to, aby byl zajištěn chod spolehlivého, kvalitního a funkčního softwaru, který uspokojuje potřeby uživatelů a cíle produktu. (2 str. 44–45)

3.1.7 Kontrola kvality

Kontrola kvality (angl. Quality Control) je proces monitorování, zajišťování a kontroly kvality vyvíjeného produktu. Tato kontrola prověřuje, že testovaný systém na výstupu splňuje stanovené podmínky a požadavky. (8)

3.1.8 Zajištění kvality

Zajištění kvality (angl. Quality Insurance) je širší pojem než kontrola kvality. Zajištění kvality je takový proces, který vstupuje od počáteční fáze vývoje produktu až do ukončení. Quality Insurance se zaměřuje na prevenci defektů a nedostatků softwaru a to tak, že se soustředí na procesy a metodologie, které zajišťují splnění standardů a kritérií kvality. (2 str. 65, 8)

3.1.9 Manuální testování

Manuální testování je způsob testování softwaru, který probíhá pomocí lidské interakce se softwarem. Osoba, která provádí testy, se jmenuje tester. Tester je zodpovědný za výběr testovací strategie, provedení testovací analýzy, tvorbu a spouštění testovacích scénářů a tvorbu reportů na základě provedených testů. Manuální testování zahrnuje fyzickou interakci s aplikací nebo softwarem. Provádí se kontrola, jestli se software chová podle očekávání a dokumentace. (6)

Při manuálním testování softwaru tester kliká na tlačítka, zadává data do určitých polí nebo formulářů, ověřuje správnou funkčnost testovaného systému a provádí kontrolu vzhledu podle stanoveného designu. (6)

3.1.10 Automatické testování

Automatické testování je způsob testování softwaru, kdy se testy na software nebo systém provádějí bez lidské interakce pomocí specializovaných nástrojů a skriptů. Tester připravuje testovací scénáře v nějakém nástroje a potom je spouští. (15)

Pomocí automatického testování je umožněna opakovatelnost testovacích scénářů a testů s minimálním lidským zásahem. Na rozdíl od manuálního testování, tento druh testování nevyžaduje velké úsilí testera. Potíž je v tom, že tester musí provést prvotní nastavení nástroje pro automatizační testy, potom tester musí vymyslet a napsat testovací skripty. (15)

Automatické testy umožňují častější kontrolu softwaru během jeho vývoje. Automaty se často používají pro regresní testování. Jsou také výhodné pro výkonnostní testy, pro testování různých scénářů nebo pro pokrytí kódu. (15)

Celkově vzato, automatické testování je vhodné pro dlouhodobé projekty, u kterých se dá využít opakované testování a výkonnostní testy. Nicméně inicializační a údržbové náklady by měly být pečlivě zváženy. (15)

3.1.11 Shrnutí

V této kapitole byl definován pojem testování. K tomuto pojmu byly sepsány jednotlivé příklady cílů. Pojmy kontrola kvality a zajištění kvality se často pletou, a proto byl vysvětlen jejich rozdíl. Aktuálně existují dva způsoby testování: manuální a automatický typ. Každý z těchto typů má své výhody a nevýhody, které byly také rozepsány. Byly definovány pojmy defekt, selhání a incident a byl vysvětlen rozdíl mezi těmito pojmy.

3.2 Proces testování

Proces testování je klíčová aktivita zaměřená na identifikaci a odstranění chyb, problémů nebo nedostatků v softwarovém produktu. Kapitola detailně rozebírá jednotlivé fáze a kroky, které tým provádí během testovacího procesu. Dále jsou definovány pojmy SDLC (Software Development Life Cycle) a STLC (Software Testing Life Cycle), popsány jsou způsoby, jak se každá fáze přesně provádí a jakým způsobem jednotlivé fáze navzájem souvisejí. Dále jsou popsány V-model a W-model a jejích rozdíl. Definovány jsou také termíny validace a verifikace.

3.2.1 Politika testování

Politika testování (angl. Test Policy) je dokument na vysoké úrovni, který stanovuje základní rámec a usměrňuje celkové principy a přístupy k testování softwaru v organizaci. (2 str. 40)

3.2.2 Strategie testování

Strategie testování (angl. Test Strategy) je popis, který definuje přístup k testování softwarového projektu. (2 str. 40)

3.2.3 Přístup k testování

Přístup k testování (angl. Test Approach) je část testovací strategie, která definuje metodologii a taktiku, které budou použity pro dosažení cílů testování. Tento přístup popisuje, jakým způsobem budou testovací aktivity organizovány a provedeny. (2 str. 40)

3.2.4 Plán testování

Plán testování (angl. Test Plan) je dokument popisující, co má testování softwarového projektu dosáhnout, jak budou testovací aktivity provedeny, jaké zdroje budou potřebné a jak budou řešena potenciální rizika. Tento plán vymezuje rámec pro testovací procesy. Tento dokument také slouží jako základní bod pro koordinaci a komunikaci mezi všemi zúčastněnými stranami. (2 str. 40)

3.2.5 Vstupní kritéria

Vstupní kritéria (angl. Entry Criteria) je sada podmínek a požadavků, které musí být splněny před zahájením jakékoli fáze testování. Tato kritéria zajišťují, že testování se začne pouze tehdy, když jsou všechny nezbytné prvky připravené a schválené. (2 str. 55)

3.2.6 Výstupní kritéria

Výstupní kritéria jsou sadou podmínek, které musí být splněny, aby bylo testování považováno za ukončené. Pomáhají testerům, manažerům a zainteresovaným stranám určit, zda bylo testování úspěšné a zda je produkt připraven k dalším krokům, jako jsou nasazení nebo uvedení na trh. (2 str. 55)

3.2.7 SDLC (Software Development Life Cycle)

SDLC (Software Development Life Cycle) je metodika, která popisuje proces vývoje softwaru od jeho konceptuální fáze až po jeho dokončení a nasazení. Jedná se o systematický přístup, který organizace používají k řízení a správě vývoje softwaru s cílem zajistit kvalitu, efektivitu a včasné dokončení projektu. (3)

SDLC zahrnuje několik fází, které jsou provedeny postupně a vzájemně se prolínají. Základními cíli SDLC jsou:

- **Sběr požadavků a jejích analýza (Requirements Gathering and Analysis)**

V této fázi probíhá sběr a analýza požadavků. Vývojový tým získává a analyzuje potřeby a očekávání koncového zákazníka. Pomocí analýzy se získává jasná a podrobná představa

o tom, co má software dělat a jaké problémy má řešit. Dále se vytvoří specifikace, která bude vodítkem pro následující fázi vývoje. (3)

- **Návrh (Design)**

V této fázi se na základě specifikovaných požadavků vytváří návrh architektury systému. Definují se jednotlivé komponenty, jak budou spolupracovat, jakým způsobem budou data ukládána a jak bude probíhat celková struktura systému. Dále se navrhují uživatelská rozhraní a interakce, tak aby byla zajištěna co nejlepší uživatelská přívětivost a efektivita. Tato fáze často zahrnuje tvorbu diagramů a technických dokumentů. (3)

- **Implementace (Implementation)**

Po dokončení návrhu přichází na řadu fáze implementace, kde se návrh proměňuje v reálný zdrojový kód. Programátoři píší kód podle definovaných specifikací a návrhů. Kód musí být dobře strukturovaný, čitelný a komentovaný, aby byl srozumitelný pro další členy týmu. Během implementace se také vytvářejí testovací scénáře, které budou následně použity k ověření funkčnosti softwaru. (3)

- **Testování (Testing)**

Fáze testování zahrnuje ověření, zda software splňuje stanovené požadavky a zda funguje bez chyb. Testování je prováděno na různých úrovních začínajících od jednotkových testů, kde se testují jednotlivé komponenty, přes integrační testy, které zkoumají interakce mezi komponentami, až po systémové testy, kde se celý systém testuje jako celek. Testování odhaluje chyby a nedostatky, které jsou následně opraveny vývojovým týmem. (3)

- **Nasazení (Deployment)**

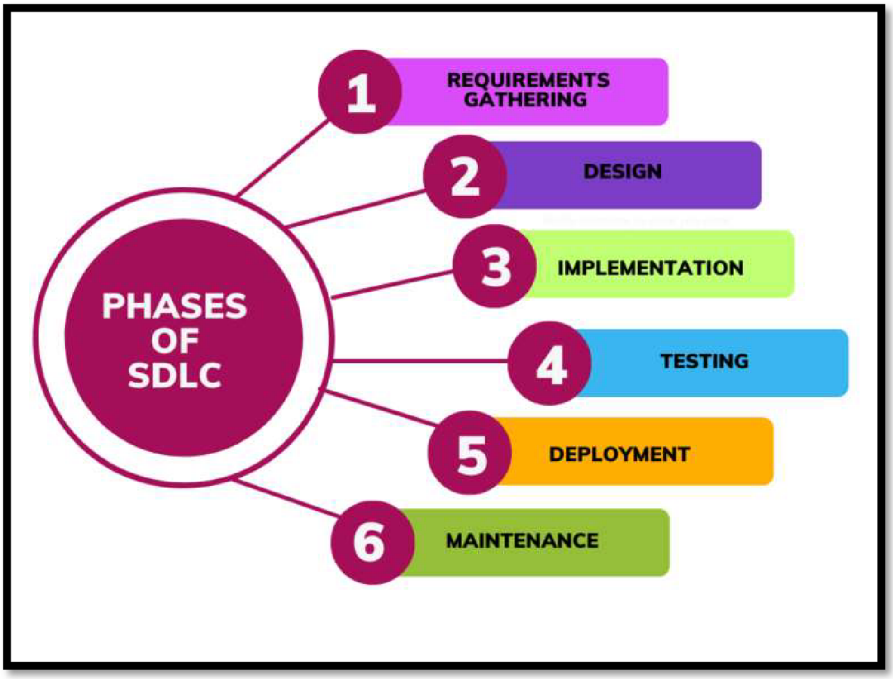
Po úspěšném testování a odstranění nalezených chyb je software připraven k nasazení. Nasazení může zahrnovat instalaci softwaru do produkčního prostředí pro klienty. Je důležité zajistit, aby nasazení proběhlo bez negativního vlivu na uživatele. (3)

- **Podpora / Údržba (Maintenance)**

V této poslední fázi vývojový tým začíná s podporou softwaru, který dostal k produkci. Během této fáze reální uživatelé začínají používat aplikaci a je velká pravděpodobnost, že budou nalezené nové chyby. Tyto chyby se hlásí, testují a následně se opravují.

SDLC může existovat v různých variantách, jako jsou Waterfall, Agile, Scrum a další, každá s vlastním zaměřením na postupy a hodnoty. (3)

Obrázek 1 SDLC



Zdroj: (3)

3.2.8 STLC (Software Testing Life Cycle)

STLC (angl. Software Testing Life Cycle) je metodika, která popisuje fáze a kroky, které se provádějí během testování softwarového produktu. STLC je součástí celkového SDLC. Zajišťuje, že software je pečlivě testován, aby se odhalily chyby a nedostatky před jeho nasazením. (3)

Fáze, které patří do STLC:

1. **Analýza požadavků (Requirements Analysis)**

V této fázi se testovací tým detailně seznámí s požadavky na software. Analyzují se funkční a nefunkční požadavky, a to včetně přesného porozumění očekáváním klienta a uživatelů. Účelem je získat obraz o tom, co software musí splňovat. Na základě této analýzy se následně definují testovací scénáře a testovací strategie. (12)

2. **Plánování testování (Test Planning)**

Následuje fáze plánování testovacích aktivit, ve které se stanovují cíle testování, identifikují se testovací zdroje (lidské zdroje a softwarové prostředky), a vybírají se testovací techniky a nástroje. Spolu s tím probíhá definice rozsahu testování a kritérii pro jeho ukončení. Dalším aspektem je vytvoření plánu testování, který zahrnuje časové plánování, alokaci zdrojů, identifikaci rizik a definování metrik pro měření pokroku a úspěšnosti testování. (3, 12)

3. **Návrh testů (Test Design)**

V této fázi se na základě stanovených požadavků a plánování vytváří konkrétní testovací scénáře. Testovací scénáře popisují postup, který tester bude následovat, a očekávané výsledky pro každý ze scénářů. Důraz je kladen na to, aby testovací scénáře pokrývaly všechny aspekty softwaru a byly co nejkomplexnější. Tato fáze zahrnuje i přípravu testovacích dat a definování testovacího prostředí. (3, 12)

4. **Exekuce testů (Test Execution)**

V této fázi začíná samotné provedení testů podle definovaných scénářů. Testy mohou být prováděny manuálně nebo automatizovaně podle potřeby. Tester nebo testovací skript sleduje, zda je očekávaných výsledků dosaženo. Výsledky testů jsou zaznamenávány, včetně zjištěných chyb a nesrovnalostí. Tyto záznamy jsou pak použity pro další analýzu, aby byla přesně pochopena příčina chyb. (12)

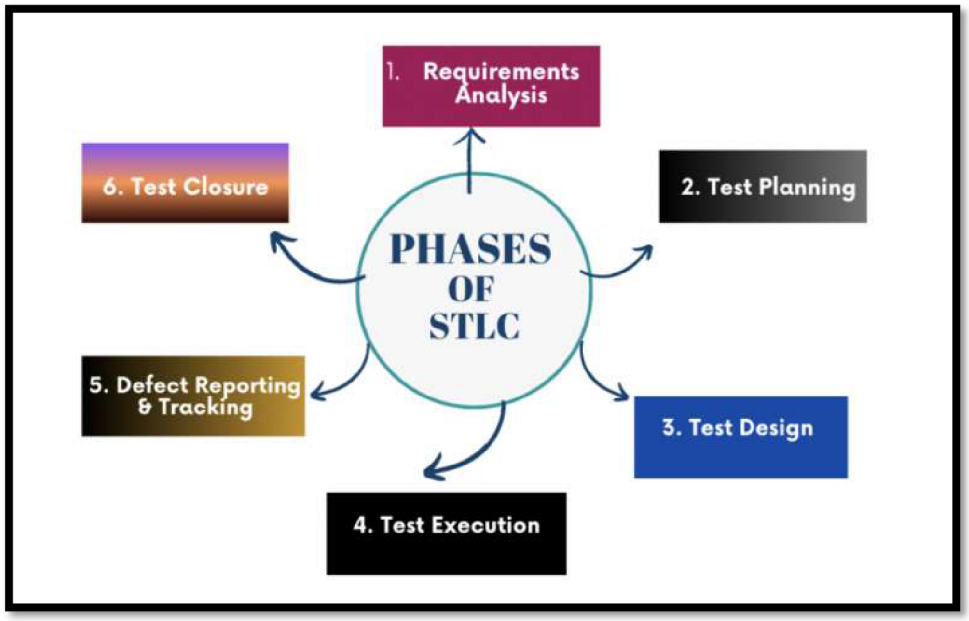
5. **Reportování a sledování chyb (Defect Reporting & Tracking)**

Pokud jsou během testování identifikovány chyby, tyto chyby jsou reportovány vývojovému týmu. Každá chyba je dokumentována. Vytváří se její popis, přiděluje se kategorie závažnosti a popisuje se očekáváné chování podle dokumentace. Vývojový tým se mezi tím zabývá sledováním stavu chyb a zajišťuje, aby byly opravy provedeny a následně došlo k uzavření chyb. (12)

6. **Uzavření testování (Test Closure)**

Poslední fází je uzavření testování, u kterého tým provádí celkovou analýzu výsledků testování a zpracuje závěrečné shrnutí. Shrnutí obsahuje detaily o provedených testech, nalezených chybách, pokrytí testování a dalších důležitých informacích. Na základě tohoto shrnutí se rozhoduje o tom, zda je software připraven k nasazení. (3)

Obrázek 2 STLC

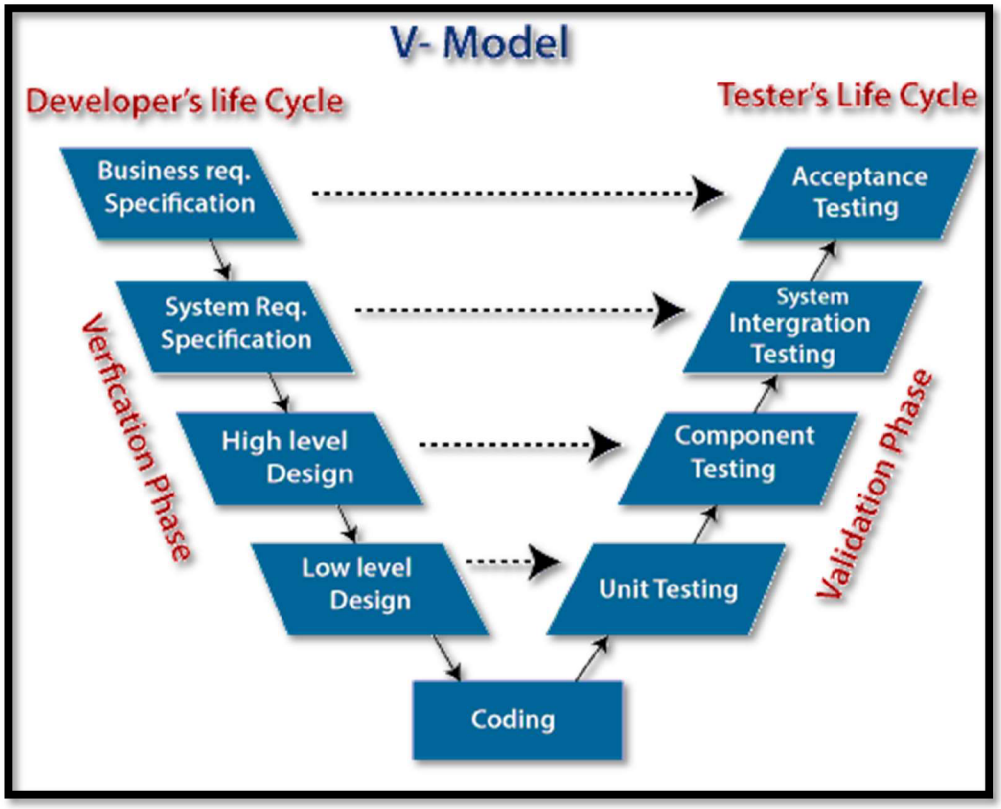


Zdroj: (3)

3.2.9 V-Model

V-model (také nazývaný Validation a Verification Model) je model vývoje softwaru, který je vylepšenou verzí tradičního vodopádového (Waterfall) modelu. Jedná se o jednoduchý model, který obsahuje lineárně sekvenciální životní cyklus, zdůrazňující proces „validace“ a „verifikaci“ v různých fázích vývoje. V tomto modelu se předpokládá, že každá fáze vývoje na levé straně „V“ má odpovídající fázi testování na pravé straně „V“. Jako například detailní návrh – integrační testy (2 str. 28, 13)

Obrázek 3 V-Model



Zdroj: (13)

Fáze na levé straně „V“:

- **Specifikace požadavků, implicitní očekávání**: prvním krokem je sběr, identifikace a zdokumentování všech funkčních a nefunkčních požadavků na software. Proces zahrnuje komunikaci s různými zúčastněnými stranami, jako jsou například zákazníci;
- **Hrubý návrh (High-Level Design):** během této fáze se vytváří hrubý návrh, nebo tzv. architektonický design. Určují se interakce mezi moduly a jejich vzájemné závislosti;
- **Detailní návrh (Low-Level Design):** architektonický design se v této fázi rozkládá na menší moduly. Každý modul nebo komponenty jsou nyní detailně navrženy. To zahrnuje návrh algoritmů, datových struktur, a dokonce i pseudokódu;
- **Programování:** v této fáze začíná kódování na základě návrhů nižší úrovně. Kódování je prováděno v programovacích jazycích a technologiích, které jsou vhodné pro vyvíjený produkt. (2 str. 28–29, 13)

Fáze na pravé straně „V“:

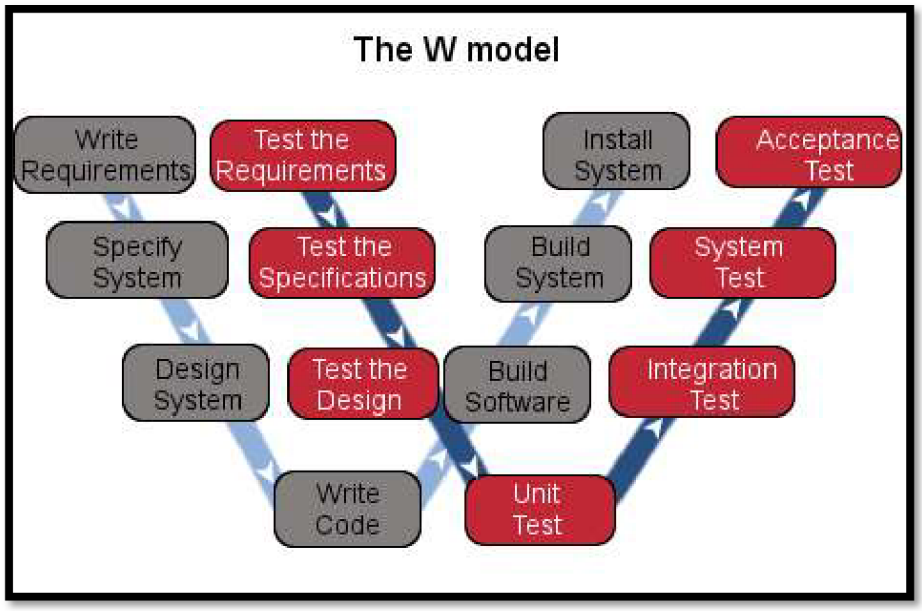
- **Jednotkové testy:** v této fázi se testují jednotlivé moduly a komponenty softwaru na úrovní kódu. Zjišťuje se, zda každá jednotka funguje podle specifikací a návrhu. Testování je provedeno vývojářem;
- **Integrační testy**: po jednotkovém testování následuje testování interakcí mezi jednotlivými moduly nebo systémy. Probíhá ověření, zda spolupracují správně a zda celý subsystém funguje podle očekávání;
- **Systémové testy**: systémovým testováním se myslí testování celého softwarového systému. Provádějí se testy všech funkčních a nefunkčních požadavků. Tyto testy probíhají na simulovaném prostředí, které co nejvíce odpovídá reálnému;
- **Uživatelské akceptační testy (UAT):** konečná fáze testování, kdy se software testuje za reálných podmínek s účastí koncových uživatelů. Až software projde touto fází, je považován za připravený k nasazení. (2 str. 28–29, 13)

V-model se často používá ve spojení s metodikami Agile, Scrum nebo i vodopádovým modelem. Je vhodný pro projekty, kde je důležitá vysoká kvalita a přesné splnění specifikací. (2 str. 29)

3.2.10 W-Model

W-Model je rozšířením tradičního V-Modelu v oblasti vývoje softwaru a dodává na jeho základní koncepty další vrstvu komplexnosti. W-Model zdůrazňuje, že aktivita vývoje a aktivita testování by měly probíhat paralelně a iterativně místo toho, aby byly vnímány jako následné kroky. V tomto modelu se vývoj a testování neodehrávají pouze lineárně, ale jsou prováděny v několika iteracích, což umožňuje snazší adaptaci na změny a nové požadavky. (2 str. 29)

Obrázek 4 W-Model



Zdroj (14)

W-Model je ve své podstatě podobný V-Modelu s tím rozdílem, že každá fáze vývoje je doprovázena paralelní fází testování. To znamená, že fáze testování se spouští spolu s fázemi vývoje. (2 str. 29–30)

Klíčové charakteristiky W-Modelu:

- **Iterativní přístup:** každá fáze testování je prováděna iterativně a paralelně s odpovídající fází vývoje;
- **Pružnost:** vzhledem k paralelnímu a iterativnímu charakteru modelu je snazší adaptovat se na změny v požadavcích nebo designu. (2 str. 29–30)

3.2.11 Definice verifikace

Verifikace se věnuje otázce: „Děláme věci správně?“ Jde o proces, v rámci kterého se ověřuje, zda produkt splňuje specifické požadavky nebo návrhy, které byly stanoveny v průběhu fází vývoje. Verifikace je zaměřena na kontrolu mezivýstupů v průběhu vývoje, jako jsou například dokumentace a kód. (2 str. 30)

3.2.12 Definice validace

Validace se ptá: „Děláme správnou věc?“ Jde o proces ověření, že konečný produkt splňuje požadavky a očekávání uživatele nebo klienta. Validace zjišťuje, zda produkt jako celek řeší to, co bylo původně zamýšleno, a zda je vhodný pro jeho zamýšlené použití v reálném světě. (2 str. 30)

3.2.13 Shrnutí

V této kapitole byly prozkoumány termíny Software Development Life Cycle (SDLC) a Software Testing Life Cycle (STLC). SDLC ukazuje celkový pohled na vývoj, zatímco STLC je zaměřeno na proces testování jako nedílnou součást vývojového cyklu. Dále byly zmíněn V-Model a W-Model, byl popsán jejich rozdíl a použití.

3.3 Typy prostředí

Je zřejmé, že při vývoji softwaru jsou potřebné prostředí s běžící aplikací pro různé účely. Tímto se bude zabývat tato kapitola. A konkrétně klíčovými aspekty spojenými s prostředím, ve kterém se provádí testování softwaru. Bude zaměřeno na různé typy testovacích a produkčních prostředí. Budou vysvětleny jejich účely.

3.3.1 Atributy testovacího prostředí

Zde jsou uvedené obecné atributy testovacího prostředí:

1. **Název** – slouží pro identifikaci prostředí;
2. **Zkratka** – zkratka testovacího prostředí, příklad: DEV, UAT;
3. **Funkce** – slouží pro vysvětlení role testovacího prostředí;
4. **Shodnost s produkčním prostředím** – produkční prostředí by se měla rovnat ostatním prostředím;
5. **Výkonnost** – výkon prostředí;

6. **Správa prostředí** – se týká řízení a konfigurace testovacího prostředí, ve kterém jsou prováděny testy softwaru. Toto prostředí zahrnuje hardware, software, konfigurace, testovací data a další součásti, které jsou potřebné k provedení testů;
7. **Integrovanost** – se týká schopnosti aplikace se napojit nebo integrovat s dalšími aplikacemi nebo systémy. (2 str. 123–124)

3.3.2 Pískoviště

Prostředí pískoviště, známé také jako sandbox, je prostředí, které je navrženo tak, aby umožnilo bezpečné spouštění a testování neověřeného kódu nebo aplikací. V takovém prostředí je kód izolován od zbytku systému, což zabraňuje potenciálním škodlivým interakcím s ostatními částmi systému. (2 str. 125)

Pískoviště je často používáno pro testování nových funkcí softwaru. Izolace od hlavního systému zajišťuje, že jakékoli chyby nebo problémy ovlivní pouze pískoviště, nikoli produkční nebo nějaké jiné prostředí. (2 str. 125)

3.3.3 Vývojové prostředí – DEV

Vývojové prostředí DEV (Development Environment) je specializované prostředí, které je záměrně navrženo pro vývojáře, který provádí vývoj, jednotkové testování a ladění softwaru. Prostředí DEV je izolované od produkčního prostředí, aby se zabránilo jakýmkoliv náhodným chybám nebo problémům, které by mohly ovlivnit skutečné uživatele. Nemusí být shodné s produkčním prostředím. (2 str. 125)

3.3.4 Testovací prostředí pro systémové testy – SYS

Testovací prostředí pro systémové testy (označované také jako SYS, zkratka z anglického „System Testing Environment“) je prostředím, které slouží k detailnímu ověření celkové funkčnosti a integračních aspektů softwarového systému před jeho nasazením do produkčního prostředí. (2 str. 125)

Vyvíjená aplikace se může skládat z několika dalších aplikací nebo mikroslužeb, které musí mezi sebou spolupracovat, a proto je zapotřebí otestovat jejich vzájemnou integraci. Systémové testy na tomto prostředí jsou zaměřeny na komplexní otestování celého softwarového systému, aby se zajistilo, že spolu s napojenými služby fungují jako celek. (2 str. 125)

3.3.5 Testovací prostředí pro integrační testy – INT

Testovací prostředí pro integrační testy (označované také jako INT) je prostředí zaměřené na zkoumání a ověřování vzájemného propojení a spolupráce mezi různými komponenty nebo moduly softwarového systému. Na tomto prostředí se provádějí testy, které identifikují případné nesrovnalosti, chyby nebo problémy v rámci interakcí mezi těmito komponenty nebo moduly. (2 str. 125–126)

3.3.6 Testovací prostředí pro podporu produkce – PRS

Testovací prostředí pro podporu produkce (angl. PRS – Production Support) je prostředí pro co nejvíce možnou simulaci produkčního prostředí. PRS poskytuje platformu pro testování chyb z produkčního prostředí. (2 str. 126)

Prostředí PRS je navrženo tak, aby co nejvíce simulovalo skutečné produkční prostředí, včetně konfigurace, infrastruktury a datových proudů. To umožňuje provádění komplexních testů za co nejbližších podmínek tomu, jak je software skutečně využíván uživateli na produkci. (2 str. 126)

3.3.7 Preprodukční testovací prostředí – PRE

Preprodukční testovací prostředí (PRE) je prostředí navržené tak, aby co nejvíce napodobovalo skutečné produkční prostředí, včetně hardwaru, softwaru, sítí a dalších technologických aspektů. Nasazení na toto testovací prostředí přichází před samotným produkčním nasazením a slouží jako důležitý mezistupeň pro ověření, že software je připravený na bezproblémový provoz v produkčním prostředí. V preprodukčním testovacím prostředí probíhají testy vysoké dostupnosti, zátěžové a výkonnostní testy. Tyto typy testů se zaměřují na zajištění kvality, spolehlivosti a výkonnosti softwarového produktu, což má za následek maximální uživatelskou spokojenost a minimalizaci rizik spojených s nasazením nových funkcionalit. (2 str. 126)

Účelem tohoto prostředí je identifikace a řešení problémů a nedostatků ještě předtím, než se dostanou ke skutečným uživatelům. A proto musí být co nejvíce podobné produkčnímu prostředí. (2 str. 126)

3.3.8 Školicí prostředí – EDU

Školicí prostředí EDU (Education Environment) je specializované prostředí, které slouží ke školení uživatelů. V tomto prostředí probíhá školení účastníků práce se softwarem a jeho

jednotlivými komponenty. To je dosaženo prostřednictvím realistických simulací procesu, interaktivních cvičení, které jsou navržena tak, aby podporovala efektivní učení a praktickou aplikaci konceptů. Školicí prostředí EDU je izolované od produkčního prostředí, což umožňuje účastníkům experimentovat a zkoušet různé postupy a scénáře bez dopadů na reálné produkční systémy. Prostředí se nemusí shodovat s produkční verzí softwaru. Slouží také jako platforma pro vytváření a distribuci vzdělávacích materiálů pro budoucí uživatele systému. (2 str. 127)

3.3.9 Produkční a záložní testovací prostředí – PROD

Produkční prostředí je často označováno zkratkou „PROD“ je prostředí, které představuje konečný produkt a cíl vývoje. Produkční prostředí je místo, kde se software stává plně funkčním a dostupným pro koncové uživatele. V produkčním prostředí jsou prováděny závěrečné testy a ověření, aby bylo zajištěno, že software pracuje bezchybně a splňuje všechny funkční požadavky. Zároveň je vytvořena záložní verze prostředí (*backup*), která slouží jako plán B v případě výpadku či nečekaných problémů v produkčním prostředí. Záložní testovací prostředí umožňuje rychlou obnovu provozu a minimalizuje potenciální dopady na uživatele. Občas se dá narazit i na to, že toto prostředí pojmenují jako *disaster recovery*. (2 str. 126–127)

3.3.10 Shrnutí

Kapitola shrnuje různé typy testovacích prostředí v softwarovém vývoji, včetně jejich charakteristik a účelů. Atributy testovacího prostředí zahrnují identifikaci (název a zkratku), funkci, shodnost s produkčním prostředím, výkonnost, správu, a integrovanost s jinými systémy. Dále jsou popsány specifické typy prostředí, jako jsou prostředí pískoviště (sandbox) pro bezpečné testování neověřeného kódu, vývojové prostředí (DEV) pro vývoj a ladění, systémové testovací prostředí (SYS) pro ověřování funkčnosti a integrace, integrační testovací prostředí (INT) pro testování vzájemného propojení komponent, prostředí pro podporu produkce (PRS) simuluje produkční prostředí pro testování oprav a aktualizací, preprodukční testovací prostředí (PRE) pro finální ověření před nasazením, školicí prostředí (EDU) pro uživatelské školení, a produkční a záložní testovací prostředí (PROD) pro finální užívání softwaru a případnou obnovu po výpadku.

3.4 Metody a techniky testování

Testování softwaru je rozmanitý a multidisciplinární proces, jenž zahrnuje široké spektrum činností od ověření jednotlivých funkcí po celkovou stabilitu a výkonnost systému. Každý typ

testu hraje specifickou roli v životním cyklu vývoje softwaru a přispívá k odhalení a nápravě potenciálních problémů, které by mohly uživatele ovlivnit. V této kapitole budou vysvětleny základní typy testů, které se provádějí při testování softwaru. Budou probrány jednotlivé techniky testování. Bude provedena klasifikace jednotlivých testů.

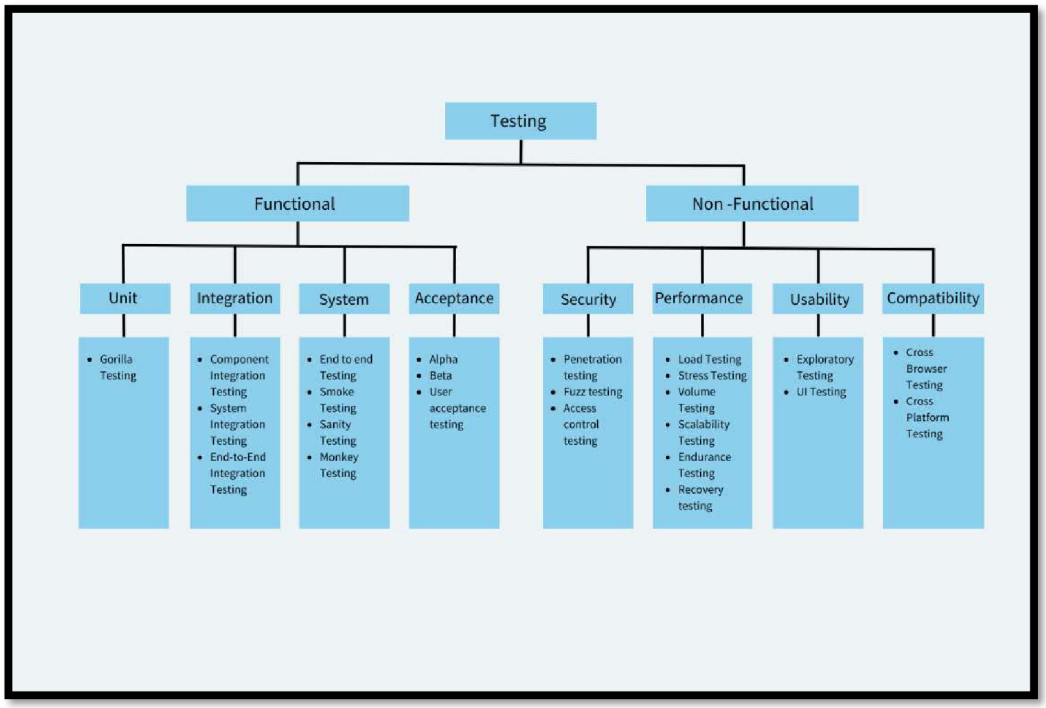
3.4.1 Funkční testování

Funkční testování je zaměřeno na testování specifických funkcionalit softwarové aplikace. Pomocí tohoto testování se ověřuje, zda software funguje v souladu s definovanými požadavky a specifikacemi. Během funkčního testování se testery snaží simulovat chování koncového uživatele a ověřují správné chování aplikace. (5)

3.4.2 Nefunkční testování

Nefunkční testování je zaměřeno na testování aspektů softwaru, které nejsou přímo spojeny s konkrétní funkcionalitou, ale spíše s kvalitou a uživatelskou zkušeností. Pomocí tohoto typu testování se ověřuje, že systém splňuje různé nefunkční požadavky, jako jsou výkon, bezpečnost, použitelnost, spolehlivost a kompatibilita s ostatními nástroji nebo prohlížeči. (5)

Obrázek 5 Klasifikace typů testování



Zdroj (5)

3.4.3 Techniky testování

Techniky testování softwaru jsou zaměřené na zajištění kvality softwaru. Mezi nejrozšířenější přístupy patří testování typu „black-box“, „white-box“ a „grey-box“. Tyto techniky poskytují různé perspektivy na testovaný software. Výběr vhodné techniky testování závisí na specifických cílech testování, dostupnosti zdrojů, časovém rámci a požadované úrovni důkladnosti. (5)

- **Black-Box testing** je technika testování, která je soustředěna na funkčnost softwaru bez nutnosti znalosti vnitřní struktury a implementace kódu. Testery vytvářejí testovací případy založené na specifikacích a požadavcích softwaru, aby ověřily, zda výstupy odpovídají očekávaným výstupům pro dané vstupy. (5)
- **White-Box testing** je technika testování, která na rozdíl od black-box vyžaduje znalost vnitřní logiky a struktury kódu aplikace. Testery píšou testy, které prověřují konkrétní části kódu, včetně větví (příklad: podmíněný příkaz if-else nebo switch), cyklů a jednotlivých funkcí. (5)

- **Grey-Box testing** kombinuje prvky černé a bílé skříňky. Testeři mají omezenou znalost vnitřní struktury aplikace, kterou využívají k efektivnějšímu vytváření testovacích případů, zatímco se stále zaměřují na funkčnost z pohledu uživatele. (5)

3.4.4 Typy funkčních testů

Jednotkové testování

Jednotkové testování (angl. Unit Testing) je testovací technika používaná k ověření funkčnosti jednotlivých komponent nebo „jednotek" kódu. Pomocí izolace každé části programu se testuje, že tyto jednotlivé části fungují správně. Příklad: otestovat správné odeslání formuláře po stisknutí tlačítka. (5)

Integrační testování

Integrační testování (angl. Integration Testing) je zaměřeno na testování interakcí mezi různými moduly nebo komponentami systému, aby se ověřilo, že tyto komponenty fungují společně podle očekávání. Na rozdíl od jednotkového testování, které se zaměřuje na ověření funkčnosti izolovaných částí kódu, integrační testování zkoumá celý systém jako soubor propojených komponent a je zaměřeno na identifikaci problémů v rozhraních a vzájemných interakcích mezi komponentami. Tento typ testování slouží pro odhalení chyb, které se mohou objevit, pouze když jsou komponenty nebo systémy spojeny dohromady, včetně problémů s komunikací, datovými strukturami nebo sdílenými stavy. (5)

Systémové testování

Systémové testování (angl. System Testing) je testování, kde se testuje kompletní a plně integrovaný softwarový systém, aby se ověřilo, že splňuje stanovené požadavky. Během systémového testování se testery snaží simulovat reálné uživatelské prostředí co nejpřesněji, aby se ujistily, že software bude fungovat správně a efektivně v reálném světě. (5)

Typy systémových testů: (5)

- **E2E testing** neboli End-to-End testování, je typ testování používaný k ověření, zda aplikace funguje správně od začátku a až do konce průběhu určitého procesu. E2E zahrnuje testování celého softwarového produktu ve scénářích, které napodobují reálné uživatelské operace. Ověřuje se, že systém funguje jako celek, včetně jeho integrace

s externími systémy, pro zajištění, že všechny součásti pracují společně podle očekávání od momentu, kdy uživatel zahájí proces, až po finální výsledek;

- **Monkey Testing** je typ testování, při kterém tester nebo automatizovaný skript provádí testy náhodným způsobem bez pevně daných scénářů nebo očekávaných výsledků;
- **Smoke Testing** je typ testování, který se provádí na nové verzi softwaru, aby se rychle ověřilo, že klíčové funkce aplikace jsou funkční a že nejsou přítomny žádné kritické chyby, které by zabránily nasazení nebo dalšímu testování.

Akceptační testování

Akceptační testování je poslední fází testování softwaru před jeho uvedením do provozu nebo předáním zákazníkovi. Pro ověření, že systém nebo aplikace splňuje uvedené požadavky, se tento typ testování provádí buď koncovými uživateli, anebo investory. (5)

Během akceptačního testování se obvykle provádějí různé typy testů, včetně uživatelského akceptačního testování (UAT), kde koncoví uživatelé testují software, aby se ujistili, že splňuje jejich potřeby a očekávání. Může zahrnovat také obchodní akceptační testování angl. Business Acceptance Testing (BAT). (5)

Typy akceptačních testů: (5)

- **Alpha Testing** je fáze testování softwaru, která se obvykle provádí interně před uvedením produktu na trh. Tento typ testů je prováděn v kontrolovaném prostředí a testy vykonávají jak vývojový tým, tak někdy i vybraní uživatelé zvenčí;
- **Beta Testing** je druhý krok v testování softwaru, který následuje po Alpha testování, a obvykle se provádí s koncovými uživateli softwaru. Tím vývojový tým získá zpětnou vazbu a identifikuje jakékoli problémy nebo chyby, které nebyly odhaleny během interního testování;
- **Sanity Testing** je povrchní úroveň testování, která se provádí po opravě chyb nebo po drobných změnách v softwaru, aby se ověřilo, že oprava nezpůsobila další problémy v souvisejících funkcionalitách.

3.4.5 Typy nefunkčních testů

Testování zabezpečení

Testování zabezpečení (angl. Security Testing) je typ testování, který se zaměřuje na identifikaci slabých míst, chyb a zranitelností v softwarovém systému, které by mohly být zneužity potenciálními útočníky. (5)

Typy testů pro testování zabezpečení: (5)

- **Penetration Testing** (česky penetrační testy) také označované jako „pen testing“, je metoda testování zabezpečení, při které testující simuluje útoky na softwarový systém, aby identifikoval zranitelnosti, které by mohly být zneužity potenciálními útočníky;
- **Fuzz Testing** je technika automatického testování zabezpečení, která zahrnuje vytváření a odesílání neplatných, neočekávaných nebo náhodně generovaných dat do softwarové aplikace. Tento typ testování pomáhá identifikovat vstupy, kde se dají vložit neplatná data;
- **Testování řízení přístupu** (angl. Access Control Testing) se zaměřuje na ověření, zda systém správně implementuje politiky přístupových práv pro uživatele. Ověřuje se, že uživatele mají přístup pouze k těm funkcím a operacím, ke kterým mají být oprávněni.

Výkonnostní testování

Výkonnostní testování (angl. Performance Testing) je typ testování, který se zaměřuje na určení výkonnosti a schopnosti softwarového systému pokračovat ve fungování pod určitým zatížením. Pomáhá identifikovat slabá místa v aplikaci, která by mohla vést k neefektivnímu využívání zdrojů, zpomalení odezvy nebo dokonce k selhání systému pod vyšším zatížením. (5)

Typy testů pro výkonnostní testování: (5)

- **Load Testing** (česky testování zátěže) je zaměřeno na simulaci skutečného zatížení na aplikaci nebo na systém za účelem určení, jak systém reaguje a zvládá očekávané množství uživatelů nebo transakcí. Během Load Testingu se ověřuje, zdali aplikace může efektivně a bez problémů zpracovávat svou běžnou zátěž. Příklad simulace zátěže 200 uživateli;
- **Stress Testing** testuje softwarový systém nebo aplikaci za podmínek extrémního zatížení, které překračují normální operační kapacitu, s cílem určit, při jakém bodu aplikace selže a jak se chová v kritických situacích, včetně zotavení po chybách;

- **Volume Testing** je soustředěno na ověření schopnosti systému zpracovávat velké množství dat. Testuje se, jak různé aspekty systému, jako jsou databáze a transakce, fungují s rozsáhlými objemy informací a zda jsou schopny udržet si výkon a stabilitu;
- **Scalability Testing** hodnotí schopnost softwaru růst a efektivně zvládat zvyšující se zatížení tím, že přidává zdroje, jako jsou procesory, paměť nebo síťové kapacity. Tento test pomáhá určit, jak dobře se systém může přizpůsobit rostoucím požadavkům bez poklesu výkonu;
- **Endurance Testing** (česky testování vytrvalosti) ověřuje, jak software funguje pod normálním nebo mírně zvýšeným zatížením po delší dobu. Tímto odhalí problémy, jako jsou úniky paměti a degradace výkonu, které by se mohly postupem času vyvinout.

Testování použitelnosti

Testování použitelností (angl. Usability Testing) je proces hodnocení softwarové aplikace nebo webové stránky z pohledu koncového uživatele. Pomocí testování použitelnosti se odhaluje, kde uživatelé čelí problémům nebo mají obtíže s používáním produktu. Pomocí výsledku tohoto testování se dá zjistit, jak lze zlepšit použitelnost produktu. Tento typ testování se zaměřuje na to, aby se zjistilo, jak intuitivní, snadno naučitelný a efektivní je produkt v očích koncových uživatelů. (5)

Typy testů na testování použitelností: (5)

- **Exploratory Testing** je přístup k testování, kde testeři současně učí o softwaru, navrhují testovací případy a provádějí testy samostatně bez předem předepsaných scénářů. Tento typ testování je řízen zvědavostí a intuicí testera, což umožňuje flexibilně reagovat na nalezené problémy, hlouběji prozkoumat a najít nějaké defekty aplikací;
- **User Interface Testing** se zaměřuje na testování uživatelského rozhraní softwarové aplikace (GUI), aby se zajistilo, že všechny grafické prvky, navigace a interaktivní funkce jsou intuitivní z pohledu uživatele;
- **Accessibility Testing** ověřuje, zda je software přístupný a použitelný pro osoby s různými druhy postižení, včetně zrakových, sluchových, motorických a kognitivních omezení.

Testování kompatibility

Testování kompatibility (angl. Compatibility Testing) je typ testování, pomocí kterého se ověřuje, zda software je kompatibilní s různými prostředími. To zahrnuje testování na různých operačních systémech, prohlížečích, síťových prostředích, hardwarových konfiguracích a dalších softwarových aplikacích. Pro testování kompatibility se používají Cross Browser Testing a Cross platform Testing. (5)

3.4.6 Shrnutí

Tato kapitola se zabývá testováním softwaru, klíčovým procesem zajišťujícím, že aplikace splňuje své požadavky a očekávání uživatelů. Testování je rozděleno na dvě hlavní kategorie: funkční a nefunkční testování, každé s vlastními specifickými cíli a metodami. Zatímco funkční testování se zaměřuje na přímou kontrolu specifických funkcí softwaru a jeho schopnost plnit definované požadavky, nefunkční testování zkoumá kvalitativní aspekty aplikace, jako jsou výkon, bezpečnost a použitelnost. Dále se věnuje různým technikám testování, včetně black-box, white-box a grey-box přístupů, které umožňují testerům efektivně hodnotit software z různých perspektiv. Kapitola také obsahuje popis typů funkčních a nefunkčních testů, od jednotkového testování po testování kompatibility, což nám poskytuje komplexní přehled o tom, jakým způsobem lze softwarové produkty testovat a ověřovat před jejich finálním nasazením.

3.5 Front-End a Back-End

Kapitola se zaměřuje na rozdělení a hlubší pochopení dvou základních stavebních bloků softwarového vývoje: back-endu a front-endu. Tato dělba je základem architektury moderních softwarových aplikací, které slouží k rozdělení odpovědností mezi serverovou (back-end) a klientskou (front-end) část aplikace. Tímto způsobem je zajištěna efektivní a uživatelsky přívětivá interakce s aplikací, zatímco komplexní logika a zpracování dat probíhá na pozadí.

3.5.1 Back-End

Back-end, v kontextu architektury softwarových aplikací, představuje soubor serverových komponent, které jsou zodpovědné za zpracování logiky aplikace, manipulaci s daty, správu databází a komunikaci se serverovými službami. Tento termín se odvíjí od dělení softwarové architektury na dvě základní části: front-end, který uživatelům poskytuje vizuální rozhraní

a umožňuje interakci s aplikací, a back-end, který funguje na serveru a zajišťuje funkcionalitu aplikace neviditelnou pro uživatele. (11)

Back-end systémy jsou navrženy tak, aby zpracovávaly požadavky přijaté od klientů (typicky od front-end aplikací) a prováděly potřebné operace, jako jsou vytváření, čtení, aktualizace a mazání dat (operace CRUD) v databázích. Dále back-end systémy by měly umožňovat autentizace uživatelů, zpracování obchodní logiky a integrace s externími API. Výsledky těchto operací jsou poté odesílány zpět klientovi. (10, 11)

Pro vývoj back-endu se využívají různé programovací jazyky a technologie. Mezi nejčastěji používané programovací jazyky patří Java, Python, Ruby, Scala, C#, C++, Perl, PHP a Node.js. Tyto jazyky poskytují rozhraní pro interakci s databázemi, které mohou být relační nebo NoSQL. Výše zmíněné jazyky také poskytují rozhraní pro vytváření samotné aplikace. (10, 11)

Důležitým aspektem back-end vývoje je zajištění vysoké dostupnosti, bezpečnosti a škálovatelnosti aplikace. To vyžaduje důkladné plánování architektury, implementaci bezpečnostních opatření, jako například šifrování dat. (10, 11)

3.5.2 Front-End

Front-end v oblasti vývoje softwaru se odkazuje na client-side část aplikace nebo webové stránky, se kterou uživatel přímo interaguje. Zahrnuje vše, co uživatelé vidí a používají, jako jsou grafické uživatelské rozhraní (GUI), obrázky, tlačítka, pole pro vstup a navigace na webu. Účelem front-endu je poskytnout způsoby interakce s aplikaci či webovou stránkou. (4)

Vývoj front-endu zahrnuje použití technologií jako HTML (HyperText Markup Language) pro tvorbu struktury a obsahu stránky, CSS (Cascading Style Sheets) pro vizuální stylizaci jednotlivých prvků stránky a JavaScript pro kontrolu formulářů, tvorbu interakci nebo dynamické povahy stránky. Moderní front-end vývoj často využívá různé frameworky a knihovny, jako jsou React, Redux, Angular a Vue. Frameworky ve své podstatě usnadňují vytváření dynamických a responzivních uživatelských rozhraní. (4, 10)

3.5.3 API

API (Application Programming Interface) je sada pravidel, protokolů a nástrojů pro vytváření softwarových aplikací. Jedná se o rozhraní, které umožňuje různým softwarovým komponentám

komunikovat mezi sebou. API definuje metody a data, která mohou být použita pro interakci mezi různými softwarovými aplikacemi, nezávisle na jejich interní implementaci. (9)

API je zaměřeno na různé úrovně softwaru, od nízkoúrovňových funkcí operačních systémů až po vysokoúrovňové webové služby. Například, webové API poskytuje metody pro komunikaci mezi webovou aplikací a serverem pomocí požadavků HTTP, což umožňuje vytvářet, číst, upravovat a mazat data na serveru. (9)

3.5.4 Shrnutí

Výše byly vysvětleny pojmy back-end a front-end a probrány jejich charakterní vlastnosti. Většina aplikací má front-end, odpovídající za vizualizaci a vzhled uživatelského rozhraní, které pomocí technologií JavaScript může být i dynamicky navrženo, a back-end, který zpracovává požadavky od uživatelů a ukládá data do databáze. Jak pro front-end, tak i pro back-end byly vyjmenovány programovací jazyky a technologie, ve kterých mohou být napsány.

3.6 Doporučení ISTQB pro automatické testování

Je zřejmé, že se v průběhu let na základě dosažených poznatků změnil způsob přístupu k testování softwarových systémů. Z těchto poznatků byla vyvinuta doporučení pro osoby, které se testingem zabývají. Tato kapitola se zabývá danými doporučeními. Budou probrána jednotlivá doporučení od ISTQB pro automatické testování.

3.6.1 Architektura automatizace testování

Architektura automatizace testování musí velmi těsně souviset se softwarovým produktem, konkrétně s jeho architekturou, která definuje funkční a nefunkční požadavky. Tyto požadavky musí být udržitelné. (1 str. 13)

3.6.2 Strategie automatizovaného testování

Strategie automatického testování by měla být nejen praktická a konzistentní, ale také by měla brát v úvahu udržitelnost a soudržnost testovaného softwarového systému. Je zapotřebí se zaměřit na dlouhodobou udržitelnost testovacího procesu a zajistit, aby testy zůstávaly relevantní i při budoucích změnách v testovaném softwaru. Při plánování strategie automatického testování je také nezbytné zvážit rizika, přínosy, a hlavně náklady, které jsou spojené s aplikací automatizovaných testů. (1 str. 14)

3.6.3 Framework pro automatizované testování

Framework by měl mít dobrou dokumentaci pro rychlejší orientování uživatelů v jeho nastavení a použití. Měl by být také dobře udržitelný. (1 str. 14)

Pro vybraný framework by také měl být vybrán způsob reportování stavu testů. Toto potom zohledňuje kvalitu vyvíjeného produktu. Pro neúspěšné testy by měl existovat způsob jejich vyřešení a revize. (1 str. 14)

Pro exekuci automatických testů by mělo existovat zvláštní prostředí, na kterém by tyto testy běžely. Toto testovací prostředí musí být konzistentní za účelem opakovaného spuštění testů. Pokud prostředí a data nejsou správně spravována a kontrola nad nimi je nedostatečná, může to vést k „falešným“ výsledkům testování. (1 str. 14)

Automatické testy musí být dobře popsány a mít uvedený název. V tomto popisu nebo názvu musí být jisté, jaká přesně komponenta nebo požadavek se testuje. (1 str. 14)

Automatické testovací scénáře, které jsou napsány pro automatizaci, by měly být navrženy tak, aby byly snadno udržovatelné. Údržba těchto testů nesmí zabírat značnou část času a úsilí věnovaného automatizaci. Je také důležité, aby změny v testovacích skriptech bylo možné snadno nasadit. V případě, kdy některé testy již nebudou aktuální, musejí být odstraněny. (1 str. 14–15)

3.6.4 Shrnutí

Výše byla vybrána a popsána jednotlivá doporučení od ISTQB pro automatické testování. Architektura automatizace testování musí souviset se softwarovým produktem. Testy by vždy měly být udržovány, aby neztratily svoji relevantnost. Framework musí být pečlivě vybírán, aby ti, kdo ten framework budou používat, měli dostupné všechny potřebné informace, například dokumentaci.

4 Vlastní práce

V této části diplomové práce bude proveden výběr softwarových řešení pomocí analýzy klíčových slov. Jako klíčová slova budou zvoleny názvy softwarů nebo frameworků pro automatické testování. Analýza klíčových slov softwarů proběhne jak pro automatické testy front-endu, tak i automatické testy back-endu a API. Následně bude s každým ze softwarů proveden experiment na zvoleném příkladu.

4.1 Analýza klíčových slov

Pro identifikaci trendu byla provedena rešerše informačních zdrojů, kde byly vybírány názvy frameworků nebo softwarů, které se používají zvlášť pro automatické testování front-endu a back-endu. Pro analýzu byla použita aplikace Google Trends.

Byly nastaveny následující filtry:

- Země: Celosvětově
- Doba: Posledních 5 let
- Kategorie: Všechny, Programování, Testování a měření, Počítačová věda, Počítače a elektronika
- Vyhledávání na webu

Kategorie se mění za účelem detailnějšího zkoumání.

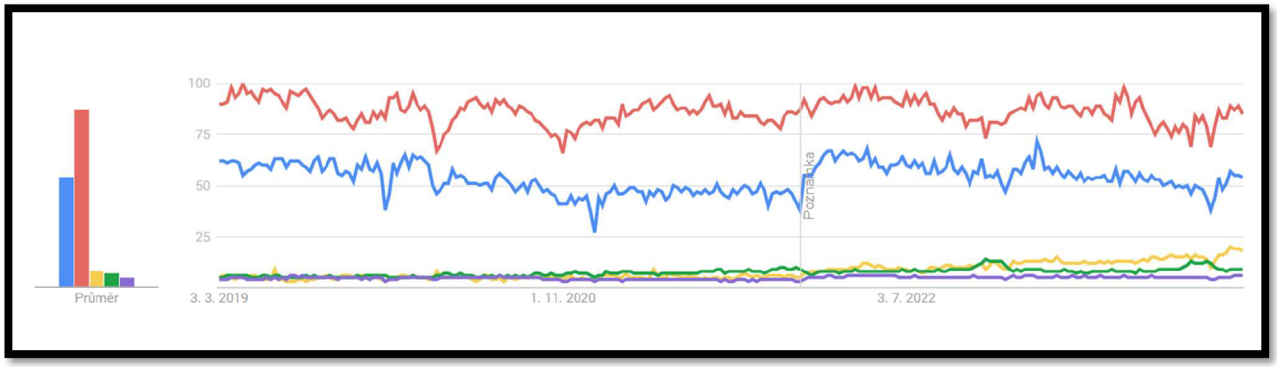
4.1.1 Analýza trendu softwaru pro testování FE

Klíčová slova:

- Selenium
- Cypress
- Testsigma
- Headspin
- Smartbear
- PlayWright
- TestComplete
- Katalon
- Ranorex

- Squish
- Nightwatch.js
- Puppeteer

Obrázek 6 Zájem v průběhu času FE



Zdroj: vlastní zpracování, Google Trends

Legenda: Selenium, Cypress, Playwright, Squish, Puppeteer

Poznámka: 0 znamená, že pro výraz nebyl shromážděn dostatek dat, 50, že výraz měl poloviční popularitu, 100 je nejvyšší popularita výrazů.

Po zkoumání klíčových slov, bylo zjištěno, že největší popularitu za 5 let mají **Selenium** a **Cypress**. Ostatní frameworky nezískaly velkou popularitu. O Selenium měly velký zájem takové země jako Indie, Turecko, Japonsko, Jižní Korea, Pákistán. Zájem o Cypress projevily země jako Spojené státy, Kanada, Austrálie, Mexiko, Švédsko. Pomocí aplikace Cypress se dá testovat API, a proto bude zvolena pro experiment BE.

4.1.2 Analýza trendu softwaru pro testování BE a API

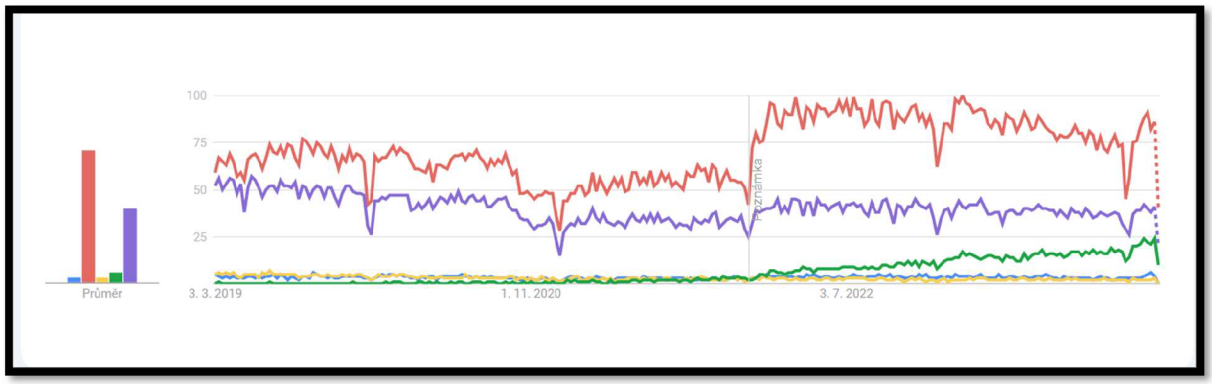
Pro analýzu l

Klíčová slova:

- Testim
- Postman
- SOAP UI
- TestRigor
- JMeter

- PlayWright
- HammerDB
- SLOB
- TOAD
- Rest Assured

Obrázek 7 Zájem v průběhu času BE a API



Zdroj: vlastní zpracování, Google Trends

Legenda: Rest Assured, Postman, SOAP UI, PlayWright, JMeter

Poznámka: 0 znamená, že pro výraz nebyl shromážděn dostatek dat, 50, že výraz měl poloviční popularitu, 100 je nejvyšší popularita výrazů.

Bylo zjištěno, že největší popularitu má **Postman**, **JMeter** a **Playwright**, Ostatní frameworky neměly vysokou popularitu. O Playwright mají největší zájem v Gruzii, Litvě, Norsku, Finsku a Polsku. O JMeter v Japonsku, Hongkongu, Číně, Šrí Lance. U PlayWrightu se dá testovat i front-end, a proto bude proveden i experiment se front-endem.

4.1.3 Výsledky

Pro napsání testů budou zvoleny následující softwary a frameworky:

- Postman
- JMeter
- Cypress

- Playwrigth
- Selenium

4.2 Typy experimentů

1. **Porovnání výkonnosti a efektivity**

Cíl: Ověřit, jak rychle a efektivně mohou jednotlivé nástroje vykonávat standardní testovací scénáře.

Metodika: Bude vytvořena sada identických nebo velmi podobných testovacích případů pro testování webového rozhraní a API. Bude změřen čas potřebný k dokončení testů. Výsledky budou porovnány.

2. **Podpora různých typů testů**

Cíl: Ověřit, jaké typy testů (např. funkční testování, výkonnostní testování, E2E testy) každý nástroj podporuje.

Metodika: Prostudováním dokumentace a analýzou existujících příkladů bude posouzeno, jaké typy testování každý nástroj podporuje. Provedení technicky možných typů testů.

3. **Znalost programovacích jazyků**

Cíl: Zjistit, jaké znalostí programovacích jazyků bude muset znát potenciální uživatel.

Metodika: Studium a analýza dokumentace softwarů a frameworků pro zjištění, jaké programovací jazyky potřebuje umět tester pro napsání testů.

4. **Porovnání přesnosti a spolehlivosti detekce chyb**

Cíl: Hodnotit, jak přesně a spolehlivě nástroje identifikují a hlásí chyby v testovaném softwaru.

Metodika: Záměrně budou zavedeny známé chyby nebo regrese do testované aplikace a bude sledováno, jak efektivně je jednotlivé nástroje detekují a reportují.

4.3 Exekuce experimentů

4.3.1 Experiment 1 Porovnání výkonnosti a efektivity – Front-End

V této části jsou definovány jednotlivé scénáře, kterým projde každý ze softwarů jak na lokálním serveru, tak i na vzdáleném. Scénáře se vyplňují postupně jeden za druhým. Počítá se čas strávený na každém ze scénářů. Zaznamenává se i celkový čas.

Scénáře pro vzdálený server

Scénář 1 – Článek o parkování:

- Navštiv https://www.pef.czu.cz/
- Klikni na **Studium**
- Klikni na **Informace pro studenty**
- Klikni na **Parkování v areálu ČZU**
- Zkontroluj název článku „Parkování v areálu ČZU“

Scénář 2 – Nákup knihy na eshop.czu.cz:

1. Navštiv https://eshop.czu.cz/
2. Klikni na **první obrázek** produktu v sekci produktů.
3. Klikni na tlačítko „**Přidat do košíku**“.
4. Ověř, že se zobrazí zpráva o úspěchu s textem „Teorie účetnictví – příklady, 362 byl úspěšně přidán do košíku.“
5. Klikni na odkaz košíku v pravém horním rohu.
6. Klikni na tlačítko „Pokračovat k pokladně“.
7. Vyber možnost „Pokračovat jako host“ nebo „Registrovat“ a klikni na tlačítko pro potvrzení výběru.
8. Vyplň fakturační údaje: Jméno („Testovací“), Příjmení („Uživatel“), E-mail („someone@example.com“), Adresa („Nějaká adresa 1“), Město („Praha“), PSČ („12000“), Telefonní číslo („666666666“).
9. Klikni na tlačítko „Pokračovat“ v sekci fakturačních údajů.
10. Vyber způsob dopravy a klikni na tlačítko „Pokračovat“.
11. Vyber způsob platby a klikni na tlačítko „Pokračovat“.
12. Počkej krátce, aby se stránka aktualizovala.
13. Zkontroluj, že v souhrnu objednávky je zobrazen produkt „Teorie účetnictví – příklady, 362“.

Scénář 3 – Prohlížení fotografií PEF na webu czu.cz:

1. Navštiv https://www.pef.czu.cz/
2. Klikni na položku „O fakultě“.
3. Vyber první dostupnou fotografii kliknutím na první prvek s lazy-load obrázkem.

4. Počkej 2 sekundy, aby se fotografie plně načetla a byla viditelná.
5. Klikni na tlačítko pro přechod na další fotografii (označeno třídou .lb-next).
6. Počkej 2 sekundy, než se načte další fotografie.
7. Opakuj krok 5 a 6 pro prohlížení další fotografie.
8. Počkej 2 sekundy a opět použij tlačítko pro přechod na další fotografii.
9. Počkej 2 sekundy, aby se načetla poslední fotografie.

Scénáře pro lokální server

Scénář 1 – Odeslání formuláře na webu Home Prague:

1. Navštiv webovou stránku http://homeprague/
2. Klikni na odkaz „O nás a kontakty“.
3. Vyplň jméno do pole s ID „name“ hodnotou „Vasily“.
4. Vyplň příjmení do pole s ID „surname“ hodnotou „Gusev“.
5. Vyplň e-mail do pole s ID „email“ hodnotou „gusevvas@mail.com“.
6. Vyplň telefonní číslo do pole s ID „phone“ hodnotou „+420666666666“.
7. Do pole pro zprávu s ID „message“ napiš: „Dobrý den, zaujala mi Vaše nabídka bytu na Václavském náměstí. A proto bych se chtěl domluvit na prohlídce. Prosím o zavolání nebo napsání e-mailu. Děkuji. Vasily Gusev.“
8. Klikni na tlačítko pro odeslání formuláře s ID „submitButton“.
9. Počkej na zobrazení modulu s poděkováním a ověř, že obsahuje text „Děkujeme Vám za odeslání nabídky, brzy Vás budeme kontaktovat.“
10. Zavři modul kliknutím na tlačítko „Zavřít“ v rámci modulu.

Scénář 2 – Kontrola vyhledávání na Webu HomePrague:

1. Navštiv domovskou stránku Home Prague na adrese http://homeprague/
2. Klikni na odkaz „Vyhledávání“.
3. Vyber dispozici bytu 1+kk kliknutím na odpovídající checkbox.
4. Vyber dispozici bytu 3+kk kliknutím na další dostupný checkbox.
5. Vyber lokalitu „Praha 2“.
6. Ověř, že první nemovitost v seznamu výsledků má popis „Pronájem bytu 1+k“.
7. Ověřte, že druhá nemovitost v seznamu výsledků má popis „Prodej bytu 3+kk“.

Scénář 3 – Prohlížení nabídky na Webu HomePrague:

1. Navštiv domovskou stránku Home Prague na adrese http://homeprague/.
2. Klikni na odkaz k detailu nemovitosti, který vede na stránku detail.php s parametrem property=1, tedy na detail první nabídky nemovitosti.
3. Počkej 2 sekundy.
4. Klikni na druhou fotografii nemovitosti a počkej 2 sekundy na její zobrazení.
5. Klikni na třetí fotografii nemovitosti a počkej 2 sekundy na její zobrazení.
6. Klikni na čtvrtou fotografii nemovitosti a počkej 2 sekundy na její zobrazení.
7. Ověř, že popis nemovitosti odpovídá textu: „Tento byt 1+kk na Lázeňské ulici v Praze 1, na Malé Straně, představuje skutečnou oázu klidu a luxusu. Obývací prostor je navržen s otevřeným konceptem, spojujícím spánek, odpočinek a pracovní činnosti v harmonickém celku. Vyznačuje se moderním designem, jemnými tkaninami a teplými neutrálními tóny, které vytvářejí pocit klidu a pohody. Ložnice je vybavena velkou a pohodlnou postelí, ideální pro odpočinek po dlouhém dni. Dominantní okno od podlahy až ke stropu nabízí dechberoucí výhled na noční oblohu a panorama města.“

Cypress

Pro provedení tohoto experimentu byl nainstalován Node.js verze v20.11.1. Dále byla nainstalována poslední verze Cypressu v13.6.6. Scénář se prováděl i v prohlížeči Chrome v122. Kód scénářů pro vzdálený server:

```
describe('Visit PEF CZU Website', () => {
  it('Successfully loads PEF CZU home page', () => {
    // Ignore uncaught exceptions
    cy.on('uncaught:exception', (err, runnable) => {
      // return false to prevent the error from failing this test
      if (err.message.includes('gtag is not defined')) {
        return false;
      }
    });

    cy.visit('https://www.pef.czu.cz/');
    cy.get('#nav > ul > :nth-child(2) > a', { timeout: 1000 }).click();
    cy.get('#nav > [role="menubar"] > li.active > ul > :nth-child(5) >
a').click();
```

```
    cy.get('[aria-labelledby="widget-3913-item-8485"] > article > .tileInner > .tile__data').click();
    cy.get('#widget-283-heading').should('have.text', 'Parkování v areálu ČZU');
  });
});


describe('Buy CZU Book', () => {
    it('Successfully loads EShop CZU home page', () => {
      cy.on('uncaught:exception', (err, runnable) => {
        // return false to prevent the error from failing this test
        if (err.message.includes('lastPrice is not defined') || err.message.includes('gtag is not defined')) {
          return false;
        }
      });
      cy.visit('https://eshop.czu.cz/');
      cy.get('.codeblock > .category-products > .products-grid > .first > .product-image > img').click();
      cy.get('.add-to-cart > .button > :nth-child(1) > span').click();
      cy.get('.success-msg > ul > li > span').should('have.text','Teorie účetnictví - příklady, 362 byl úspěšně přidán do košíku.')
      cy.get('.top-link-cart').click();
      cy.get('.totals > .checkout-types > li > .button > :nth-child(1) > span').click();
      cy.get('.col-1 > .form-list > :nth-child(1) > label').click();
      cy.get('#onepage-guest-register-button > :nth-child(1) > span').click();
      cy.get('#billing\\:firstname').type("Testovácí");
      cy.get('#billing\\:lastname').type("Uživatel");
      cy.get('#billing\\:email').type("someone@example.com");
      cy.get('#billing\\:street1').type("Nějaká adresa 1");
      cy.get('#billing\\:city').type("Praha");
      cy.get('#billing\\:postcode').type("12000");
      cy.get('#billing\\:telephone').type("666666666");

      cy.get('#billing-buttons-container > .button > :nth-child(1) > span').click();



      cy.get('#s_method_flatrate_flatrate').click();
      cy.get('#shipping-method-buttons-container > .button > :nth-child(1) > span').click();
```

```
      cy.get('#payment-buttons-container > .button > :nth-child(1) >
span').click();
      cy.wait(1000);
      cy.get('.product-name').should('have.text','Teorie účetnictví -
příklady, 362');

    });
  });

  describe('Look through PEF pictures', () => {
    it('Successfully loads PEF CZU home page', () => {
      // Ignore uncaught exceptions
      cy.on('uncaught:exception', (err, runnable) => {
        // return false to prevent the error from failing this test
        if (err.message.includes('gtag is not defined')) {
          return false;
        }
      });

      cy.visit('https://www.pef.czu.cz/');
      cy.get('#nav > ul > :nth-child(6) > a').click();
      cy.get(':nth-child(1) > a > .lazy').click();
      cy.wait(2000);
      cy.get('.lb-next').click();
      cy.wait(2000);
      cy.get('.lb-next').click();
      cy.wait(2000);
      cy.get('.lb-next').click();
      cy.wait(2000);
    });
  });
```

Kód scénářů pro lokální server:

```
describe('Visit Home Prauge Website', () => {
  it('Successfully loads home page', () => {
    // Ignore uncaught exceptions
    cy.on('uncaught:exception', (err, runnable) => {
      // return false to prevent the error from failing this test
      if (err.message.includes('gtag is not defined')) {
        return false;
      }
    });
```

```
    cy.visit('http://homeprague/');
    cy.get('[href="about.php"]').click();
    cy.get('#name').type("Vasily");
    cy.get('#surname').type("Gusev");
    cy.get('#email').type("gusevvas@mail.com");
    cy.get('#phone').type("+420666666666");
    cy.get('#message').type("Dobrý den, \nzaujala mi Vaše nabídka bytu na
Václavském náměstí"+
    "\nA proto bych chtěl se domluvit o prohlídce"+
    "\nProsím o zavolání nebo nepsání emailu.\nDěkuji.\nVasily Gusev.");
    cy.get('#submitButton').click();
    cy.get('.modal-content > p').should('have.text','Děkujeme Vám za odeslání
nabídky, brzy Vás budeme kontaktovat.');
    cy.get('.modal-content > button').click();
  });
});

describe('Visit Home Prauge Website - check search', () => {
  it('Successfully loads  home page', () => {
    // Ignore uncaught exceptions
    cy.on('uncaught:exception', (err, runnable) => {
      // return false to prevent the error from failing this test
      if (err.message.includes('gtag is not defined')) {
        return false;
      }
    });

    cy.visit('http://homeprague/');
    cy.get('[href="search.php"]').click();
    cy.get('.dispositions > :nth-child(1) > :nth-child(3) > input').click();
    cy.get('.dispositions > :nth-child(2) > :nth-child(1)').click();
    cy.get('.dispositions > :nth-child(2) > :nth-child(3)').click();
    cy.get('.localities > :nth-child(1) > :nth-child(2)').click();
    cy.get(':nth-child(1) > .property-info >
.description').should('have.text','Pronájem bytu 1+kk');
    cy.get(':nth-child(2) > .property-info >
.description').should('have.text','Prodej bytu 3+kk');
  });
});

describe('Check the offer on Home Prauge Website', () => {
  it('Successfully loads home page', () => {
    // Ignore uncaught exceptions
```

```
    cy.on('uncaught:exception', (err, runnable) => {
      // return false to prevent the error from failing this test
      if (err.message.includes('gtag is not defined')) {
        return false;
      }
    });

    cy.visit('http://homeprague/');
    cy.get('[href="detail.php?property=1"] > .property-item > .property-image
> img').click();
    cy.wait(2000);
    cy.get('[src="properties/1/2.png"]').click();
    cy.wait(2000);
    cy.get('[src="properties/1/3.png"]')
    cy.wait(2000);
    cy.get('[src="properties/1/4.png"]');
    cy.wait(2000);
    cy.get('.property-description > :nth-
child(2)').should("have.text",'\nTento byt 1+kk na Lázeňské ulici v Praze 1,
na Malé Straně, představuje skutečnou oázu klidu a luxusu. Obývací prostor je
navržen s otevřeným konceptem, spojujícím spánek, odpočinek a pracovní
činnosti v harmonickém celku. Vyznačuje se moderním designem, jemnými
tkaninami a teplými neutrálními tóny, které vytvářejí pocit klidu a pohody.
Ložnice je vybavena velkou a pohodlnou postelí, ideální pro odpočinek po
dlouhém dni. Dominantní okno od podlahy až ke stropu nabízí dechberoucí výhled
na noční oblohu a panorama města.');
  });
});
```

Tabulka 1 Výsledky pro vzdálený server – Cypress

	Scénář 1	Scénář 2	Scénář 3	Čas celkem
Čas (s)	7	8	11	26

Celkový čas, který Cypress potřeboval pro zpracování scénářů na vzdáleným serveru je 26 sekund.

Tabulka 2 Výsledky pro lokální server – Cypress

	Scénář 1	Scénář 2	Scénář 3	Čas celkem
Čas (s)	5	3	9	17

PlayWright

Pro spuštění PlayWrightu je také zapotřebí mít instalovaný Node.js, a proto byl použit stejný baliček verze v20.11.1 jako i pro Cypress. Scénáře byly provedeny PlayWrightem verze 1.42.1.

Tabulka 3 Výsledky pro vzdálený server – PlayWright

	Scénář 1	Scénář 2	Scénář 3	Čas celkem
Čas (s)	6	6.4	11,3	23,7

Tabulka 4 Výsledky pro lokální server – PlayWright

	Scénář 1	Scénář 2	Scénář 3	Čas celkem
Čas (s)	1.2	1	9,1	12,3

Scénáře se exekvovaly celkem 23.7 sekundy. Ve scénáři č 2. PlayWright nenabodoval psaní na klávesnici do polí, ale rovnou vkládal hodnoty do polí.

Selenium

Pro napsání testu byla použita Java 11 verze. Vývoj jednotlivých testů probíhal v Intellij Idea 2019.3.3. Jednotlivé dependence se řešily pomocí Mavenu. Verze frameworku je selenium-java-4.18.1 Čas pro framework se počítal bez inicializací Web driveru.

Tabulka 5 Výsledky pro vzdálený server – Selenium

	Scénář 1	Scénář 2	Scénář 3	Čas celkem
Čas (s)	7	9	13	29

Tabulka 6 Výsledky pro lokální server – Selenium

	Scénář 1	Scénář 2	Scénář 3	Čas celkem
Čas (s)	6	3	9	18

4.3.2 Experiment 1 Porovnání výkonnosti a efektivity – Back-End a API

Pro tento experiment budou provolány tři požadavky na https://dummyjson.com/ a localhost:8000, kde běží lokální server. Bude zaznamenán celkový čas potřebný k provolání všech tří požadavků. Testovací scénáře se provádějí ve Playwrightu, Cypressu, Postmanu a Jmetru.

Scénáře pro vzdálený server

Scénář 1 – Volání metody Login:

1. Provolej POST metodu **Login -** https://dummyjson.com/auth/login
2. Ověř, že se vrátil:
    - status code 200
    - objekt username
    - objekt email
    - objekt token

Scénář 2 – Volání metody getProducts:

1. Provolej GET metodu **getProducts** - https://dummyjson.com/products/1
2. Ověř, že se vrátil:
    - status code 200
    - objekt id
    - objekt title
    - objekt images

Scénář 3 – Volání metody getCarts:

1. Provolej GET metodu **getCarts -** https://dummyjson.com/carts
2. Ověř, že se vrátil:
    - status code 200
    - objekt total
    - objekt limit

Scénáře pro lokální server:

Scénář 1 – Volání metody UpdateIngredient:

1. Provolej PUT metodu **UpdateIngredient –** localhost:8000/ingredient/update
2. Ověř, že se vrátil:
    - Status code 200
    - Objekt name
    - Objekt id

Scénář 2 – Volání metody getRecipeList:

1. Provolej PUT metodu **getRecipeList–** localhost:8000/recipe/list
2. Ověř, že se vrátil:
    - Status code 200

Scénář 1 – Volání metody createRecipe:

1. Provolej PUT metodu **createRecipe–** localhost:8000/recipe/create
2. Ověř, že se vrátil:
    - Status code 200
    - Objekt name
    - Objekt id
    - Objekt categoryIdList
    - Objekt ingredientList
    - Objekt imageId

Pro přesnost volání bude zopakováno 5krát u každého softwaru. Bude zvolen nejlepší výsledek.

4.3.3 Experiment 2 Podpora různých typů testů

Postman

Jak již bylo ukázáno v experimentu číslo 1, Postman je schopen vykonávat **funkční testy a unit testy**, ověřovat, že aplikace nebo software odpovídá požadovaným specifikacím. Příklad:

```
// Validate that the response code should be 200
pm.test("Status code is 200", function () {
    pm.response.to.have.status(200);
});

// Validate that the response has an id object
pm.test("Response to have 'id' object", function () {
    pm.expect(pm.response.json()).to.have.property('id');
});
```

Ačkoli Postman není primárně určen jako nástroj pro **testování výkonnosti**, kterou nabízejí takové specializované nástroje jako například JMeter, může být použit k základnímu hodnocení rychlostí API. Například bude kontrolovat zpracování požadavků za méně než 300 ms.

```
pm.test("Response time is less than 300ms", function () {
    pm.expect(pm.response.responseTime).to.be.below(300);
});
```

Postman také nabízí testy pro **testování bezpečností** API. Testuje se zranitelnost a autorizace jako u tohoto příkladu:

```
pm.test("Unauthorized request", function () {
    pm.response.to.have.status(401);
});
```

Testuje se, jestli se na požadavek vrátí stav 401 – Unauthorized.

Dále Postman umožňuje kombinovat více API volání do jediného testovacího workflow. Tímto způsobem se dá otestovat **integrace** mezi různými komponenty systému nebo mezi různými mikroslužbami. Například, kdyby první volání bylo typu POST / user pro vytvoření nového uživatele a druhé volání by bylo GET / e-mail pro získání informace, jestli potvrzovací e-mail byl odeslán novému uživateli. Potom skript pro kontrolu by mohl vypadat následovně:

```
pm.test("Email was sent to the user", function () {
    var jsonData = pm.response.json();
    pm.expect(jsonData).to.some(function (email) {
        return email.to === pm.variables.get("newUserEmail");
    });
});
```

Z prvního experimentu také vyplývá, že Postman je také schopen provádět **regresní testování** API a Back-endu. Nechť existuje kolekce metod, která se používá pro funkční testy API. Tato kolekce se dá pouštět po každém nasazení, pro ověření, že s novými změnami aplikace se nic dalšího nerozbilo. Spolu s tím by šlo dělat i **smoke testy**. Stejným způsobem se dají provádět i **E2E testy**.

V případě, kdy API nebo Back-End není dostupný, Postman je schopen provádět tvorbu mock serveru. Tento server bude simulovat API odpovědi bez potřeby skutečného back-endu. Dělá se to tak, že na mock serveru budou definovány nějaké API požadavky a k nim budou přidruženy odpovědi, které by měly být vráceny.

Například, mock server bude simulovat požadavky odpovědí na endpoint GET /products. Tomuto způsobu testování se říká **mock testování**.

JMeter

Z popisu dokumentace se dá soudit, že aplikace JMeter slouží k **testování výkonnosti** na různých back-end komponentách softwarového systému. Tento typ testování zahrnuje:

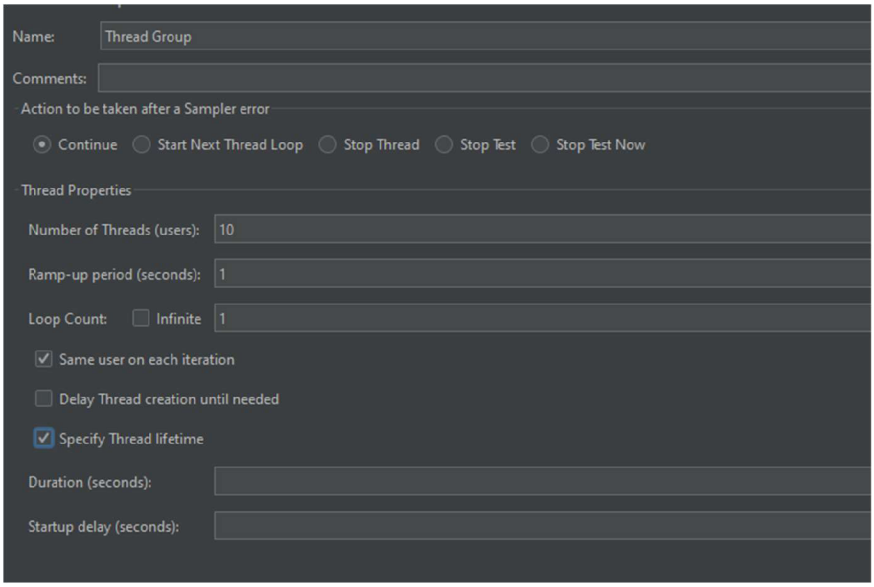
- **zátěžové testování**, které se zaměřuje na simulaci reálných zátěžových podmínek;
- **tressové testování,** které je určeno pro zjištění limitu testovaného systému

V některých případech se dá testovat i front-end pro client-side performance.

Jak bylo zjištěno z experimentu č. 1, pomocí tohoto softwaru lze provádět **funkční testy** aplikace. Ale tento software je více určen pro provádění zátěže na aplikaci.

JMeter dovoluje uživateli nastavit vlákna, ve kterých se tvoří jednotlivá volání do aplikace. V každém vlákně se dá nastavit počet uživatelů, kteří budou daný endpoint provolávat. Dá se také nastavit perioda, počet volání, počet cyklů a nastavit dobu „života“ vlákna.

Obrázek 8 JMeter – Nastavení vlákna



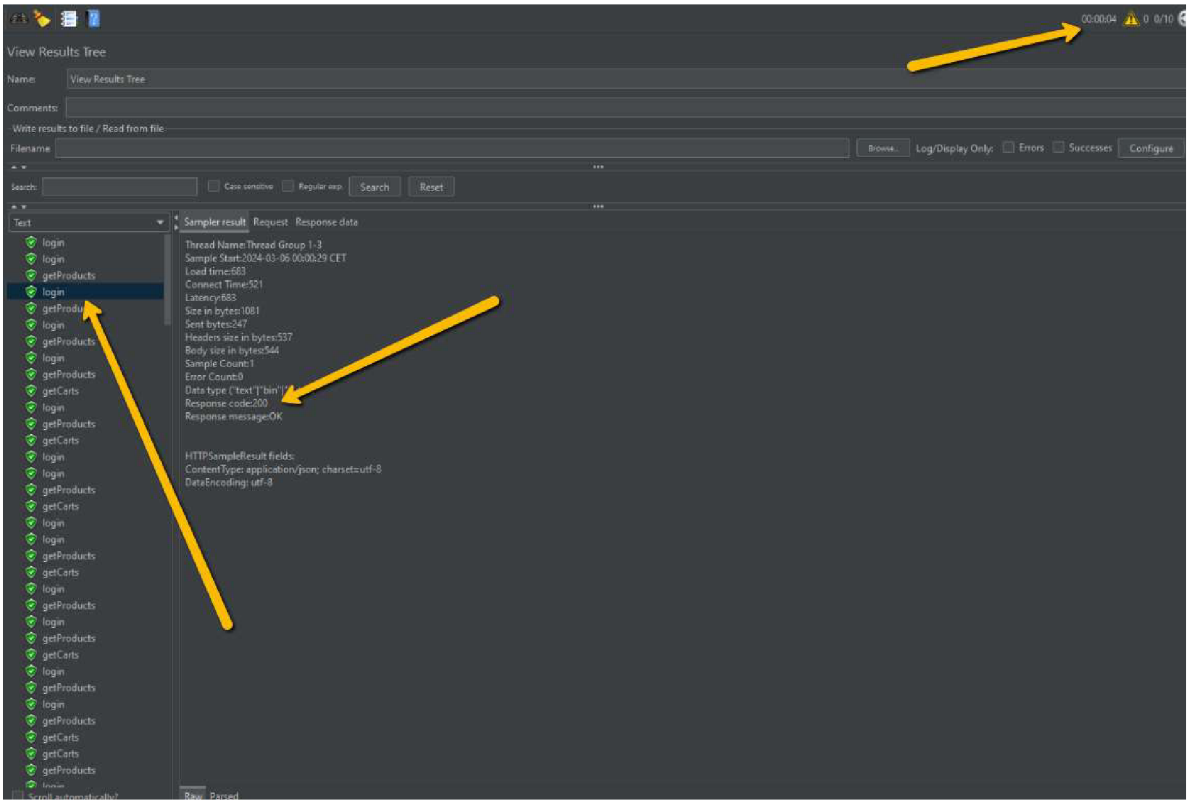
Zdroj: vlastní zpracování, JMeter

Nechť budou provolány scénáře z experimentu č. 1 s následujícími parametry:

- Počet uživatelů –10
- Čas nárůstu zátěže – 1 sekunda
- Počet cyklů – nekonečno
- Doba tvorby požadavků (Duration of thread lifetime) – 60 sekund

JMeter začne zátěžový test serveru. Bude vytvořena zátěž z 10 uživatelů. Každý call bude svítit zeleně v případě úspěšného volání a červeně v případě nějaké chyby.

Obrázek 9 JMeter – Zátěžový test



Zdroj: vlastní zpracování, JMeter

Cypress

Z prvního experimentu bylo ukázáno, že Cypress umí otestovat jak API, tak i Front-end. A proto jednotlivé typy testů budou rozděleny podle komponent.

Back-end a API:

Z pohledu Back-Endu, jak již bylo ukázáno, je Cypress schopen testovat jednotlivé HTTP požadavky. U těchto požadavku je možné ověřit kód odpovědí a tímto způsobem je možné zahajovat **unit testy**, **E2E testy** a **bezpečnostní testy**. Příklad E2E scénáře:

1. Pošli POST požadavek na vytvoření receptu
2. Pošli GET požadavek pro získaní receptu
3. Pošli PUT požadavek pro aktualizaci receptu
4. Pošli DELETE požadavek pro odstranění vytvořeného receptu

Bezpečnostní testy by bylo možné provést následujícím způsobem:

1. Pokusit se přistoupit k chráněnému endpointu bez poskytnutí autentizačního tokenu.
    - Ověřit, že server vrátí HTTP stavový kód 401 (Unauthorized) indikující, že požadavek byl zamítnut kvůli chybějící autentizaci.

V případě, že pro aplikaci není hotové API nebo je nedostupné, framework Cypress dovoluje provést tzv. **mock testování**, pomocí kterého se dá simulovat chování endpointu. Díky tomu je možné specifikovat očekávané chování mock objektů jako například, jaké metody budou volány a jaké hodnoty mají vracet. Příklad mock objektu login:

```
cy.intercept('POST', '/login', {
  statusCode: 200,
  body: {
    token: 'fake-bearer-token',
    user: {
      id: 1,
      name: 'Vasily Gusev',
      email: 'vasily@mail.com'
    }
  }
}).as('loginRequest');
```

Nechť existuje aplikace s blogem, kde uživatelé si mohou číst články a přidávat k nim komentáře. Je zapotřebí se ujistit, že když uživatel přidá komentář k článku, komentář se správně uloží na server a poté se zobrazí na stránce článku. V tomto scénáři je zapotřebí provést **integrační testování.** V Cypressu se uloží id článku a zavolá se stránka článku s tímto ID:

```
    cy.visit(`/articles/${articleId}`);
```

Dále se na stránce přidá nějaký komentář:

```
    cy.get('textarea[name="comment"]').type(commentText);
    cy.get('form').submit();
```

Následně se ověří, že server správně komentář uložil:

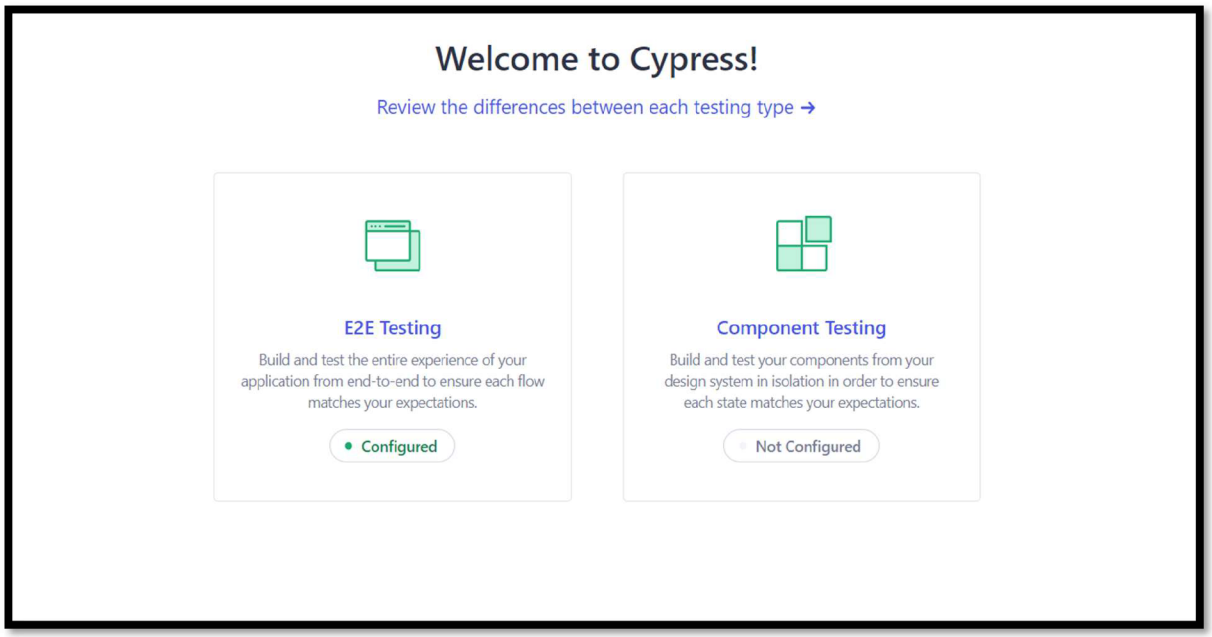
```
    cy.request(`/api/articles/${articleId}/comments`).then((response) => {
      expect(response.body).to.deep.include({ content: commentText });
```

Front-End:

Cypress je framework primárně dělaný pro testování front-endu. A proto zvládá i jednoduché testování **jednotlivých komponent** a **smoke testy**. Případ: ověření funkčností tlačítka, jak bylo ukázáno v prvním experimentu.

V prvním experimentu také proběhl proces na objednávku knihy na webu. Z toho vyplývá, že Cypress zvládá i testování **End-to-end** procesu od začátku a do konce. Oba typy testu se nabízejí na úvodní obrazovce Cypressu.

Obrázek 10 Cypress – úvodní obrazovka



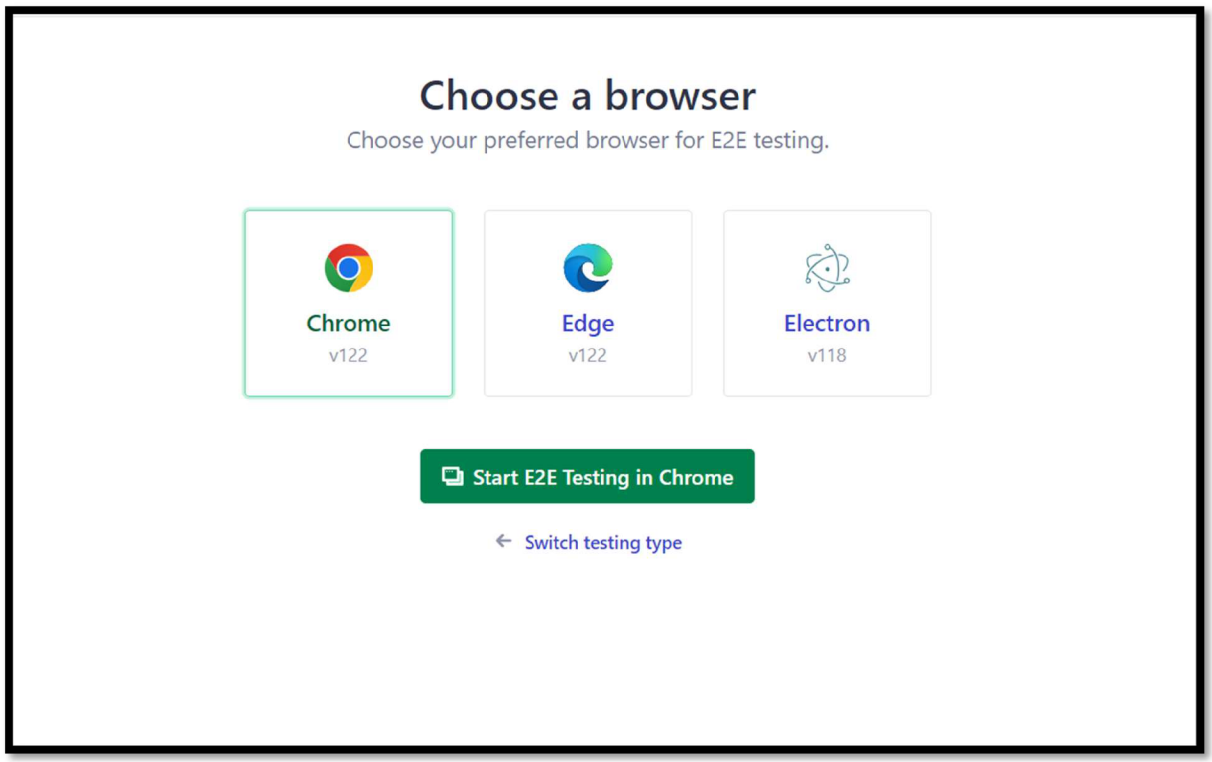
Zdroj: vlastní zpracování, Cypress

Při vyplnění a doplnění určitých scénářů aplikace je možné také docílit **regresních testů** v případě, kdyby bylo zapotřebí otestovat N scénářů s různými kombinacemi dat a průchodů.

Jakákoli webová aplikace se v jiném prostředí nebo prohlížeči může chovat jinak. Může se také stát, že nějaké komponenty budou mít jiný vzhled. Proto je zapotřebí provádět **testování kompatibility** s jinými prohlížeči. Framework Cypress po volbě komponenty zobrazuje nabídku s prohlížeči. Testy lze provádět v:

1. Google Chromu
2. Mozilla Firefoxu
3. Microsoft Edge
4. Electron

Obrázek 11 Cypress – Volba prohlížeče



Zdroj: vlastní zpracování, Cypress

Playwright

Podobně jako Cypress framework Playwright zvládá otestovat Front-end, API a Back-End. Rozdíly mezi těmito frameworky spočívají v technických detailech implementace a podpory různých prohlížečů, zatímco v základních testovacích metodikách a cílech jsou si velmi podobné. To znamená, že jakékoliv typy testů, které lze provést v Cypressu, je možné provést také v Playwrightu.

Selenium

Selenium je framework, který se používá primárně pro testování Front-Endu. Z testovacích scénářů výše bylo ukázáno, že se dají provádět **unit testy, regresní testy** a **E2E testy.** Z pohledu regresních testů se také dají nastavit různé průchody aplikací.

V Seleniu se dá nastavit browser, ve kterém bude probíhat nějaký testovací scénář. Tímto se testuje **kompatibilita** s jinými prohlížeči. Ukázka v programovacím jazyce Java:

```
System.setProperty("webdriver.gecko.driver", "/path/to/geckodriver");
WebDriver driver = new FirefoxDriver();
```

```
System.setProperty("webdriver.edge.driver", "/path/to/msedgedriver");
WebDriver driver = new EdgeDriver();
```

Selenium je také schopen provádět **smoke testy.** Příklad scénáře: proběhne přihlášení na přihlašovací stránku a poté se zkontroluje přítomnost prvku na dashboardu pro potvrzení úspěšného přihlášení.

4.3.4 Experiment 3 – Znalost programovacích jazyků

Pro testování s vybranými nástroji je zapotřebí mít znalost určitých programovacích jazyků. Bude zjištěno, jaký nástroj je pružnější z ohledu podpory programovacích jazyků. Tento experiment předpokládá, že tester, který bude s nástrojem pracovat, rozumí technologiím autorizace pro provolání API, RESTful a SOAP API, hlavičkám API požadavků, HTML, CSS, XML a HTTP stavům.

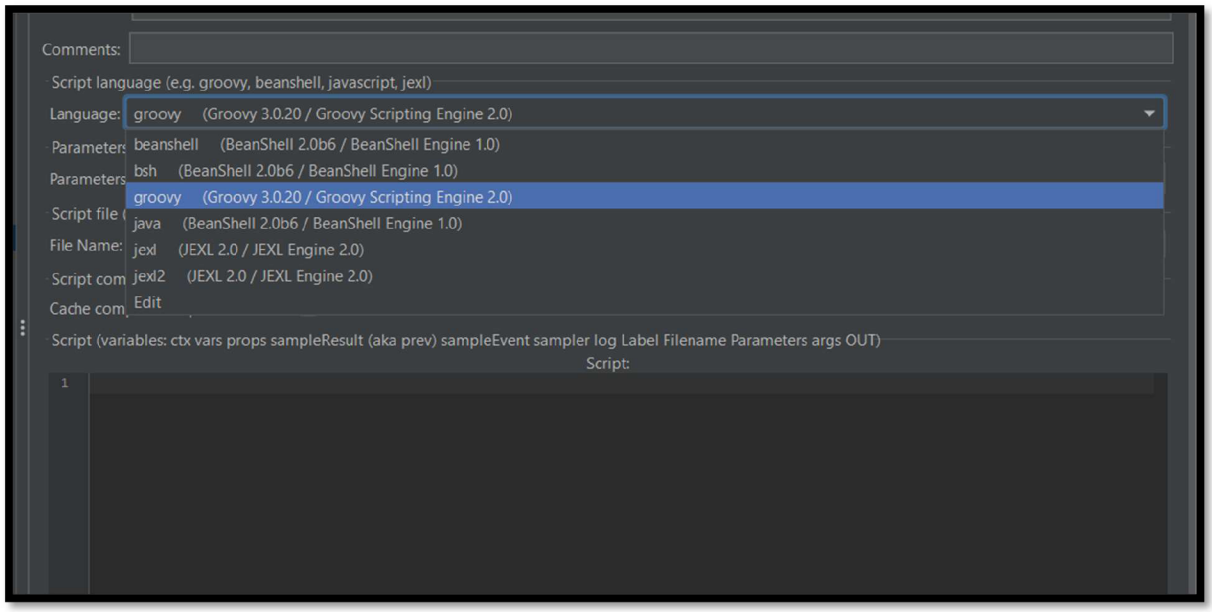
Postman

Postman nabízí vestavěné prostředí pro psaní testů pro API požadavky. Testy se píšou zvlášť ke každému volání v záložce „Tests“. Všechny testy se píšou ve programovacím jazyce **JavaScript**. Příklad je uveden v prvním experimentu.

JMeter

The Apache JMeter je aplikace, která je napsána ve programovacím jazyce Java. V tomto jazyce tester má možnost přizpůsobit funkcionalitu JMetru v případě potřeby. Samotné testy se píšou v uživatelském rozhraní aplikaci. V nástroji JSR223 Listner, který se používá k získávání a zpracování informací o odpovědích a výsledcích testů v průběhu nebo na konci testovacích běhů, je nabídka skriptovacích jazyků, ve kterých se dají napsat testy.

Obrázek 12 JMeter – Nabídka skriptovacích jazyků



Zdroj: vlastní zpracování, JMeter

Z obrázku je vidět, že testy se dají napsat v jazyce **Beanshell**, **Groovy**, **Java**, **Jexl**. Následně se testy dají psát také v **JavaScriptu**

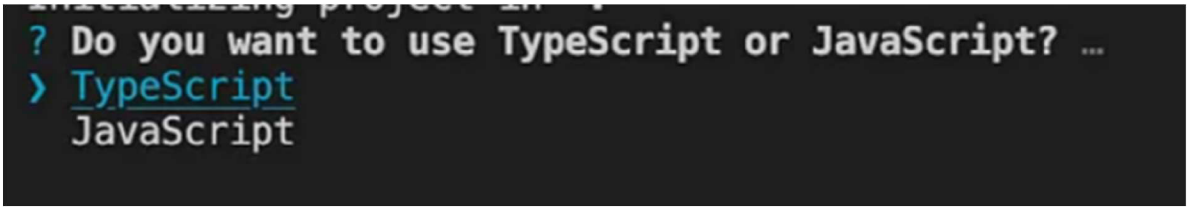
Cypress

Pro zprovoznění Cypressu je zapotřebí mít nainstalovaný npm (Node Packege Manager) a Node.js. NPM je správcem balíčků pro jazyk **JavaScript**. Testy v Cypressu se píšou jenom v tomto programovacím jazyce.

Playwright

PlayWright stejně jako Cypress používá knihovny Node.js a npm. Při instalaci PlayWrightu bude upřesněno, jaký jazyk chce uživatel použít při psaní testů.

Obrázek 13 PlayWright – Volba programovacího jazyku



Zdroj: vlastní zpracování, Visual Code Studi, PlayWright

Testy se mohou psát buď v TypeScriptu nebo v JavaScriptu.

Selenium

Selenium je jedním z frameworků, který má nejvíce podporovaných jazyků. Příklady pro tuto diplomovou práci byly vypracovány v Seleniu v programovacím jazyce **Java**. Tento framework podle oficiálních stránek má pět podporovaných programovacích jazyků. K nim patří: (zdroj: „Selenium: Downloads.“ Selenium, dostupné z https://www.selenium.dev/downloads/.)

- C#
- Ruby
- JavaScript
- Python
- Java

4.3.5 Experiment 4 – Porovnání přesnosti a spolehlivosti detekce chyb

Tento experiment se zabývá detekcemi chyb ve vybraných softwarech a frameworcích. Bude vymyšlen scénář, který záměrně bude vést ke špatnému výsledku a bude se posuzovat, jak nástroje detekují chyby. Scénáře budou zvlášť jak pro Fornt-End tak i pro API Back-end.

Scénáře pro FE:

Scénář 1 – Článek o parkování:

1. Navštiv https://www.pef.czu.cz/
2. Klikni na **Studium**
3. Klikni na **Informace pro studenty**
4. Klikni na **Parkování v areálu ČZU**
5. Zkontroluj název článku „Pkování v areálu ČZU“

Očekávaný výsledek: „Pkování v areálu ČZU“

Skutečný výsledek: „Pakování v areálu ČZU“

Scénář 2 – Článek o parkování:

1 Navštiv https://www.pef.czu.cz/

2 Klikni na **O fakultě**

3 Klikni na **Odkaz na současného děkana** (doc. Ing. Tomáš Šubrt, Ph.D.)

4 V nově otevřeném okně zkontroluj jméno **doc. Ing. Tomáš Šubrt, Ph.D. (1325)**

Očekávaný výsledek: Odkaz na současného děkana (doc. Ing. Tomáš Šubrt, Ph.D.)

Skutečný výsledek: Objekt s odkazem nenalezen

Scénář pro BE a API:

Scénář 1 – Volání metody Login:

1. Provolej POST metodu **Login -** https://dummyjson.com/auth/login
2. Ověř, že se vrátil:
   - status code 400
   - objekt usrname
   - objekt e-mail
   - objekt token

Očekávaný výsledek:
- status code 400
- objekt usrname
- objekt e-mail
- objekt token

Skutečný výsledek:
- status code 200
- objekt username
- objekt email
- objekt token

5 Výsledky a diskuse

5.1 Výsledky – Front-end – Experiment 1

Tabulka 7 Výsledky pro vzdálený server – Front-end

Aplikace/Framework	Cypress	Playwrigth	Selenium
Čas (s)	26	23,7	29

U testů na vzdáleném serveru nejlepšího výsledku 23.7 sekund dosáhl Playwright.

Tabulka 8 Výsledky pro lokální server – Front-end

Aplikace/Framework	Cypress	Playwrigth	Selenium
Čas (s)	17	12,3	18

U testů na lokálním serveru nejlepšího výsledku 12.3 sekund dosáhl také Playwright.

Nejrychlejších výsledů Playwright dosáhl tím, že místo napodobování psaní textu z klávesnice prováděl rovnou vložení textových hodnot do polí. Navíc po načtení stránky počká na existenci tlačítka a rovnou na to kliká. Texty se mohou spouštět paralelně v několika vláknech, což má také dopad na rychlost. Pokud tester musí otestovat čtyři scénáře, tak pro ušetření času u těchto testů může spustit čtyři scénáře najednou. Cypress má podobnou funkci v placené verzi.

5.2 Výsledky – Back-end a API – Experiment 1

Tabulka 9 Výsledky pro vzdálený server – Back-End a API

Aplikace/Framework	Postman	JMeter	Cypress	Playwright
Čas (ms)	993	1171	625	1710

Po provedení scénářů na vzdáleném serveru v Postmanu, Jmetru, Playwrightu a Cypressu nejlepších výsledků dosáhl framework Cypress.

Tabulka 10 Výsledky pro lokální server – Back-End a API

Aplikace/Framework	Postman	JMeter	Cypress	Playwright
Čas (ms)	177	155	217	226

Po provedení scénářů na vzdáleném serveru v Postmanu, Jmetru, Playwrightu a Cypressu nejlepších výsledků dosáhl framework JMeter.

5.3 Výsledky – Experiment 2

Tabulka 11 Výsledky – podporované typy testů – Back-end a API

Software / Framework	Podporované typy testů
Postman	• Unit testy • Výkonnostní testy • Bezpečnostní testy • Regresní testy • Smoke testy • Mock testy • E2E testy • Integrační testy
JMeter	• Funkční testy • Výkonnostní testy
Cypress	• Unit testy • Integrační testy • E2E testy • Mock testy • Bezpečnostní testy
Playwright	• Unit testy • Integrační testy • E2E testy • Mock testy • Bezpečnostní testy

Tabulka 12 Výsledky – podporované typy testů – Front-end

Software / Framework	Podporované typy testů
Cypress	• E2E testy • Regresní testy • Unit testy • Testování kompatibility
Playwright	• E2E testy • Regresní testy • Unit testy • Testování kompatibility
Selenium	• E2E testy • Regresní testy • Unit testy • Testování kompatibility • Smoke testy

5.4 Výsledky – Experiment 3

Tabulka 13 Výsledky – znalost programovacích jazyků

	Postman	JMeter	Cypress	PlayWright	Selenium
Programovací jazyky	JavaScript	JavaScript Beanshell Groovy Java Jexl	JavaScript	JavaScript TypeScript	JavaScript Python Ruby C# Java

5.5 Výsledky – Experiment 4

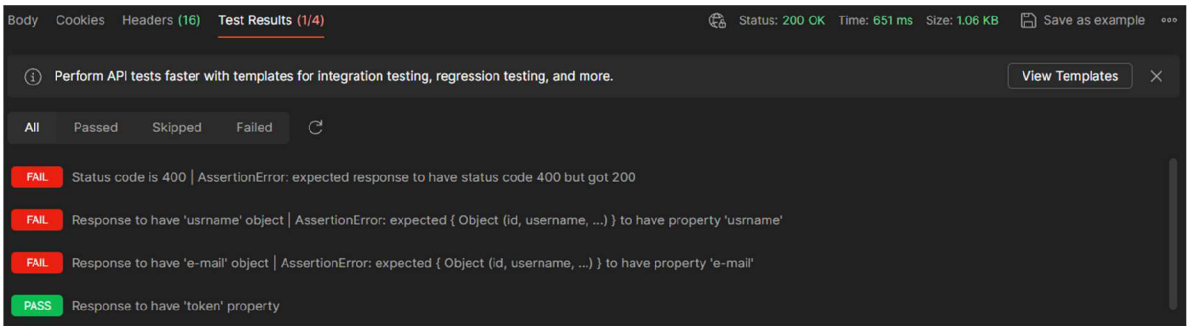
5.5.1 Postman

Pro ověření nového scénáře je zapotřebí vytvořit novou kolekci, okopírovat volání a přepsat existující testy. Přepsané testy budou vypadat následovně:

```
// Validate that the response code should be 400
pm.test("Status code is 400", function () {
    pm.response.to.have.status(400);
});

// Validate that the response has an usrname object
pm.test("Response to have 'usrname' object", function () {
    pm.expect(pm.response.json()).to.have.property('usrname');
});

// Validate that the response has e-mail object
pm.test("Response to have 'e-mail' object", function () {
    pm.expect(pm.response.json()).to.have.property('e-mail');
});

// Validate that the response has token property
pm.test("Response to have 'token' property", function () {
    pm.expect(pm.response.json()).to.have.property('token');
});
```

Po spuštění volání spadly tři testy, protože se očekával jiný výsledek, který je vidět na následujícím obrázku:

Obrázek 14 Postman – reportování špatných výsledků



Zdroj: vlastní zpracování, Postman

Z obrázku je vidět, že pokud některé z testů nedopadly dobře, neboť nebyly úspěšné, tak v záložce „Test Results“ bude uvedeno, kolik prošlo. Po rozkliknutí záložky budou zobrazeny výsledky jednotlivých testů. Neúspěšné testy mají stav „FAIL“. U každého z nich je napsán očekáváný a skutečný výsledek. Příklad: „Response to have 'usrname' object | AssertionError: expected { Object (id, username, ...) } to have property 'usrname'“

Při větším počtu testů pomáhají lépe se orientovat záložky Passed (Úspěšné), Skipped (Vynechané) a Failed (Neúspěšné). Po kliknutí na záložku se vyfiltrují testy se zvoleným stavem.

Jsou-li nastavené automatické regrese nebo end-to-end testy po nasazení u nějakého testovaného systému, tak potom tento způsob reportování pomáhá testerovi rychle zjistit, kde přesně je problém v testovaném systému. Následně daný problém může být reportován vývojářům pro rychlé vyřešení.

5.5.2 JMeter

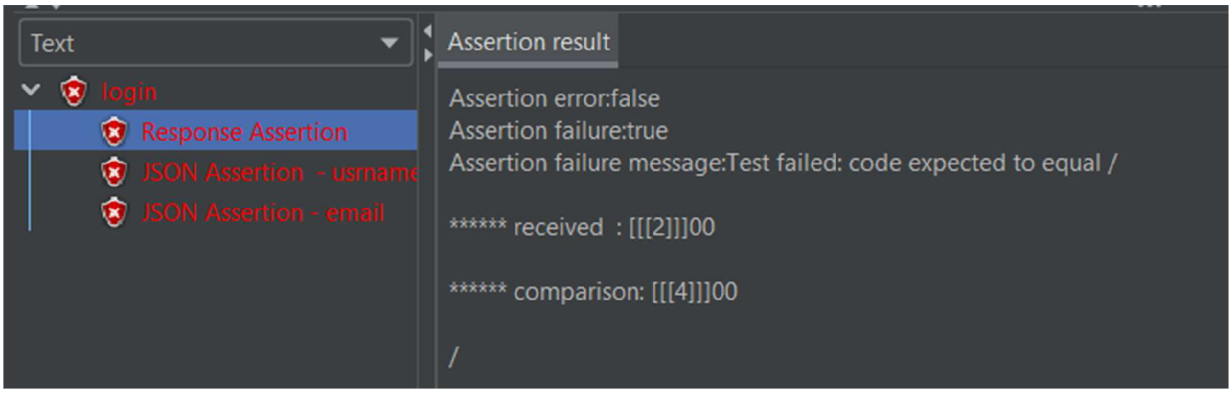
Pro ověření nového scénáře bylo vytvořeno nové vlákno. Následně byly upraveny testy a kontroly. Po volání požadavku výsledek vypadá následovně:

Obrázek 15 JMeter – Výsledek volání

Zdroj: vlastní zpracování, JMeter

Testy, které dopadly špatně, budou svítit červeně. Celkové volání se počítá jako nevalidní, pokud je jeden z testů neúspěšný. Po rozkliknutí testu na http stav, který byl neúspěšný, bude podobně jako u Postmanu napsán očekáváný a skutečný výsledek. Testy, které dopadly dobře, zobrazeny nebudou.

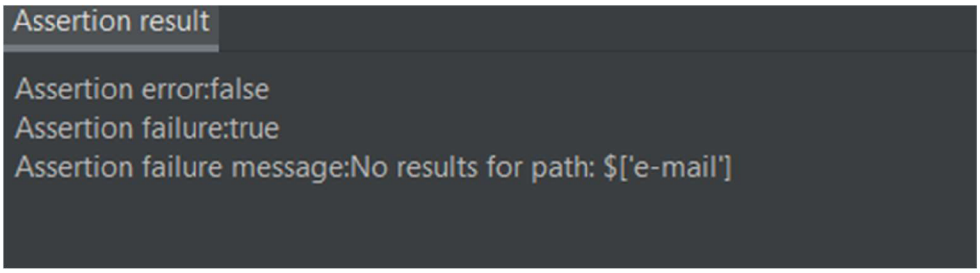
Obrázek 16 JMeter – Očekáváný a skutečný výsledek



Zdroj: vlastní zpracování, JMeter

U testu na přítomnost objektu, bude napsána hláška, že nebyly nalezeny výsledky pro objekt „e-mail“. Příklad:

Obrázek 17 JMeter – Assertion Failure



Zdroj: vlastní zpracování, JMeter

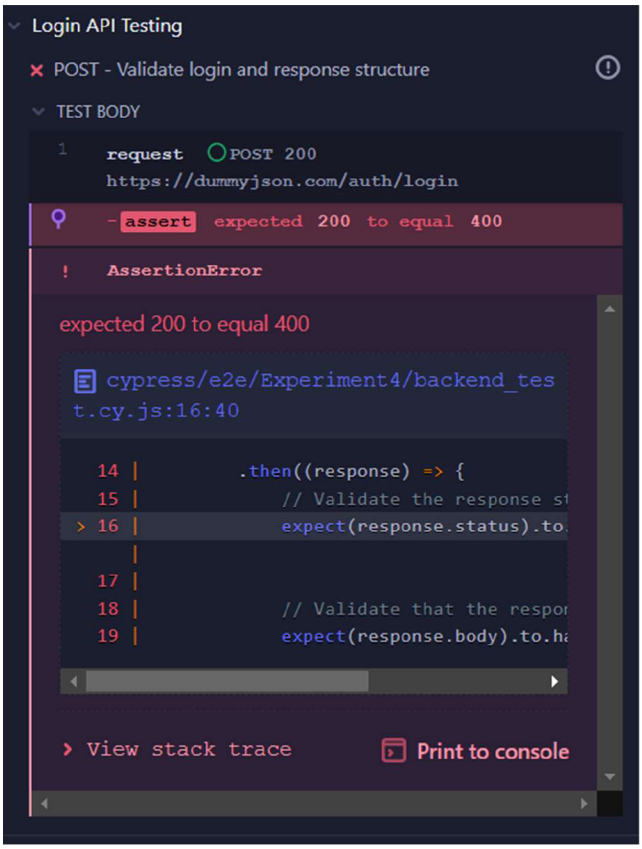
5.5.3 Cypress

Pro exekuci testů v Cypressu pro obě komponenty byly vytvořeny soubory zvlášť pro front-end a API. Testy byly upravené na základě definovaných scénářů.

BE a API:

Při spuštění testu na API Cypress detekoval, nápodobně jako Postman a JMeter, že test nedopadl dobře, ale přitom nepokračoval v dalších testech. Dokud se nevyřeší první test, tak nepokračuje k druhému.

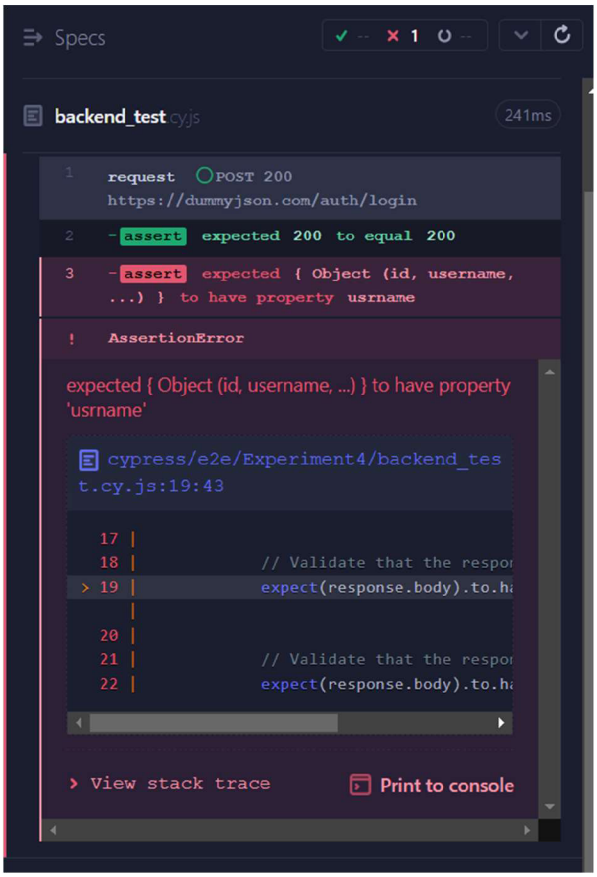
Obrázek 18 Cypress – API Test – Reportování chyby



Zdroj: vlastní zpracování, Cypress

Z obrázku je vidět, že je vypsán očekáváný a skutečný výsledek, ale na další testy nepokračuje, dokud se neupraví řádek 16. Když se upraví test na HTTP stav, tak ten test projde a označí se jako úspěšný, ale znovu se zastaví na neúspěšném testu.

Obrázek 19 Cypress – API Test – Reportování chyby po úpravě



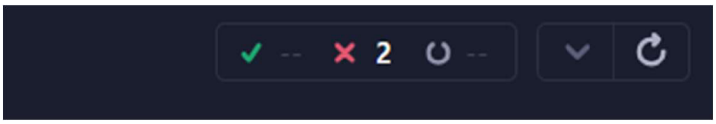
Zdroj: vlastní zpracování, Cypress

Cypress je framework určený více pro testování front-endu, a proto při první nesrovnalosti hned zastavuje testování, což by se pro API testy moc nehodilo.

FE:

Po provedení testu scénářů byly dosaženy dva neúspěšné výsledky. Scénáře se prováděly postupně bez zastavení. Uživatelské rozhraní Cypressu zobrazuje neúspěšné testy následujícím způsobem:

Obrázek 20 Cypress – Stav testů



Zdroj: vlastní zpracování, Cypress

U fajfky je zobrazen počet úspěšných testů, u křížku – neúspěšné. Během exekuce testů se provádí logování každého kroku, což potom může pomoci uživateli rychleji najít potřebný krok v kódu.

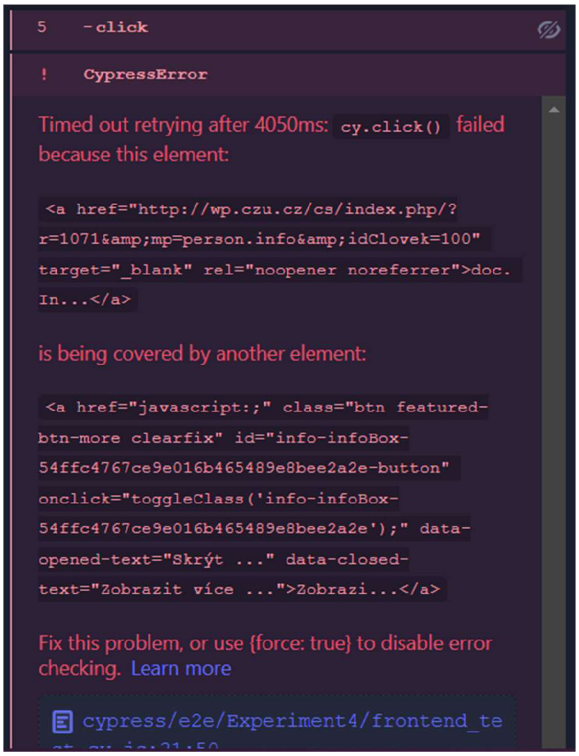
Po provedení prvního scénáře Cypress detekoval chybu ve tvaru: očekáváný výsledek a skutečný výsledek. Před vypsáním záporného výsledku testu se počkalo 4 sekundy pro případ, že definovaný element se objeví o trochu později. Následně je také zobrazen řádek kódu, u kterého test byl neúspěšný.

Obrázek 21 Cypress – FE test – Výsledek 1. scénáře

Zdroj: vlastní zpracování, Cypress

Při exekuci druhého scénáře test nedopadl úspěšně, protože hledaný element byl překryt jiným elementem. Kvůli tomu se na tento element nedalo kliknout. Nápodobně jako u prvního scénáře se počkalo 4 sekundy na objevení elementu, ale v tomto případě se čekalo na to, až tento element bude dostupný pro klikání.

Obrázek 22 Cypress – FE test – Výsledek 2. scénáře



Zdroj: vlastní zpracování, Cypress

Cypress vypisuje chyby hned do okna s aplikací. U testů píše, co přesně nedopadlo dobře, což by mohlo uživateli pomoci v hledání případných problémů.

5.5.4 PlayWright

Podobně jako pro Cypress byly připraveny dva soubory se zápornými testovacími scénáři pro front-end a back-end.

BE a API:

Nevalidní API test se označí křížem v konzoly. Obdobně jako u Cypressu framework Playwright se hned zastaví na prvním neúspěšném testu a dále nepokračuje. Do konzole je

napsána chyba, očekáváný a skutečný výsledek. Je označen řádek, u kterého se test zastavil. Příklad chyby:

```
Error: expect(received).toBe(expected) // Object.is equality

Expected: 400
Received: 200
```

Při špatném výsledku objektu „e-mail“ se chyba zapisuje do konzole s celou odpovědí na volání.

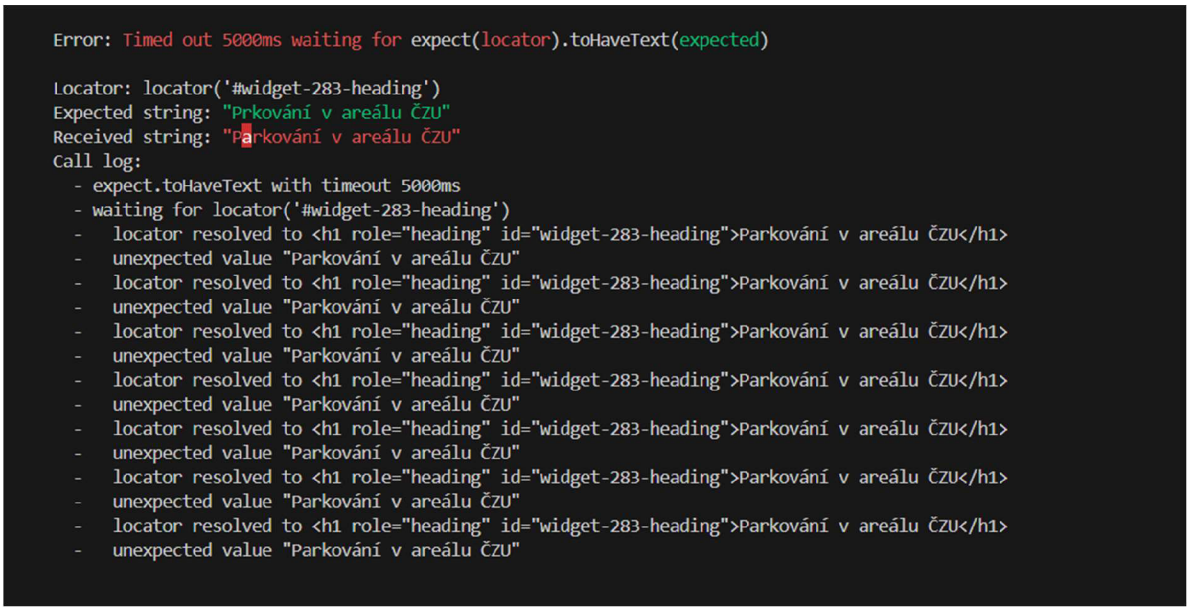
```
Expected path: "usrname"
Received path: []

Received value: {"email": "kminchelle@qq.com", "firstName": "Jeanne",
"gender": "female", "id": 15, "image":
"https://robohash.org/Jeanne.png?set=set4", "lastName": "Halvorson", "token":
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6MTUsInVzZXJuYW1lIjoia21pbmNoZWxs
ZSIsImVtYWlsIjoia21pbmNoZWxsZUBxcS5jb20iLCJmaXJzdE5hbWUiOiJKZWFubmUiLCJsYXN0Tm
FtZSI6IkhhbHZvcnNvbiIsImdlbmRlciI6ImZlbWFsZSIsImltYWdlIjoiaHR0cHM6Ly9yb2JvaGFz
aC5vcmcvSmVhbm5lLnBuZz9zZXQ9c2V0NCIsImlhdCI6MTcxMDg4MjM2OCwiZXhwIjoxNzEwODg1OT
Y4fQ.jm4Gey4BGUhqEeXreFGK7e0EwODg1OTY4fQ.jm4Gey4BGUhqEeXreFGK7e0n4mD3juAi3ScPr
T6dhOQ", "username": "kminchelle"}
```

FE:

Po exekuci front-end scénářů je každý krok zapsán do konzoly včetně veškerých opakování kroků. V případě, že očekávaný element nebyl nalezen, tak krok se opakuje několikrát V případě chyby se do konzoly vypisuje část kódu, kde se chyba stala.

Po exekuci prvního scénáře Playwright zobrazil chybu, že očekával hodnotu „Prkování v areálu ČZU“, ale dostal „Parkování v areálu ČZU“. Poté několikrát se pokusil o hledání nenalezeného elementu.

Obrázek 23 Playwright – FE test – výsledek 1. scénářů



```
Error: Timed out 5000ms waiting for expect(locator).toHaveText(expected)

Locator: locator('#widget-283-heading')
Expected string: "Prkování v areálu ČZU"
Received string: "Parkování v areálu ČZU"
Call log:
  - expect.toHaveText with timeout 5000ms
  - waiting for locator('#widget-283-heading')
  -   locator resolved to <h1 role="heading" id="widget-283-heading">Parkování v areálu ČZU</h1>
  -   unexpected value "Parkování v areálu ČZU"
  -   locator resolved to <h1 role="heading" id="widget-283-heading">Parkování v areálu ČZU</h1>
  -   unexpected value "Parkování v areálu ČZU"
  -   locator resolved to <h1 role="heading" id="widget-283-heading">Parkování v areálu ČZU</h1>
  -   unexpected value "Parkování v areálu ČZU"
  -   locator resolved to <h1 role="heading" id="widget-283-heading">Parkování v areálu ČZU</h1>
  -   unexpected value "Parkování v areálu ČZU"
  -   locator resolved to <h1 role="heading" id="widget-283-heading">Parkování v areálu ČZU</h1>
  -   unexpected value "Parkování v areálu ČZU"
  -   locator resolved to <h1 role="heading" id="widget-283-heading">Parkování v areálu ČZU</h1>
  -   unexpected value "Parkování v areálu ČZU"
  -   locator resolved to <h1 role="heading" id="widget-283-heading">Parkování v areálu ČZU</h1>
  -   unexpected value "Parkování v areálu ČZU"
```

Zdroj: vlastní zpracování, Cypress

U druhého testu Playwright se pokusil několikrát počkat na viditelnost elementu a projít webovou stránku. Poté napsal, že test je neúspěšný.

```
  - retrying click action, attempt #14
      -   waiting 500ms
      -   waiting for element to be visible, enabled and stable
      -   element is visible, enabled and stable
      -   scrolling into view if needed
      -   done scrolling
      -   <a href="javascript:;" data-opened-text="Skrýt ..."…>Zobrazit více
...</a> from <div class="featured-decor">…</div> subtree intercepts pointer
events
      - retrying click action, attempt #15
      -   waiting 500ms
```

Podrobné logování a opakované pokusy mohou zefektivnit testovací proces. V případě problému se tester může podívat do konzoly a dozvědět se, co se přesně nepovedlo a kde se program zasekl. To by mohlo pomoci v budoucí nápravě kódu nebo testovacího scénáře.

5.5.5 Selenium

Testovací scénáře byly napsány v Java Seleniu pro otestování. Stejně jako u Cypressu a u Playwrightu se jednotlivé kroky zapisují do konzole. Pokud nastane chyba, tak se také zapisuje do konzole a okýnko s Chromem se hned zavírá. Test svítí jako neúspěšný. Selenium čeká na element 30 sekund.

Po exekuci prvního scénáře do konzoly byla vypsaná chyba:

```
Exception in thread "main" org.junit.ComparisonFailure: expected:<P[]rkování v
areálu ČZU> but was:<P[a]rkování v areálu ČZU>
```

Takže stejně jako u dvou předchozích frameworku se vypisuje očekáváný a skutečný výsledek.

Po exekuci druhého scénáře Selenium počkal na viditelnost elementu 30 sekund, poté vypsal chybu do konzoly:

```
org.openqa.selenium.TimeoutException: Expected condition failed: waiting for
element to be clickable: By.cssSelector: :nth-child(15) > :nth-child(3) > a (tried
for 30 second(s) with 500 milliseconds interval)
```

S ohledem na to, na jakém programu je postaven framework, se může reportování chyb lišit. V případě Javy se všechny kroky a výsledky zapisují do konzoly. A proto by ji pro zjištění problému tester měl prohledat.

6 Závěr

Diplomová práce se zabývá problematikou automatického testování pomocí nástrojů. Část s teoretickými východisky vysvětluje testování softwaru. V této části se objasňují pojmy chyba, selhání, defekt a incident, definují se cíle testování, kontrola a zajištění kvality a jsou vyjmenovány způsoby testování softwaru: manuální a automatické (pomocí nástrojů). Dále se rozebírá proces testování, který zahrnuje takové pojmy, jako jsou politika, strategie, plán a přístup k testování. Definují se vstupní a výstupní kritéria, modely V a W, cykly vývoje SDLC a STLC. Vysvětluje se rozdíl mezi ponětím verifikace a validace. Za procesem testování následují typy prostředí a jejich účely. Dále se rozebírají jednotlivé metody a techniky testování, kam patří funkční, nefunkční testování a typy jejich testů. Poté jsou definovány pojmy jednotlivých komponent aplikace: back-end, front-end a API. Po těchto komponentách jsou rozebrána jednotlivá doporučení od ISTQB.

V kapitole vlastní práce je provedena analýza klíčových slov a trendů softwarů a frameworků pro automatické testování. Analýza probíhala za pomocí Google Trends a měla za účel výběr softwarů a frameworků pro automatické testování. Těmito softwarovými řešeními jsou: Postman, JMeter, Cypress, PlayWright, Selenium, rozdělené podle schopností otestování komponent: front-endu, back-endu nebo API. Frameworky, které jsou schopné testovat všechny komponenty, proto testovaly všechny. Příklad: Cypress a Playwright jsou frameworky, které jsou vytvořeny spíše pro testování front-endu, mohou otestovat i back-end. Byly definovány čtyři experimenty pro dané frameworky a softwary. Každý z experimentů měl svůj cíl a metodiku.

První experiment se zabýval výkonností a efektivitou. Byly definovány tři identické scénáře pro každou komponentu. Scénáře byly definovány pro lokální server a pro vzdálený server. Proběhla exekuce těchto scénářů, kde byl změřen čas vykonávání těchto scénářů. Bylo zjištěno, že u front-endu byl nejrychlejším PlayWright, a to tím, že do polí vkládal text, místo napodobování psaní tohoto textu.

Druhý experiment se zabýval typy testů. Posuzovalo se, jaký typ testů umí každé softwarové řešení podpořit. Podpora testu se také dělila podle front-endu, back-endu a API. Byl vytvořen přehled, jaké typy testů podporují daná softwarová řešení.

Třetí experiment měl za cíl zjistit, jaké dovednosti v programovacích jazycích by měl znát uživatel pro práci s danými nástroji. Bylo zjištěno, že nejvíc programovacích jazyků podporuje framework Selenium, který je určen pro front-end, a JMeter, který je určen pro back-end. Selenium podporuje JavaScript, Python, Ruby, C# a Javu. Jmeter nabízí psaní testů JavaScriptu, Beanshellu, Groovy, Javě a Jexlu.

Čtvrtý experiment se zabýval přesností a spolehlivostí detekce chyb. Byly vyvinuty scénáře pro front-end a zvlášť pro back-end. Každý nástroj detekoval chybu po svém, ale ve své podstatě to byl stejný vzor: očekáváný a skutečný výsledek.

Cíle, které byly stanoveny v diplomové práci, jsou splněny. Byla provedena analýza možností zvolených existujících metod a technologií automatického testování. Nástroje pro automatické testování byly zvoleny na základě analýzy klíčových slov. Na daných nástrojích byly provedeny experimenty. Po provedení experimentů byl vypracován přehled metod automatického testování a přehled vývoje strategií a metodologií automatického testování.

7 Seznam použitých zdrojů

1. BAKKER, Bryan, Graham BATH, Mark FEWSTER, Armin BORN, Judy MCKAY, Andrew POLLNER, Raluca POPESCU a Ina SCHIEFERDECKER. Certified Tester Advanced Level Syllabus. International Software Testing Qualifications Board [online]. 2016, 2016 [cit. 2023-06-07]. Dostupné z: https://istqb-main-web-prod.s3.amazonaws.com/media/documents/ISTQB-CT-TAE_Syllabus_v1.0_2016.pdf

2. BUREŠ, Miroslav, Miroslav RENDA, Michal DOLEŽEL, Peter SVOBODA, Zdeněk GRÖSSL, Martin KOMÁREK, Ondřej MACEK a Radoslav MLYNÁŘ. *Efektivní testování softwaru*. Grada, 2016. ISBN 9788024755946.

3. CHATTERJEE, Shormistha. Difference between SDLC and STLC. Browserstack [online]. 2023, 3. 7. 2023, 2023 [cit. 2023-08-05]. Dostupné z: https://www.browserstack.com/guide/difference-between-sdlc-and-stlc

4. CODESIDO, Ivan. What is front-end development? The Guardian [online]. 2009, 2023 [cit. 2023-06-07]. Dostupné z: https://www.theguardian.com/help/insideguardian/2009/sep/28/blogpost

5. DOSHI, Kalpesh. Types of Testing: Different Types of Software Testing in Detail [online]. 2023, 22. 3. 2023 [cit. 2024-02-11]. Dostupné z: https://www.browserstack.com/guide/types-of-testing

6. HAMILTON, Thomas. Manual Testing Tutorial: What is, Types, Concepts. Guru99 [online]. 2023, 5. 8. 2023, 2023 [cit. 2023-08-05]. Dostupné z: https://www.guru99.com/manual-testing.html

7. HAMILTON, Thomas. What is Software Testing? Definition. Guru99 [online]. 2023, 1. 8. 2023, 2023 [cit. 2023-08-01]. Dostupné z: https://www.guru99.com/software-testing-introduction-importance.html

8. HAYES, Adams, Quality Control: What It Is, How It Works, and QC Careers, investopedia [online], 30. 3. 2023., 2023, [cit. 2023-08-05]. Dostupné z: https://www.investopedia.com/terms/q/quality-control.asp

9. KOĎOUSKOVÁ, Barbora. CO JE TO API A JAKÉ JSOU MOŽNOSTI JEHO VYUŽITÍ? Rascasone [online]. 2023 [cit. 2023-06-07]. Dostupné z: https://www.rascasone.com/blog/co-je-api

10. LEMONAKI, Dionysia. Frontend VS Backend – What's the Difference? Freecodecamp [online]. 2021, 18. 8. 2021 [cit. 2024-02-11]. Dostupné z: https://www.freecodecamp.org/news/frontend-vs-backend-whats-the-difference/

11. SHIOTSU, Yoshitaka. A Beginner's Guide to Back-End Development. Upwork [online]. 2021, 17. 8. 2021 [cit. 2024-02-11]. Dostupné z: https://www.upwork.com/resources/beginners-guide-back-end-development

12. TESTIM. What Is the Software Testing Life Cycle? A Complete Guide. Testim.io [online]. 3. 2. 2023 [cit. 2023-08-05]. Dostupné z: https://www.testim.io/blog/software-testing-life-cycle/

13. V-model. In: Javatpoint.com [online]. 2021, 2011–2021 [cit. 2023-08-15]. Dostupné z: https://www.javatpoint.com/software-engineering-v-model

14. W Model. In: Professionalqa [online]. 2022 [cit. 2023-08-15]. Dostupné z: https://www.professionalqa.com/w-model

15. ZAPTEST. Co je automatizace testování? Jednoduchý průvodce bez žargonu. Zaptest [online]. 2023, 2023 [cit. 2023-06-07]. Dostupné z: https://www.zaptest.com/cs/co-je-automatizace-testovani-jednoduchy-pruvodce-bez-zargonu

8 Seznam obrázků, tabulek, grafů a zkratek

8.1 Seznam obrázků

8.2 Seznam tabulek

8.3 Seznam použitých zkratek

API – Application Programming Interface (Aplikační programovací rozhraní)

BAT – Business Acceptance Testing (Obchodní akceptační testování)

BE – Back End

CRUD – Create, Read, Update, Delete (Vytvořit, Číst, Aktualizovat, Smazat)

CSS – Cascading Style Sheets (Kaskádové styly)

DEV – Development (Vývojové prostředí)

E2E – End-to-End

EDU – Education

FE – Front-End

GUI – Graphical User Interface

HTML – HyperText Markup Language

HTTP – HyperText Transfer Protocol

INT – Integration

ISTQB – International Software Testing Qualifications Board

PHP – PHP: Hypertext Preprocessor

PRE – Pre-production

PROD – Production

PRS – Pre-release system

SDLC – Software Development Life Cycle

STLC – Software Testing Life Cycle

SYS – System

UAT – User Acceptance Testing

Přílohy

Příloha 1 – Zdrojový kód: Cypress – Experiment 1 - backend_test.cy.js

Příloha 2 – Zdrojový kód: Cypress – Experiment 1 -backend_test_local.cy.js

Příloha 3 – Zdrojový kód: Cypress – Experiment 1 - frontend_test.cy.js

Příloha 4 – Zdrojový kód: Cypress – Experiment 1 - frontend_test_local.cy.js

Příloha 5 – Zdrojový kód: Cypress – Experiment 2 - backend_test.cy.js

Příloha 6 – Zdrojový kód: Cypress – Experiment 2 - frontend_test.cy.js

Příloha 7 – Zdrojový kód: Cypress – Experiment 4 - backend_test.cy.js

Příloha 8 –Zdrojový kód: Cypress – Experiment 4 - frontend_test.cy.js

Příloha 9 – Zdrojový kód: Postman – Experiment 1 - Local calls.postman_collection.json

Příloha 10 – Zdrojový kód: Postman – Experiment 1 - Server calls.postman_collection.json

Příloha 11 – Zdrojový kód: Postman – Experiment 2 - testing types.postman_collection.json

Příloha 12 – Zdrojový kód: Postman – Experiment 4 - Error check.postman_collection.json

Příloha 13 – Zdrojový kód: JMeter – API calls.jmx

Příloha 14 – Zdrojový kód: PlayWright – Experiment 1 – BE - APItets.spec.js

Příloha 15 – Zdrojový kód: PlayWright – Experiment 1 – BE - APItets_local.spec.js

Příloha 16 – Zdrojový kód: PlayWright – Experiment 1 – FE - frontend_test-all.spec.js

Příloha 17 – Zdrojový kód: PlayWright – Experiment 1 – FE - frontend_test_local.spec.js

Příloha 18 – Zdrojový kód: PlayWright – Experiment 4 – APItets.spec.js

Příloha 19 – Zdrojový kód: PlayWright – Experiment 4 – frontend_test-all.spec.js

Příloha 20 – Zdrojový kód: Selenium - SeleniumTest.java