Návrh a realizace automatizovaných testů webové aplikace

Bc. Marek Trefil

Diplomová práce
2024

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
Ústav informatiky a umělé inteligence

Akademický rok: 2023/2024

ZADÁNÍ DIPLOMOVÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Bc. Marek Trefil**
Osobní číslo:	**A22401**
Studijní program:	**N0613A140022 Informační technologie**
Specializace:	**Softwarové inženýrství**
Forma studia:	**Kombinovaná**
Téma práce:	**Návrh a realizace automatizovaných testů webové aplikace**
Téma práce anglicky:	**Design and Implementation of Automated Web Application Tests**

Zásady pro vypracování

1. Vypracujte literární rešerši na dané téma.
2. Zvolte vhodné technologie pro realizaci diplomové práce.
3. Navrhněte řešení pro automatizované testování webové aplikace.
4. Navržené řešení implementujte a vyzkoušejte.
5. Řešení zdokumentujte a vhodně prezentujte výsledky práce.

Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam doporučené literatury:

1. KINSBRUNER, Eran a BAHMUTOV, Gleb. *A Frontend Web Developer's Guide to Testing: Explore leading web test automation frameworks and their future driven by low-code and AI*. Packt Publishing, 2022. ISBN 978-1803238319.
2. PALANI, Narayanan. *Automated Software Testing with Cypress*. Auerbach Publications, 2021. ISBN 978-0367759681.
3. SHARMA, Pallavi. *Selenium with Java – A Beginner's Guide: Web Browser Automation for Testing using Selenium with Java*. BPB Publications, 2022. ISBN 978-9391392680.
4. RICHARDSON, Alan. *Selenium Simplified*. 0002-Revised. Compendium Developments, 2012. ISBN 978-0956733238.
5. SUDEN, G. *Automated Web Testing: Step by Step Automation Guide*. CreateSpace Independent Publishing Platform, 2016. ISBN 978-1535285988.

Vedoucí diplomové práce: **Ing. Petr Žáček, Ph.D.**
Ústav informatiky a umělé inteligence

Datum zadání diplomové práce: **5. listopadu 2023**
Termín odevzdání diplomové práce: **13. května 2024**

UNIVERZITA TOMÁŠE BATI VE ZLÍNĚ
Fakulta aplikované informatiky
DĚKAN
L.S.

doc. Ing. Jiří Vojtěšek, Ph.D. v.r.
děkan

prof. Mgr. Roman Jašek, Ph.D., DBA v.r.
ředitel ústavu

Ve Zlíně dne 5. ledna 2024

Prohlašuji, že

- beru na vědomí, že odevzdáním diplomové práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, bez ohledu na výsledek obhajoby;
- beru na vědomí, že diplomové práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk diplomové práce bude uložen v příruční knihovně Fakulty aplikované informatiky. Univerzity Tomáše Bati ve Zlíně;
- byl/a jsem seznámen/a s tím, že na moji diplomovou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- beru na vědomí, že podle § 60 odst. 1 autorského zákona má Univerzita Tomáše Bati ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- beru na vědomí, že podle § 60 odst. 2 a 3 autorského zákona mohu užít své dílo – diplomovou práci nebo poskytnout licenci k jejímu využití jen připouští-li tak licenční smlouva uzavřená mezi mnou a Univerzitou Tomáše Bati ve Zlíně s tím, že vyrovnání případného přiměřeného příspěvku na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše) bude rovněž předmětem této licenční smlouvy;
- beru na vědomí, že pokud bylo k vypracování diplomové práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tedy pouze k nekomerčnímu využití), nelze výsledky diplomové práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem diplomové práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá. Neodevzdání této součásti může být důvodem k neobhájení práce.

Prohlašuji,

- že jsem na diplomové práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.
- že odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou totožné.

Ve Zlíně, dne .....

podpis studenta

ABSTRAKT

Diplomová práce se zabývá návrhem a implementací automatizovaných testů pro webové aplikace. Cílem je zvýšit efektivitu testovacích procesů a snížit náklady spojené s prováděním manuálních testů. Práce zahrnuje teoretický přehled základních principů softwarového testování a popisuje praktické využití automatizace testů na konkrétních příkladech. Výsledkem práce jsou testovací sady pro ověření základních funkcionalit webových aplikací pro místní firmu.

Klíčová slova: Automatizované testování, Webová aplikace, Testovací případ, Selenide, TestNG, Allure, Java

ABSTRACT

The thesis focuses on the design and implementation of automated tests for web applications. The aim is to increase the efficiency of testing processes and reduce the costs associated with manual testing. The thesis includes a theoretical overview of the basic principles of software testing and describes the practical use of test automation with concrete examples. The outcome of the work is test suites for verifying basic functionalities of web applications for a local company.

Keywords: Automated testing, Web application, Test case, Selenide, TestNG, Allure, Java

Děkuji svému vedoucímu panu Ing. Petru Žáčkovi, Ph.D. za vedení mé práce a pomoc při jejím plnění. Dále také děkuji své rodině a přátelům za neustálou podporu.

OBSAH

ÚVOD

Webové aplikace jsou v dnešní době nedílnou součástí našeho života. Nacházejí se všude kolem nás, i když si to vždy nemusíme uvědomovat. Proto je klíčové, aby byly tyto aplikace spolehlivé a pro nás uživatele přívětivé. To se snažíme zajistit pomocí testování, které bývalo v minulosti často opomíjeno vývojovými týmy, nyní se však jedná o nezbytnou součást vývojového procesu softwaru. Důvodem je potřeba zajištění vysoké kvality a funkčnosti aplikací při současném urychlení vývoje a snižování nákladů. Pro tyto účely se vytvářejí automatizované testy, které umožňují organizacím rychle a efektivně provádět komplexní testování bez zásadního zásahu člověka, jež vede k výraznému zlepšení celého vývojového cyklu.

Tato diplomová práce se zaměřuje na návrh a realizaci automatizovaných testů pro webové aplikace vyvíjené místní firmou. Hlavním cílem je zefektivnit proces testování a zejména snížit lidské zdroje potřebné pro pravidelné ověřování základních funkcionalit těchto aplikací. Celý tento proces zahrnuje několik klíčových kroků. Nejprve je nutné provést rešerši potřebných podkladů týkajících se testování, aby byl získán přehled o metodách a přístupech v této oblasti. Následuje výběr vhodného nástroje pro tvorbu testů na základě požadavků projektu a specifik webových aplikací. Dále je proveden návrh řešení automatizace, který zohledňuje strukturu aplikace a cíle testování. Po pečlivém návrhu začne implementace samotných testů, přičemž je dbáno na správné zachycení všech kritických funkcionalit aplikací. V poslední řadě je nezbytné výstupy testování pečlivě zdokumentovat pomocí vhodného reportovacího nástroje, který umožní přehledné zobrazení výsledků a poskytne důležité informace pro další analýzu.

Teoretická část práce se nejprve věnuje vysvětlení pojmu softwarového testování a jeho významu v procesu vývoje. Poskytuje také přehled základních principů automatizace testování, které stanovují pravidla pro provádění testů. Další kapitola se zabývá popisem jednotlivých kroků životního cyklu testování, definovaných vodopádovým modelem. Ten zahrnuje fáze od analýzy požadavků, přes vývoj testovacích případů, až po uzavření testování. V teoretické části je rovněž kapitola zaměřená na obecný popis typů testů a na představení nástrojů a technologií, které se dnes nejčastěji používají pro automatizaci testování. Na konci této části jsou popsány klíčové pojmy jako testovací případ, testovací sada a techniky lokalizace elementů na webové stránce.

V začátku praktické části této práce dochází k představení použitých technologií při vytváření automatizovaných testů jako programovacího jazyku, vývojového prostředí nebo nástroje pro tvorbu testů. Následující kapitola obsahuje detailní přehled webových aplikací, jejichž nejdůležitější funkcionality jsou v práci testovány. Analýza, návrh a implementace řešení, ve které je zaznamenán podrobný popis vývoje testů, je další kapitolou praktické části. V závěru můžeme nalézt přehled vytvořených testovacích pří-

padů s jejich detailním popisem a z jejich grafickým znázorněním pomocí tabulek. Také jsou zde popsány možnosti prezentace výsledků, které je možné pro přehledný výpis testů využít. V poslední řadě je uvedeno celkové shrnutí vytvořené automatizace, kde se zaměřuji na hlavní přínosy práce, problémy, se kterými jsem se při tvorbě potýkal a následné představení budoucího vývoje testů.

I. TEORETICKÁ ČÁST

1 TESTOVÁNÍ SOFTWARU

První kapitola pojednává o klíčovém prvku ve vývoji softwarových produktů, kterým je testování softwaru. Jedná se o v dnešní době nezbytnou část samotného vývojového procesu, který udává pravidla a postup při vytváření nových softwarů.

Jeho hlavním účelem je ověřit, zda výsledný produkt splňuje stanovené požadavky a očekávání cílových uživatelů. Tento proces zahrnuje systematické vyhledávání, identifikaci a odstraňování chyb či nedostatků nebo nekonzistencí ve funkčnosti námi vyvíjeného softwarového systému.

V průběhu let se testování softwaru stalo komplexním oborem s mnoha metodami, technikami a nástroji, které se používají k zajištění kvality výsledných softwarových produktů. Existuje celá řada různých typů testů například testování použitelnosti, systémové testování, akceptační testování, jednotkové testy nebo také integrační či testy zátěžové. Každý ze zmíněných typů má své specifické cíle a přístupy, jakými napomáhá ke zlepšení softwaru.

Softwaroví testeři mají zásadní roli v procesu vývoje softwaru, neboť jsou zodpovědní za plánování, navrhování a provádění testovacích případů a definování testovacích postupů. Mohou využívat manuální testování, kdy testují softwarový produkt ručně nebo automatizované testování, kde využívají specializované nástroje k automatickému provedení testovacích případů. Během testování jsou shromažďovány data a informace o chybách a nedostatcích, které jsou následně předány vývojovému týmu k opravě. Efektivní testování může výrazně snížit riziko výskytu chyb v produkčním prostředí a minimalizovat náklady spojené s opravami a aktualizacemi softwaru.

Nicméně testování softwaru není pouze proces odhalování chyb, ale také přispívá k neustálému zlepšování kvality a spolehlivosti softwarových produktů. Poskytuje zpětnou vazbu k vylepšení uživatelského zážitku a zajištění optimálního fungování softwaru ve všech možných situacích a podmínkách. Testování se také může zabývat otázkou bezpečnosti, kde se jedná o navýšení ochrany před různými druhy útoků. Snažíme se například identifikovat zranitelné části v kódu a architektuře aplikace, které by mohly vést k zneužití či poškození softwaru. [1]

1.1 Webové testování

Webové testování je podkapitolou ke softwarovému testování, jelikož se jedná už o využití specifických druhů testů k naší potřebě ošetřit konkrétní aspekty námi tvořených webových aplikací nebo webových stránek. Samotný význam tohoto testování také spočívá v zajištění kvality webových aplikací a služeb, což je klíčové pro uživatelskou spokojenost a důvěryhodnost firmy či produktu.

Testeři zabývající se touto problematikou se zaměřují především na aspekty jako jsou funkčnost, výkonnost, bezpečnost a uživatelská přívětivost námi vytvářených webových stránek či aplikací. Díky práci testerů se snažíme koncovému uživateli zajistit to, že tento testovaný software bude spolehlivý, bezchybný, a že s ním bude moci efektivně pracovat napříč škálou různých zařízení a webových prohlížečů.

Webové testování zajišťuje nejen funkčnost a výkonnost, ale také se snaží optimalizovat uživatelský zážitek z používání webu, jenž vyžaduje důkladné porozumění interakcí uživatelů. Testování uživatelské přívětivosti se zaměřuje na intuitivnost a použitelnost webových aplikací pro koncové uživatele. Zahrnuje evaluaci navigace, grafického rozhraní a celkového dojmu, který je podstatný pro udržení zájmu a spokojenosti uživatele. Kromě toho, s narůstajícími požadavky na přístupnost webových aplikací, testování dostupnosti získává na významu. To po provedení zaručuje, že webové aplikace jsou přístupné a použitelné pro osoby se specifickými potřebami, včetně těch s různými druhy postižení. Implementace a testování směrnic pro webovou dostupnost pomáhá zajistit, že aplikace jsou dostupné pro širokou veřejnost a tím zvyšují image firmy. V oblasti zabezpečení, testování webové bezpečnosti ochraňuje citlivé údaje před útoky a zneužitím. Bezpečnostní testy, včetně penetračních testů a testování zranitelností, jsou nezbytné pro identifikaci a odstranění potenciálních slabých míst ve webových aplikacích.

Výsledky webového testování přinášejí nejen technická vylepšení, ale také poskytují klíčové informace pro strategické rozhodování v rámci firmy. Systémové testování webových aplikací, které zahrnuje všechny výše uvedené aspekty, zvyšuje celkovou kvalitu vývoje, jenž je zásadní pro udržení konkurenční výhody na trhu. [2]

2 PRINCIPY TESTOVÁNÍ SOFTWARU

Při implementaci testů ke snaze dosáhnout co nejlepší identifikace a odstranit tak, co největší počet defektů a chyb v softwaru potřebujeme dodržovat správnou strategii. Tu nám dohromady tvoří sedm základních principů testování softwaru, se kterými se dále seznámíme blíže. Každý z těchto principů nabízí unikátní pohled na testovací proces a hraje klíčovou roli v zajištění, že software splňuje požadavky uživatelů a poskytuje jim tak požadovanou přidanou hodnotu. Přehled jednotlivých principů viz. obrázek č. (1).

Obrázek 1 Principy testování softwaru [3] - přeloženo autorem

2.1 Testování ukazuje přítomnost defektů

První princip testování softwaru stanovuje, že hlavním účelem testování není prokázat bezchybnost softwaru, ale spíše identifikovat přítomnost defektů. I když se testováním snižuje pravděpodobnost, že v aplikaci zůstanou neobjevené chyby, absence nalezených chyb není důkazem plné funkčnosti testovaného softwaru.

Představte si situaci, kdy se zdá, že pracujete na produktu intenzivně, přijímáte veškerá opatření a vyvíjený softwarový produkt se jeví z 99 % bezchybný. Avšak, i když nebyly identifikovány žádné chyby, není to dostatečný důkaz pro stanovení toho, že software plně vyhovuje potřebám a požadavkům klienta. [3] [4]

2.2 Kompletní testování není možné

Princip pojednávající o tom, že v reálném čase není možné otestovat veškeré možné scénáře. Tedy takzvané testování do vyčerpání není proveditelné. Místo toho musíme vybrat pouze nějaký optimální počet. Takové vybírání se provádí na základě rizik, což znamená, že jsou zvoleny ty testovací případy, které mají největší pravděpodobnost na selhání, tudíž by vedly k identifikování chyby. [3] [4]

2.3 Včasné testování šetří čas a peníze

Třetí pravidlo v pořadí se nám snaží vnutit implementování testů v co nejpočátečnější fázi vývoje softwaru. To zajišťuje zjištění případných nedostatků nebo zachycení špatného návrhu v době, kdy pro nás tato oprava nebude náročná. Je totiž mnohem méně pracné jak z hlediska peněz, tak investovaného času, takovýto defekt odstranit v rané fázi vývoje, oproti například nalezení fatální chyby při konečné implementaci a přepisování tak zásadní struktury naší aplikace. [3] [4]

2.4 Shlukování defektů

V anglickém jazyce známý jako „Defect Clustering“ je dalším důležitým principem testování v pořadí. Jeho problematika se zabývá počtem defektů v určité části našeho softwaru, která je označována jako modul. Na základě takzvaného Paretova pravidla nebo také známého jako pravidlo 80/20 říká, že z pravidla malý shluk modulů obsahuje nejvíce chyb v daném softwaru. Tedy pokud najdeme na nějakém místě chybu, tak jich tam s největší pravděpodobností bude více. Konkrétně se jedná o procentuální zastoupení kde 80 % problému se vyskytuje ve 20 % modulů. [3] [4]

2.5 Pesticidní paradox

Pesticidní paradox, který je v pořadí již pátým principem, opírá svou myšlenku o zkušenost, která může nastat při hubení hmyzu. Konkrétně se jedná o používání stejného přípravku na hubení škůdců opakovaně dokola. To během času povede k navyknutí hmyzu na tento substrát pomocí vývinu jejich rezistence, tudíž bude takovýto přípravek dále k hubení neúčinný. Přesně tato stejná myšlenka se dá převést i do světa testování softwaru, akorát hmyzem budou chyby v aplikaci a přípravek pro odstranění škůdců jsou samotné testy. Pokud se tedy provádí stejná sada testů neustále dokola bez

pravidelné kontroly, aktualizování a přidávání nových případů, bude takovýto test proti odhalování nových defektů dále nepoužitelný. Proto i po zdárném vytvoření testovacích případů musíme neustále dbát na jejich obsluhu. [3] [4]

2.6 Testování je závislé na kontextu

Princip kontextové závislosti testování upozorňuje na to, že není možné aplikovat nějaký univerzální přístup k testování softwaru pro všechny druhy projektů. Specifikace každého projektu, jako jsou cíle, použité technologie, přítomná rizika a očekávání koncových uživatelů, významně ovlivňují volbu testovacích metod a procesů.

Existuje spoustu faktorů, které mají přímý dopad na implementaci testovacích případů. Mezi ně patří například různé odvětví, ve kterém software operuje, regulační požadavky, či velikost a složitost projektu. Proto se v praxi může při tvorbě softwaru pro bankovnictví klást důraz na bezpečnostní aspekty a šifrování dat, zatímco u mobilních her se prioritou stává uživatelský zážitek a výkon aplikace. Tento princip tedy vyzývá k hlubokému porozumění kontextu projektu, které umožňuje efektivní adaptaci testovacích postupů. [3] [4]

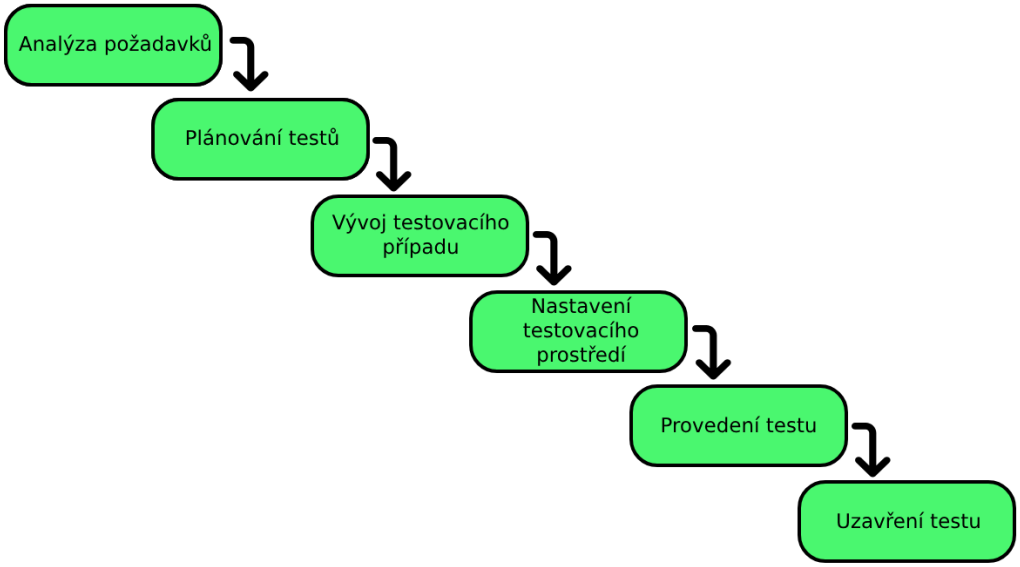
2.7 Chybné tvrzení o nepřítomnosti chyb

Finálním bodem kapitoly pojednávající o základních principech testování se zaobírá tématem ohledně chybného tvrzením o nepřítomnosti chyb. To poukazuje na skutečnost, že testování může být implementované na vysoké úrovni kvality a tím pádem je schopno efektivně identifikovat mnoho přítomných defektů. Avšak nikdy nemůžeme s čirou jistotou prohlásit, že náš software je zcela bezchybný. Je důležité sdělit tento fakt všem zainteresovaným stranám, abychom nastavili realistická očekávání ohledně možné přítomnosti chyb i po pečlivém testování.

Ve výsledku to znamená, že pokud je testování prováděno vedle chybných nebo neúplných specifikací, které neodpovídají skutečným potřebám a očekáváním klienta, nemůže ani správná identifikace a oprava všech zjištěných chyb zajistit, že vyvíjená aplikace bude v praxi úspěšná. Nedostatky v pochopení nebo v komunikaci požadavků mohou vést k situaci, kde je software technicky bez chyb, ale z praktického hlediska nevhodný nebo neefektivní pro jeho zamýšlené použití. [3] [4]

3 ŽIVOTNÍ CYKLUS TESTOVÁNÍ SOFTWARU

Životní cyklus testování softwaru, v anglickém jazyce známý jako Software Testing Life Cycle (STLC), představuje postupný systematický proces, který přispívá k zajištění kvality a efektivity softwarových produktů. STLC je zásadní součást celkového cyklu vývoje softwaru, která umožňuje testovacím týmům plánovat, provádět a hodnotit testy s cílem maximalizovat kvalitu finálního produktu. Tento proces se skládá z celkově šesti fází, viz. obrázek č. (2), které udávají postup k vytvoření kvalitního testu sloužícího k identifikaci a opravě chyb, jenž vede k vývoji robustnějšího a spolehlivějšího softwaru.



Obrázek 2 Životní cyklus testování softwaru

Aby bylo možné každý krok STLC řádně ukončit a mohlo se tak v hierarchii posunout na krok další, musí být vždy nejprve splněna jeho takzvaná vstupní a výstupní kritéria. Kritéria jsou definována pro jednotlivé fáze životního cyklu. Dá se tedy říci, že představují jakési milníky začátku a konce každé fáze. Jejich definování přispívá k dodržení struktury a efektivity celého životního cyklu. [5] [6]

- **Vstupní kritéria -** Definují podmínky, které musí být splněny před zahájením jakékoliv fáze testování. Zajišťují, že testování je zahájeno pouze v případě, že jsou dostupné všechny potřebné informace a zdroje. Mezi typické příklady vstupních kritérií patří připravenost kompletní dokumentace požadavků pro analýzu a přípravu testů nebo také schválení provedení připravených testovacích případů.
- **Výstupní kritéria -** Určují, kdy může být daná fáze testování nebo celkový testovací cyklus považován za ukončený. Tato kritéria jsou základem pro zajištění, že software splňuje předem stanovené standardy kvality a je připraven pro další

fáze vývoje nebo pro uvedení na trh. Příkladem výstupního kritéria může být například úspěšné provedení všech testovacích případů s korektními výsledky či opravení a ověření všech závažných chyb.

3.1 Analýza požadavků

Analýza požadavků je první a klíčovou fází STLC. V ní hlavní činnost provádí takzvaný Quality assurance (QA) tým neboli skupina lidí odpovědných za zajištění kvality softwaru. Ten se v této fázi zaměřuje na pochopení, co přesně má být testováno. To zahrnuje důkladnou kontrolu dokumentů s požadavky na software a komunikaci se zainteresovanými stranami k vyjasnění nejasností. Důležitou součástí je vytvoření matice sledovatelnosti požadavků (RTM), která pomáhá mapovat požadavky na specifické testovací případy, identifikovat případná rizika a připravit půdu pro následující fáze. [5] [6]

3.2 Plánování testů

Fáze plánování testů definuje základ pro celý proces testování. Tým v této fázi formuluje cíle a rozsah testování, vybírá metodiky a techniky testování, dále například identifikuje potřebné zdroje a testovací prostředí. Kromě toho se zde stanovují časové a finanční odhady pro testovací aktivity. Plánování také zahrnuje přidělení rolí a odpovědností členům týmu, což zajišťuje, že každý z nich má přesně dané instrukce co má dělat a testování tak probíhá hladce a systematicky. Některé z výstupních kritériích jsou například schválené testovací plány, automatizační skripty nebo již zmíněné finanční náklady celého projektu. [5] [6]

3.3 Vývoj testovacího případu

Po plánování následuje už samotná etapa vývoje testovacích případů, kde jsou připravovány detailní testovací procedury. Testovací tým také připraví požadovaná data. Tato fáze zahrnuje dále psaní jasných, stručných a přesných testovacích případů, které definují specifikace pro vstupní data, postup provádění testů a jejich očekávané výsledky. Tento proces je nezbytný pro zajištění, že všechny funkcionality softwaru jsou adekvátně pokryty testy a že potenciální chyby jsou efektivně identifikovány. [5] [6]

3.4 Nastavení testovacího prostředí

Příprava a konfigurace testovacího prostředí je samostatná a kritická aktivita, která může probíhat paralelně s vývojem testovacích případů. V této fázi je zajištěno, že testovací tým má k dispozici veškerý potřebný hardware, software a síťové konfigurace

potřebné pro provedení testů. Správné nastavení testovacího prostředí je klíčové pro simulaci reálných podmínek, pod kterými bude software fungovat. [5] [6]

3.5 Provedení testu

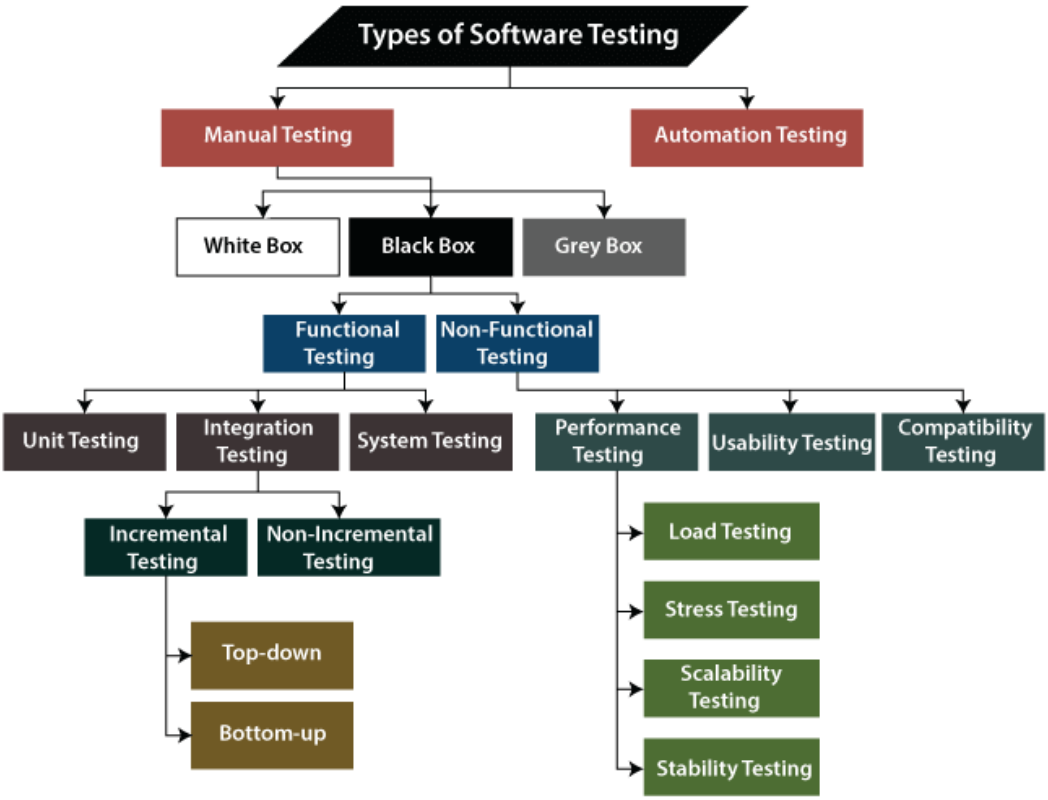
Po dokončení nastavení testovacího prostředí a přípravy testovacích případů následuje fáze provedení testů. V této fázi se testovací případy aplikují na software, aby se ověřila jeho funkčnost a identifikovaly případné chyby. Proces zahrnuje spouštění testovacích případů, zaznamenávání výsledků, analýzu chyb a dokumentaci všech identifikovaných problémů. Efektivní provedení testů vyžaduje systematické sledování a aktualizace testovacích procedur, aby se zajistilo, že všechny aspekty softwaru jsou důkladně ověřeny. [5] [6]

3.6 Uzavření testu

Poslední etapa STLC, uzavření testu, zahrnuje shrnutí všech testovacích aktivit a formalizaci výsledků testování. Hlavním úkolem této fáze je zajistit, že všechny testy byly provedeny, všechny identifikované chyby byly buďto opraveny nebo řádně zdokumentovány, a že software je připraven k vydání. Tato fáze také zahrnuje vyčištění testovacího prostředí, archivaci testovacích dat a přípravu finální zprávy o testování, která poskytuje přehled o provedených testech, nalezených defektech a celkové kvalitě softwaru. Uzavření testů slouží k poskytnutí zpětné vazby pro budoucí projekty a zlepšení testovacích procesů. [5] [6]

4 ROZDĚLENÍ TYPŮ SOFTWAROVÝCH TESTŮ

Existuje řada různých typů testů, které se zaměřují na odlišné aspekty softwarového systému. Od testování funkčních vlastností po testování výkonu a bezpečnosti, každý typ testu má svůj specifický účel a přínos. Tato kapitola se zabývá podrobným popisem konkrétních typů testů softwaru, jejich cíli, metodami provedení a významem v celkovém procesu vývoje softwarových aplikací. Porozumění těmto typům testů je klíčové pro efektivní řízení a optimalizaci kvality softwarových produktů. Jejich přehled a rozdělení můžeme vidět níže na obrázku č. (3) .



Obrázek 3 Typy softwarových testů [7]

4.1 Manuální vs Automatické testování

Prvním hlavním rozdělením je na testování manuální a automatické. Toto rozdělení nám v podstatě říká pouze kdo testování provádí. V případě manuálního typu je aktérem samotný člověk, naopak u automatického jím je stroj. Je tedy důležité porozumět těmto dvěma základním metodám, jelikož tvoří pomyslný základní kámen, dle kterého se další typy testů odvíjejí. [7]

4.1.1 Manuální testování

Manuální testování vyžaduje interaktivní prověření softwaru lidmi. Tester si tedy v tomto případě počíná jako běžný uživatel aplikace a snaží se klikáním na dané webové prvky (např. tlačítka) ověřit jejich správnou funkci. Poskytuje flexibilitu, intuitivní pohled na věc a je nezbytné pro ověření uživatelského rozhraní, překlepů nebo gramatických chyb. Na druhou stranu bývá z pravidla časově náročné, náchylné k lidským chybám a znovu nepoužitelné, tedy testy se musí pokaždé provádět od začátku. Manuální testování je ve výsledku vhodné především pro menší firmy vytvářející jednodušší projekty. [7] [8]

4.1.2 Automatické testování

Automatické testování je prováděno pomocí automatizovaných skriptů a nástrojů, které tyto testy obsluhují. Ty mají po spuštění za úkol ověřit chod aplikace dle definovaných procedur. Hlavní výhody automatického testování jsou znovu použitelnost, časová efektivita, vyšší pokrytí testovaných prvků a vyjmutí lidské chybovosti ze samotného procesu testování. Dále vyžaduje také obvykle vysoké inicializační náklady a pravidelnou údržbu. Je to právě potřeba těchto prvotních nákladů, která mnohé firmy odrazuje před implementací tohoto typu. Automatické testování je vhodné pro projekty s rozsáhlou funkcionalitou, protože manuální testování by v takových případech bylo časově náročné a mohlo by být neefektivní. Tím, že se spouští automatizovaně, umožňuje automatické testování opakovatelně ověřovat širokou škálu funkcí bez nutnosti manuálního zásahu, jenž šetří čas a zvyšuje efektivitu testování. [7] [9] [10]

4.1.3 Zhodnocení

Při rozhodování, která metoda je pro náš kontext vhodnější, záleží na faktorech jako komplexnosti testovaného produktu, dostupných zdrojích a stanovených cílech testování. Celkově je tedy důležité si uvědomit, že jak automatické, tak manuální testování mají své místo v procesu vývoje softwaru a jejich kombinace může přinést nejlepší výsledky. Tímto způsobem lze dosáhnout optimálního zajištění kvality softwarového produktu, a tak naplnit očekávání uživatelů. [7]

4.2 Statické vs Dynamické testování

Tato kapitola se zaměřuje na porovnání a kontrast mezi dvěma základními přístupy k testování softwaru: statickým a dynamickým testováním. Zatímco statické testování se soustředí na analýzu softwarových defektů bez jejich spuštění, dynamické testování se zabývá spouštěním softwaru za různých podmínek s cílem ověřit správnost jeho chování a funkčnost implementace. [13]

4.2.1 Statické testování

Statické testování představuje další klíčovou metodiku zajištění kvality softwaru, která umožňuje analyzovat softwarové chyby bez nutnosti spuštění kódu. Tento proces se zaměřuje na identifikaci chyb a nedostatků v rámci dokumentů jako jsou specifikace požadavků, návrhové dokumenty či samotný zdrojový kód programu. Hledání chyb může provádět jak ručně, tak i pomocí automatických kontrol. Hlavní výhodou statického testování je však to, že odhaluje a řeší problémy již v raných fázích vývoje, kdy jsou náklady na opravu nejnižší.

V rámci statické testování se využívá řada metod a technik, mezi které patří například neformální recenze, technické kontroly, návody a formální revize dokumentace. Tyto techniky umožňují detailní prozkoumání dříve zmíněných pracovních dokumentů a testovacích plánů. Každá z nich má své specifické postupy a cíle, ale společně tvoří komplexní framework pro kontrolu a ověřování kvality softwarových prvků.

Dalším důležitým prvkem tohoto typu testování je statická kontrola kódu, která umožňuje systematickou analýzu zdrojového kódu softwaru bez jeho spuštění. Tato kontrola se zaměřuje na syntaktickou správnost kódu, dodržování standardů kódování a optimalizaci kódu, jenž přispívá k vytvoření efektivního a spolehlivého softwaru. Statické testování je klíčovou součástí procesu zajištění kvality softwaru a jeho správná implementace může výrazně snížit rizika spojená s vývojem a nasazením softwarových aplikací. [11] [13]

4.2.2 Dynamické testování

Dynamické testování se vyznačuje aktivním využíváním softwaru za účelem ověření jeho funkčnosti a chování za různých podmínek. Tento proces zahrnuje spuštění softwarové aplikace a porovnání jejího výstupu s očekávanými výsledky. Klíčovým cílem dynamického testování je potvrdit, že software odpovídá specifikovaným požadavkům a pracuje správně v souladu s očekávanými obchodními scénáři.

Seznam typů dynamických testovacích technik:

1. **Unit testing** (Testování jednotek): V této fázi jsou individuální moduly softwaru testovány samostatně jedna po druhé, a to buď vývojáři nebo inženýry QA.
2. **Integration testing** (Integrační testování): Jednotlivé moduly jsou integrovány a testovány společně, aby se určilo, zda fungují podle očekávání po jejich spojení.
3. **System testing** (Testování systému): Tato metoda testuje celý softwarový systém a ověřuje, zda aplikace splňuje požadavky dle příslušné specifikace.

Dále tento typ umožňuje identifikovat chyby, které by mohly zůstat nepovšimnuté v rámci statického testování. Díky důkladné analýze funkčnosti programu z uživatelského hlediska se zvyšuje kvalita aplikace a umožňuje opravit složité chyby, které by mohly zůstat neodhalené ve fázi kontroly kódu. Automatizace dynamického testování pak přináší další výhody v podobě efektivity a opakovatelnosti testovacích procesů.[12][13]

4.3 Rozdělení dle znalosti kódu

Vývojáři a testovací týmy se často setkávají s otázkou, jak hluboce by měli pronikat do interní implementace kódu softwaru při navrhování a provádění testovacích procedur. Tato otázka ovlivňuje typy testování, které jsou vybrány a provedeny v průběhu vývoje. Rozlišujeme tři hlavní kategorie testování: black-box, white-box a grey-box, které se liší podle úrovně znalosti interního kódu. Každá z těchto metodik má své vlastní výhody a vhodné scénáře použití. Zde se zaměříme na tuto rozmanitost testovacích přístupů a jejich dopad na celkovou strategii testování softwarových produktů.

4.3.1 Black-box

V českém jazyce známa jako černá skříňka je typ testování zaměřující se na zkoumání funkčnosti základních i pokročilých funkcionalit softwaru, avšak bez jakékoliv znalosti jeho vnitřního kódu. Při jeho použití nám tedy není vůbec známo, jak tento náš testovaný produkt funguje. Jsme v tomto přístupu pouze omezeni na jeho vnější chování. To zahrnuje pozorování pouze toho, jak software bude reagovat po vložení námi zvoleného vstupu a jaký od něj dostaneme výstup. Jinými slovy lze říci, že testování černé skříňky je proces kontroly funkčnosti aplikace podle požadavků zákazníka. [14] [15]

4.3.2 White-box

Dalším pojmem v oblasti testování softwaru je metoda známá jako white-box nebo bílá skříňka. Tato testovací strategie je též označována jako otevřená, transparentní či skleněná, což přesně ilustruje její schopnost umožnit nám „nahlédnout dovnitř“ softwarového systému. Na rozdíl od testování black-box, které se zaměřuje pouze na vstupy a výstupy systému, white-box testování provádí důkladnou analýzu interního fungování zkoumaného systému. Jedná se tedy o jeho pravý opak. Dále tento přístup klade důraz na testování softwaru z perspektivy vývojáře a detailně analyzuje vnitřní logiku, výkonnost a efektivitu systému. Tester, který tuto metodiku provádí, musí mít hluboké znalosti o vnitřních mechanismech systému, včetně jeho zdrojového kódu a implementace. Díky tomu je možné provést detailní prozkoumání chování softwaru a odhalit potenciální chyby nebo nedostatky ve zdrojovém kódu, které by bylo obtížné identifi-

kovat bez dostatečných znalostí. [16]

4.3.3 Grey-box

Testování grey-box, tedy šedé skříňky, je sofistikovaná technika testování softwaru, která kombinuje přístupy z obou předchozích skříněk. Tento hybridní přístup se zaměřuje na testování softwaru z perspektivy vývojáře, kde se analyzují určité vnitřní funkce systému, avšak ne všechny. Tester má částečné znalosti o vnitřních mechanismech systému, což umožňuje provádět komplexní testování, aniž by byla zapotřebí úplná znalost interní implementace. Testování šedé skříňky se často využívá při testování systémové integrace, kdy je nezbytné zkoumat interakce mezi různými komponentami systému a ověřit jejich správnou funkčnost a kompatibilitu. Tento přístup poskytuje vyvážený kompromis mezi detailní analýzou interního kódu a nezávislostí na specifických implementačních detailech. [17]

4.4 Funkční vs Nefunkční testování

Nadcházející kapitola popisuje a srovnává rozdíly mezi dalšími dvěma rozděleními v rámci přístupu ke tvorbě softwarových testů. Těmi jsou funkční testování, která obecně kontroluje veškeré funkce implementované ve zkoumané aplikaci a nefunkční testování, to se zaměřuje na testování kritérií jako je výkon či celková použitelnost. Obě tato rozdělení se navzájem doplňují a jsou proto klíčová k zajištění vysokých kvalit vyvíjeného softwaru. [18] [20] [21]

4.4.1 Funkční testování

Funkční testování je metodika, která patří mezi nezbytnou součást každé plnohodnotné kontroly správnosti chodu softwaru. Jedná se tedy o způsob zjištění, zda naše aplikace funguje dle očekávání, tedy zda splňuje funkční požadavky a definovanou specifikaci. V podstatě ověřujeme, že daný software pracuje tak, jak je uvedeno v požadavcích zákazníka. Tyto požadavky jsou často specifikovány pomocí způsobů užití (UseCase), které detailně popisují scénáře, jakým způsobem bude aplikace používána a jaké funkce bude poskytovat.

Dále tento typ testování využívá metodiky černé skříňky (viz. předešlá kapitola Black-box), díky které se nezajímáme o zdrojový kód aplikace, nýbrž o porovnání správnosti výstupů s očekávanými výsledky. Jedná se například o kontrolu různých interakcí s uživatelským rozhraním, prověření správného chování aplikace při zadávání dat nebo kontrolu robustnosti zabezpečení aplikace.

Příklady konkrétních technik funkčního testování:

- **Unit testing** (Testování jednotek)
- **Smoke testing** (Testování kouře)
- **Integration testing** (Integrační testování)
- **Regression testing** (Regresní testování)

Funkční testy mohou být dále prováděny buďto manuálně nebo pomocí automatizovaných nástrojů. Při manuálním testování tester simuluje různé situace a stavy aplikace podle definovaných scénářů, často stejným způsobem, jakým by je používali skuteční uživatelé. Avšak pro větší efektivitu a možnost opakování testování mohou být testy automatizovány pomocí různých nástrojů. Tímto způsobem lze snížit lidskou chybovost a zrychlit celý proces testování. [18] [19] [20] [21]

4.4.2 Nefunkční testování

Druhým zmiňovaným přístupem k tvorbě softwarových testů nese název nefunkční testování. Jedná se o metodiku, při které se provádějí kontroly, které poskytují informace o takzvaných nefunkčních aspektech softwaru. Pod nimi si lze představit testování zaměřené na spolehlivost, použitelnost, výkonnost nebo výšku možné zátěže aplikace.

Spolehlivost se tedy konkrétně zabývá odolností aplikace vůči selhání a chybám. Použitelnost hodnotí uživatelské rozhraní a celkovou uživatelskou zkušenost z používání produktu. Výkonnost kde se zkoumá reakční doba systému za zátěžových podmínek a v poslední řadě zjištění možné maximální zátěže současně přihlášených uživatelů do aplikace.

Příklady konkrétních technik nefunkčního testování:

- **Performance testing** (Testování výkonu)
- **Usability testing** (Testování použitelnosti)
- **Load testing** (Zátěžové testování)
- **Stress testing** (Stresové testování)

Nefunkční testování má stejně důležitou roli jako funkční testování, protože má přímý dopad na spokojenost uživatelů a úspěch softwarového produktu. Nedostatky v nefunkčních aspektech, jako je špatný výkon nebo nízká spolehlivost, mohou negativně ovlivnit uživatelskou zkušenost a reputaci aplikace. [18] [21]

5 NÁSTROJE PRO TVORBU AUTOMATIZOVANÝCH TESTŮ

Kapitola se podrobně věnujeme popisu a analýze vybraných nástrojů používaných pro automatizaci testování webových aplikací. Specificky se zaměřím na nástroje Selenium, Selenide a Cypress, které reprezentují různé přístupy k řešení problematiky testování. Selenium, jakožto průkopník webového testování, poskytuje rozsáhlou podporu pro mnoho prohlížečů a programovacích jazyků. Selenide, který je postaven na Selenium, zjednodušuje interakci s webovými prvky prostřednictvím vylepšeného API a automatické synchronizace. Naopak Cypress představuje moderní řešení, které je zcela integrované a optimalizované pro rychlý vývoj v rámci moderních webových technologií. Tato kapitola nabídne detailní porovnání těchto nástrojů, jenž čtenářům umožní lépe pochopit jejich unikátní vlastnosti a přínosy pro efektivní vývoj a testování webových aplikací. [22] [27] [32]

5.1 Selenium

Selenium je výkonný a flexibilní nástroj pro automatizaci webových aplikací, který se v průběhu let stal standardem v oblasti testování softwaru. Disponuje licencí open-source, tedy otevřeným zdrojovým kódem, která uživateli zpřístupňuje dosažení plné kontroly nad aplikací. Nástroj dále umožňuje testy automatizovat nad různými webovými prohlížeči na různých platformách, což značně zvyšuje efektivitu testovacích procesů. Selenium také nabízí snadnou integraci do různých vývojových prostředí díky tomu, že podporuje širokou škálu programovacích jazyků jako je Java, C#, Python anebo Ruby. Tento fakt také přináší větší rozmanitost pro samotné programátory, jelikož si každý může zvolit ten pro něj nejvhodnější.

Jeho univerzálnost a programátorská přívětivost činí Selenium oblíbenou volbou pro organizace všeho druhů a velikostí, od malých startupů po velké korporace. Tento nástroj umožňuje vývojářům a testerům automatizovat opakující se úkoly testování, zjednodušit složité testovací procedury a rychle identifikovat potenciální problémy s aplikacemi, což vede k zvýšení kvality a snížení času potřebného pro uvedení softwaru na trh.

Nástroj Selenium se celkově skládá ze čtyř nástrojů nebo komponent, které mají každý přesně definovaný svůj rozsah použití. Všechny však vedou k jednomu a tomu samému a tím je testování webových aplikací. Těmito nástroji jsou:

- **Selenium RC (Remote Control)**
- **Selenium WebDriver**
- **Selenium IDE (Integrated Development Environment)**

- **Selenium Grid**

Více se o těchto komponentách dozvíte v následujících kapitolách, kde jsou jednotlivě představeny a v jednoduchosti popsány. [25] [26]

5.1.1 Historie

Selenium bylo prvotně vyvinuto v roce 2004 Jasonem Hugginsem jako interní nástroj společnosti ThoughtWorks. Huggins brzy zjistil, že jeho nástroj může prospět širší komunitě vývojářů a testerů a začal pracovat na tom, aby Selenium bylo dostupné jako open-source. To vedlo k vývoji první verze, známé jako Selenium Core, která umožňovala zaznamenávat interakce s webovou stránkou a opakovaně je přehrávat v prohlížeči. Selenium RC bylo prvním nástrojem, který umožnil testování webových aplikací v prostředích, která byla dříve nedostupná.

Dalším významným milníkem bylo představení Selenium WebDriver v roce 2008, což byla odpověď na některé omezení Selenium RC, zejména v oblasti architektury a výkonu. WebDriver přinesl jednotnější a stabilnější rozhraní pro ovládání prohlížečů, které dále zvýšilo schopnost a výkonnost Selenium.

Spojení komponent Selenium RC a WebDriveru vedla k vytvoření Selenium 2.0, jenž standardizovalo nástroje a práci v rámci Selenium spektra a posunulo automatizaci testování na novou úroveň. Tato verze poskytla vývojářům komplexnější a efektivnější nástroje pro tvorbu testovacích případů, přičemž udržela podporu široké škály programovacích jazyků a integračních možností.

Vytvoření nástroje Selenium je příkladem úspěšného vývoje v oblasti open-source softwaru, kde spolupráce mezi jednotlivci a organizacemi umožnila vytvoření nástroje, který dnes definuje průmyslový standard pro automatizaci testování webových aplikací. [23] [24] [26]

5.1.2 Selenium RC

Selenium RC, celým název Remote Control, tedy vzdálený přístup, byl jednou z klíčových součástí sady nástrojů Selenium pro automatizaci testování webových aplikací. V době své největší popularity umožňoval Selenium RC programátorům psát testovací skripty pro webové aplikace v různých programovacích jazycích. RC kombinoval klientskou knihovnu a server, který fungoval jako HTTP proxy pro prohlížeče, čímž zajišťoval možnost testování aplikací, jako by byly hostovány na jakékoli doméně.

Selenium RC zahrnovalo dvě hlavní komponenty: RC Server a RC Client. Server RC fungoval jako prostředník mezi prohlížečem a testovacími příkazy v programovacím jazyce testera. Tento proxy mechanismus umožnil testování ve více prohlížečích a na

různých platformách, ale přinesl s sebou i několik významných omezení:

1. **Omezení prohlížeče:** Protože RC muselo pracovat prostřednictvím JavaScriptového proxy, bylo často náchylné k problémům s výkonem a omezeními prohlížeče.

2. **Rychlost a výkon:** Použití JavaScriptu pro kontrolu prohlížeče znamenalo zvýšenou režii a často pomalejší provádění testů, což bylo zvláště zřetelné při práci s komplexními webovými aplikacemi.

3. **Komplexní nastavení a údržba:** Nastavení Selenium RC vyžadovalo více kroků, včetně konfigurace serveru a zajištění správné interakce mezi serverem a klientskými knihovnami. Tato skutečnost výrazně komplikovala jeho použití a celkovou údržbu.

S příchodem Selenium WebDriver, který představoval modernější a efektivnější architekturu pro automatizaci prohlížeče, začalo postupné vyřazování Selenium RC. WebDriver přinesl přímou komunikaci s prohlížečem bez nutnosti proxy serveru, což výrazně zlepšilo stabilitu, rychlost a jednoduchost použití. S vydáním verze Selenium 3.0 byl oficiálně z nástroje vyřazen kód Selenium RC a stal se tak plnohodnotnou integrací WebDriveru. [22] [25] [26]

5.1.3 Selenium WebDriver

Selenium WebDriver představuje nejnovější generaci sady Selenium, která byla speciálně navržena pro překonání omezení a nedostatků svého předchůdce, Selenium RC. WebDriver poskytuje rozhraní pro vytváření a spouštění testovacích skriptů, které umožňují přímou interakci s webovým prohlížečem a jeho komponentami na nejnižší úrovni bez nutnosti proxy serveru. Tento přístup umožňuje vývojářům vytvářet sofistikovanější a spolehlivější testovací případy s lepší kontrolou nad chováním prohlížeče.

WebDriver dále podporuje všechny hlavní prohlížeče jako jsou Google Chrome, Mozilla Firefox, Microsoft Edge a Safari, přičemž pro každý prohlížeč existují specifické „ovladače“. Ty jsou navrženy tak, aby optimalizovaly výkon a maximalizovaly kompatibilitu s různými funkcemi prohlížeče, jenž umožňuje testerům vytvářet více robustní a spolehlivé testovací skripty.

Jedním z hlavních přínosů WebDriveru je jeho schopnost přímo ovládat prohlížeč. To má za následek eliminaci zpoždění a chyb spojených s JavaScript proxy, které byly běžné u RC. Díky tomu je WebDriver ideální pro složité webové aplikace, které intenzivně využívají JavaScript a AJAX. Další výhodou je jednodušší a intuitivnější API, které umožňuje snazší psaní a udržování testovacích skriptů.

WebDriver je široce přijímán v průmyslu zabývající se automatizovaným testování webů. Patří dokonce na přední příčky, jako jeden z nejvíce využívaných a preferovaných nástrojů. Jsou to právě všechny předem zmíněné výhody, které testery k jeho využití tvrdě přesvědčují. [22] [25] [26]

5.1.4 Selenium IDE

Selenium Integrated Development Environment, zkráceně IDE, je jednoduché a snadno použitelné rozšíření pro webové prohlížeče, které umožňuje rychlé vytváření a spouštění testů webových aplikací bez potřeby pokročilých programovacích dovedností. Uživatelům tedy umožňuje zaznamenávat své interakce s webovou stránkou a automaticky generovat testovací skripty v jazycích jako JavaScript nebo HTML. IDE bylo původně navrženo pro prohlížeč Mozilla Firefox a později rozšířeno i pro Google Chrome.

Jeho hlavní výhodou je intuitivní grafické uživatelské rozhraní pro záznam, úpravu a ladění testů. To nabízí funkce jako je záznam a přehrávání, které umožňují uživatelům snadno vytvářet testovací případy bez potřeby psaní kódu. Testy lze spouštět buďto po jednom nebo hromadně a jejich výsledky jsou okamžitě k zobrazení, jenž usnadňuje rychlé odhalení a opravu chyb ve vývoji aplikace. Díky své přístupnosti a jednoduchosti je Selenium IDE ideální pro začínající testery nebo vývojáře, kteří se chtějí rychle seznámit s automatizací testů. Také je nástroj užitečný pro vývoj prototypů testů nebo pro rychlé testování malých funkcí, kde psát testy pomocí složitějších testovacích nástrojů by bylo nadbytečné a neefektivní.

Přestože se jedná o na pohled vynikající nástroj má však své nevýhody, které omezují jeho použití v rozsáhlejších a složitějších testovacích projektech. Neobsahuje některé pokročilé funkce, jako je testování v rozdílných prohlížečů kromě Firefoxu a Chromu, také není optimalizován pro velmi složité nebo rozsáhlé testovací procedury, které vyžadují pokročilou logiku a interakci. [22] [25] [26]

5.1.5 Selenium Grid

Selenium Grid je poslední důležitou součástí sady nástrojů Selenium, která umožňuje souběžné spouštění testů na různých strojích a prohlížečích. Tato funkce je klíčová pro zefektivnění procesů testování v prostředích, kde je potřeba testovat aplikace na různých platformách a verzích prohlížečů. Výrazně snižuje čas potřebný pro provádění rozsáhlých testů tím, že umožňuje distribuci testů na více prostředí současně.

Hlavní myšlenka tohoto nástroje funguje na principu hub-and-node. Hub slouží jako centrální bod, který spravuje seznam dostupných nodes (uzlů). Ty představují servery nebo virtuální stroje, které mohou spouštět testy. Grid má díky tomu schopnost paralelně zpracovávat více požadavků na testování zároveň, což minimalizuje zpoždění

spojené s čekáním na dostupnost prohlížeče nebo testovacího prostředí. To je zásadní pro týmy, které pracují na agilním vývoji softwaru a potřebují rychle iterovat a ověřovat změny v aplikacích. [22] [25] [26]

5.2 Selenide

Dalším představovaným nástrojem pro automatizaci testování webových aplikací nese název Selenide. Jedná se o open-source framework, který je zcela postavený na již představeném nástroji Selenium WebDriver v předešlé kapitole [5.1.3]. Byl navržen s cílem zjednodušit proces psaní automatizovaných testů tím, že poskytuje jednodušší a výrazně čistší API pro testování než tradiční Selenium. Selenide automaticky řeší běžné problémy, které bylo nutné v předchůdci složitě implementovat. Mezi ně patří například čekání na elementy, aby se staly viditelnými nebo klikatelnými nebo zjednodušení práce s AJAX aplikacemi. „Concise UI Tests in Java“ je slogan, který doslovně definuje Selenide, v překladu „Stručné testy uživatelského rozhraní v Javě“, zdůrazňuje jeho zaměření na přehlednost a efektivitu kódu.

Díky zavedení automatického čekání umožňuje Selenide testům pokračovat ihned po tom, co jsou požadované webové elementy připraveny k interakci. Tato vlastnost výrazně snižuje množství kódu potřebného k zajištění stability testů a eliminuje tak potřebu explicitně specifikovat čekání ve většině případů. Selenide také integruje přímou podporu pro jazyky Java a Kotlin. Selenide dále podporuje všechny hlavní prohlížeče a umožňuje snadno spustit testy jak lokálně, tak v cloudových službách, jako je BrowserStack nebo Sauce Labs.

Selenide také zjednodušuje interakci s DOM webové stránky poskytováním vysoce abstraktních metod, které zahrnují běžné operace jako jsou vyhledávání prvků, jejich manipulace a ověřování jejich stavu. Práce s Selenide znamená méně starostí s nízkoúrovňovými detaily WebDriveru, což umožňuje testovacím inženýrům soustředit se více na logiku testů než na technické podrobnosti implementace.

Jeho jednoduchost a vysoká úroveň abstrakce činí Selenide oblíbeným nástrojem mezi vývojáři a QA inženýry, kteří chtějí psát čisté, udržitelné a snadno čitelné testy. [27] [28]

5.3 Srovnání Selenium a Selenide

Kapitola se zabývá srovnáním obou nástrojů se zaměřením na jejich klíčové operace, které jsou běžně používány při automatizaci testování webových aplikací.

5.3.1 Inicializace prohlížeče

Inicializace prohlížeče je prvním krokem v testovacím skriptu, kde se vytvoří instance prohlížeče, který bude použit pro testování.

- **Selenium WebDriver:**

```
WebDriver driver = new ChromeDriver();
driver.get("https://example.com");
```

Zdrojový kód 1 Selenium WebDriver - Inicializace prohlížeče

- **Selenide:**

```
open("https://example.com");
```

Zdrojový kód 2 Selenide - Inicializace prohlížeče

Zatímco Selenium vyžaduje explicitní deklaraci WebDriveru a následné zavolání metody pro načtení URL, Selenide zjednodušuje proces na jediný příkaz. Toto zjednodušení může výrazně urychlit psaní testů a zlepšit jejich přehlednost. [29] [30]

5.3.2 Ukončení prohlížeče

Správné ukončení prohlížeče po dokončení testů je klíčové pro uvolnění systémových zdrojů.

- **Selenium WebDriver:**

```
driver.quit();
```

Zdrojový kód 3 Selenium WebDriver - Ukončení prohlížeče

- **Selenide:**

```
// Automaticke zavreni po kazdem testu.
```

Zdrojový kód 4 Selenide - Ukončení prohlížeče

Selenide eliminuje potřebu ručně spravovat životní cyklus prohlížeče, což minimalizuje riziko, že by prohlížeč zůstal otevřen po skončení testů. Toto automatické řízení je obzvláště užitečné v komplexních testovacích procedurách a zvyšuje robustnost testů. [29] [30]

5.3.3 Vyhledávání elementů

Vyhledávání elementů je základní operace, která umožňuje interakci s webovou stránkou.

- **Selenium WebDriver:**

```
WebElement element = driver.findElement(By.id("example"));
```

Zdrojový kód 5 Selenium WebDriver - Vyhledávání elementů

- **Selenide:**

```
SelenideElement element = $("#example");
```

Zdrojový kód 6 Selenide - Vyhledávání elementů

Selenide poskytuje jednodušší a čistší syntaxi pro vyhledávání elementů, která snižuje množství kódu a zvyšuje čitelnost. Tato zjednodušená syntaxe může být méně náchylná k chybám a je rychlejší na psaní. [29] [30]

5.3.4 Kontrola správnosti obsahu elementu

Kontrola, zda element obsahuje očekávaný text, je běžná verifikační operace v testech.

- **Selenium WebDriver:**

```
assertEquals("Expected text",
driver.findElement(By.id("example")).getText());
```

Zdrojový kód 7 Selenium WebDriver - Kontrola správnosti obsahu elementu

- **Selenide:**

```
$("#example").shouldHave(text("Expected text"));
```

Zdrojový kód 8 Selenide - Kontrola správnosti obsahu elementu

Selenide opět zjednodušuje proces aserce s použitím metod, které jsou přímočaré a snadno čitelné. Tyto metody navíc implicitně obsahují čekání, čímž se zvyšuje stabilita testů oproti Selenium, kde je potřeba spravovat čekání explicitně. [29] [30]

5.3.5 Podpora Ajax a asynchronních operací

Asynchronní operace, jako jsou Ajax volání, vyžadují v testech speciální zacházení, aby bylo zajištěno, že elementy jsou dostupné a interakce s nimi jsou platné.

- **Selenium WebDriver:**

```
new WebDriverWait(driver,
Duration.ofSeconds(10)).until(ExpectedConditions.
visibilityOfElementLocated(By.id("example")));
```

Zdrojový kód 9 Selenium WebDriver - Podpora Ajax a asynchronních operací

- **Selenide:**

```
$("#example").shouldBe(visible);
```

Zdrojový kód 10 Selenide - Podpora Ajax a asynchronních operací

Selenide integruje čekání přímo do svých metod porovnání. To znamená, že testy jsou nejen jednodušší na psaní, ale také robustnější vzhledem k asynchronním změnám na stránce. Jedná se o významnou výhodu oproti Selenium, které vyžaduje explicitní nastavení čekacích podmínek. [29] [30]

5.3.6 Přepínání mezi okny nebo záložkami prohlížeče

V moderních webových aplikacích může být nutné přepínat mezi různými okny nebo záložkami. Proto bylo nutné implementovat tyto funkce i pro nástroje automatizace.

- **Selenium WebDriver:**

```
for (String windowHandle : driver.getWindowHandles()) {
    driver.switchTo().window(windowHandle);
}
```

Zdrojový kód 11 Selenium WebDriver - Přepínání mezi okny

- **Selenide:**

```
switchTo().window(1);  // index nebo nazev okna
```

Zdrojový kód 12 Selenide - Přepínání mezi okny

Selenide usnadňuje práci s více okny nebo záložkami poskytnutím přímých metod pro přepínání, které snižují množství kódu a zvyšují čitelnost. Tento přístup je méně náchylný k chybám a rychlejší než manuální správa oken ve WebDriveru. [29] [30]

5.4 Cypress

Další zástupce populárního nástroje pro tvorbu automatizovaných testů webových aplikací nese název Cypress. Tento speciálně navržený framework pro tento typ testovací úlohy je založený na prostředí Node.js. Rychle se stal oblíbeným obzvláště díky svému modernímu přístupu, který se zásadně liší od tradičních Selenium-based nástrojů. Cypress je totiž navržen tak, aby zjednodušil proces testování tím, že běží přímo v prohlížeči a umožňuje tak vývojářům a testerům rychleji iterovat a dosahovat přesnějších výsledků.
Přehled některých zásadních výhod přicházející s použitím nástroje Cypress:

1. **Automatické čekání:** Cypress, obdobně jako předešlý nástroj Selenide, automaticky provádí čekání na elementy a události. To znamená, že není nutné explicitně definovat čekací doby. Toto výrazně zjednodušuje psaní testů.

2. **Real-time testování a reloady:** Cypress nabízí okamžitou zpětnou vazbu během psaní testů díky svému Test Runneru, který okamžitě zobrazuje výsledky a automaticky znovu načítá testy po každé uložení změn.

3. **Integrace s CI/CD:** Cypress lze snadno integrovat do procesů „continuous integration and delivery“, které umožňují automatizaci testování v rámci procesu vývoje softwaru.

4. **Podpora všech moderních prohlížečů:** Ačkoliv Cypress byl původně vytvořen pouze pro Google Chrome, nyní podporuje i další prohlížeče jako Firefox a Edge, tato skutečnost rozšiřuje jeho rozsah využitelnosti.

5. **Snadná ladění:** Cypress usnadňuje identifikaci problémů tím, že poskytuje rozsáhlé možnosti pro ladění testů, včetně snímků obrazovky a videí z testů.

Přehled vybraných limitací nástroje Cypress, které mohou být pro jisté vývojáře testů zásadní:

1. **Omezená podpora jazyků:** Cypress podporuje pouze programátorská jazyk JavaScript. Tato skutečnost může být omezující pro vývojáře a týmy, které používají jiné programovací jazyky.

2. **Neschopnost testovat v několika záložkách nebo oknech:** Cypress neumožňuje testování případů, které vyžadují interakci s více záložkami nebo okny prohlížeče současně.

3. **Nepodporuje všechny typy iFrame**[1]**:** Ačkoliv Cypress dělá pokroky ve zlepšování podpory iFrame, stále existují omezení, která mohou komplikovat testování stránek, které intenzivně využívají tento webový element.

4. **Omezená podpora pro starší prohlížeče:** Cypress neumožňuje testování v prohlížečích jako je Internet Explorer nebo starší verze Safari. Tato skutečnost může být problém pro projekty vyžadující širokou kompatibilitu.

Cypress lze tedy shrnout jako průkopníka, který přináší řadu inovativních funkcí a výhod, které činí automatizaci testování webových aplikací rychlejší, spolehlivější a uživatelsky přívětivější. Jeho schopnosti real-time testování a integrované nástroje pro ladění jsou obzvláště ceněny ve vývojových týmech. I když obsahuje několik omezení, pro mnoho projektů představuje ideální řešení pro automatizaci testů. [32] [33] [34]

5.4.1 Příklady základních testů v nástroji Cypress

V této podkapitole si ukážeme čtyři základní testy v Cypress, které demonstrují jeho schopnosti a jak lze jednoduše testovat různé aspekty webových aplikací. Každý příklad zahrnuje kód testu a vysvětlení klíčových kroků.

1. **Test návštěvy stránky**
   Test číslo jedna demonstruje, jak za pomocí Cypress navštívit webovou stránku. Využívá příkaz *cy.visit()* k otevření zadané URL. Toto je základní krok pro většinu testů, který zajišťuje, že testovaná aplikace je přístupná.

```
describe('Navsteva stranky', () => {
  it('uspesne nacte stranku', () => {
    // Navstivi danou URL
    cy.visit('https://example.com')
  })
})
```

Zdrojový kód 13 Cypress - Test návštěvy stránky

2. **Test vyhledání textu na stránce**
   Tento test ukazuje, jak pomocí *cy.contains()* hledat text na stránce. Tato metoda automaticky čeká na to, až bude text viditelný na stránce, což eliminuje potřebu explicitního čekání.

```
describe('Hledani textu', () => {
  it('najde text "exapmleText" na strance', () => {
    cy.visit('https://example.com')
    // Overi, ze na strance je text "exampleText"
```

[1] Element HTML vymezující plochu na webové stránce, do které umožňuje vložení jiné webové stránky [31]

```
    cy.contains('exampleText')
  })
})
```

Zdrojový kód 14 Cypress - Test vyhledání textu na stránce

3. **Test kliknutí na odkaz**

   V tomto testu se demonstruje, jak interagovat s elementy na stránce. Po návštěvě stránky test hledá odkaz s textem *'exampleText'* a klikne na něj. Cypress poté automaticky zpracuje přechod na novou URL, pokud odkaz vede na jinou stránku.

```
describe('Kliknuti na odkaz', () => {
  it('klikne na odkaz "exampleText"', () => {
    // Navstivi danou URL
    cy.visit('https://example.com')

    // Najde a klikne na element s textem "exampleText"
    cy.contains('exampleText').click()

    // Overi, ze URL obsahuje '/commands/actions'
    cy.url().should('include', '/commands/actions')
  })
})
```

Zdrojový kód 15 Cypress - Test kliknutí na odkaz

4. **Test ověření elementu**

   Poslední ukázka testu je pokročilejší a zahrnuje interakci s formulářovým polem. Test navštíví stránku, najde textové pole pro e-mail, do něhož zadá adresu a pomocí metody *cy.should()* ověří, že hodnota v poli odpovídá zadanému textu.

```
describe('Prace s formularem', () => {
  it('vyplni a overi emailove pole', () => {
    // Navstivi danou URL
    cy.visit('https://example.com/commands/actions')

    // Overi, ze URL obsahuje '/commands/actions'
    cy.url().should('include', '/commands/actions')

    // Najde pole podle tridy a zada do nej email
    cy.get('.action-email').type('example@email.com')

    // Overi, ze pole obsahuje spravnou hodnotu
    cy.get('.action-email').should('have.value',
'example@email.com');
  })
})
```

Zdrojový kód 16 Cypress - Test ověření elementu

Každý z těchto příkladů testů ilustruje, jakou má Cypress psanou syntax a jakým způsobem zjednodušuje testování webových aplikací pomocí jeho mocných a flexibilních funkcí, díky kterým umožňuje testerům efektivněji vytvářet a spravovat testy. [33]

6 NÁVRH TESTŮ

Navrhnutí samotných testů, které po sléze budou programátoři implementovat, je dalším z důležitých aspektů testování. Přece žádný projekt by bez správného návrhu nebyl schopen úspěšně vzniknout, vždy je totiž důležité se držet předem stanovených mezí, a to stejné platí i v oboru testování. Tato kapitola se tedy zaměřuje na definici pojmů jako testovací případ, testovací sada a testovací procedura. Pro úspěšný návrh testů je potřebné tyto pojmy znát.

Při samotném návrhu testovacích případů, sad a procedur je důležité dodržovat několik klíčových principů, aby bylo zajištěno co nejefektivnější testování:

1. **Jasnost a přesnost:** Každý testovací případ by měl být jasný a přesný, aby nedocházelo k žádným nedorozuměním o tom, co a jak testovat.
2. **Zaměření na uživatele:** Testování by mělo reflektovat skutečné uživatelské scénáře a zajišťovat, že aplikace splňuje potřeby a očekávání uživatelů.
3. **Údržba a aktualizace:** Testovací případy by měly být pravidelně revidovány a aktualizovány, aby odpovídaly novým požadavkům a změnám ve funkčnosti aplikace. [35]

6.1 Testovací případ (Test case)

Testovací případ je nejzákladnější stavebním kamenem ve tvorbě testovacího plánu. Jeho hlavním cílem je prověření specifických funkcí aplikace podle požadavků definovaných ve specifikaci. Testovací případ obecně může obsahovat informace ohledně těchto bodů:

- **ID testovacího případu:** Jedinečný identifikátor pro sledování a reportování.
- **Název:** Pojmenování testu.
- **Datum vytvoření**
- **Autor:** Jméno tvůrce testu.
- **Verze:** Aktuální verze testu.
- **Prioritu spuštění:** Kolikátý v pořadí má být test spuštěn.
- **Popis:** Stručný popis toto, čím se testovací případ zabývá.
- **Testovací kroky:** Přesná činnost, kterou bude test vykonávat.

- **Vstupní data:** Data nutná pro vykonání testu.
- **Předpoklady (Pre-condtions):** Podmínky nezbytné pro provádění testu.
- **Očekávaný výsledek (Post-conditions):** Definovaný výstup, který by měl následovat po provedení testovacích kroků.
- **Aktuální výsledek:** Reálný výsledek získaný během testu.
- **Stav:** Určuje, zda testovací případ proběhl úspěšně či nikoliv.

Testovací případy by měly být navrženy tak, aby byly komplexní, ale zároveň jasné a stručné. Musí zajistit, aby každý aspekt testované funkcionality byl důkladně ověřen. [36]

6.2 Testovací sada (Test suite)

Testovací případy se sdružují do testovacích sad, neboli test suite. Ty jsou logicky seskupeny pro testování specifických aspektů aplikace nebo pro provedení testů s různými podmínkami. Pro představu například testovací sada pro přihlášení do aplikace může obsahovat testovací případy pro kontrolu spuštění aplikace, ověření zobrazení vyskakování chybové hlášky po nevyplnění údajů, či jejich chybném zadání, ověření přesunu do hlavního okna aplikace po vyplnění správných údajů apod. Testovací sady tak ve výsledku pomáhají organizovat testovací případy do skupin, kde se každá z nich zaměřuje na kontrolu správného chodu specifické oblasti nebo funkčnosti aplikace. [37]

6.3 Testovací procedura (Test procedure)

Testovací procedury definují způsoby, jakými je možné webovou aplikaci testovat. Může se jednat o jakoukoliv funkcionalitu používanou koncovým uživatelem. Na rozdíl od testovacího případu, který je velmi detailní a zaměřuje se na konkrétní kroky a očekávané výsledky, testovací procedura poskytuje širší pohled na to, co by mělo být testováno bez podrobného popisu, jak testování provést.
Testovací procedury jsou klíčové pro plánování testů, protože:

- **Zjednodušují identifikaci a organizaci testovacích případů:** Pomocí procedur mohou testovací týmy efektivněji plánovat a organizovat testovací případy podle konkrétních obchodních funkcí nebo uživatelských cest.
- **Usnadňují komunikaci s obchodními analytiky a vývojáři:** Umožňují lepší porozumění obchodním požadavkům a pomáhají při návrhu testovacích případů, které odpovídají obchodním cílům. [35]

7 VYHLEDÁVÁNÍ ELEMENTŮ TESTOVANÉHO WEBU

Při vytváření automatizovaných testů uživatelského rozhraní webových stránek je pro vývojáře klíčové umět správně a efektivně identifikovat jednotlivé webové elementy. Webovým elementem můžeme považovat jakýkoli objekt, který uživatel na stránce vidí nebo s nímž může nějakým způsobem interagovat. Tato definice však není úplně přesná, neboť za webové elementy jsou považovány všechny prvky definované pomocí HTML kódu, včetně těch, které nejsou uživateli viditelné.

Webové elementy nám tedy udávají základní stavební bloky webových stránek a jsou zásadní pro tvorbu struktury a funkcionality webového rozhraní. Mezi ně patří například nadpisy, tlačítka, odkazy, rozbalovací menu, textová pole, obrázky a mnoho dalších. Tyto typy se tedy jako programátor automatizovaných testů webů snažíme v našem kódu lokalizovat. Správné vybírání elementů poté zaručuje, že testy jsou rychlé, přesné, a hlavně udržitelné pro budoucí změny. Existuje několik metod, jak na stránkách lokalizovat potřebné webové elementy, každá obsahuje vlastní specifikaci a možnosti využití. [38] [39]

7.1 Příklady lokátorů elementů

Následující podkapitola představuje šest nejpoužívanějších způsobů lokalizování elementů na webových stránkách pro potřeby jejich využití v automatizovaných testech. Vzhledem k tomu, že programátoři při jejich vývoji tráví značnou porci času lokalizací těchto prvků, přijde mi důležité vás čtenáře s těmito základními technikami seznámit.

7.1.1 Atribut elementu - ID

Prvním v řade je lokátor dle jedinečného identifikátoru *'id'*, který patří do definovaných atributů elementu. Jedná se o jeden z nejběžnějších a nejspolehlivějších způsobů pro identifikaci webových elementů, protože každý element na stránce by měl mít svoje unikátní *'id'*. Využití tohoto přístupu je obzvláště rychlé, protože právě podle něj prohlížeče vyhledávání optimalizují. To je důvod, proč jsou tyto lokátory často preferovány pro automatizaci testů, pokud jsou tedy u elementu dostupné, tedy programátorem definované. [38] [39]

7.1.2 Atribut elementu - Name

Lokátory podle názvu jsou další rychlou metodou pro identifikaci elementů, které mají definovaný atribut *'name'*. Tyto lokátory jsou ideální pro formulářové prvky, jako jsou textové pole, rádiové tlačítka, checkboxy a další. Name lokátory jsou výhodné, když více prvků sdílí stejný atribut *'Class'* nebo když *'id'* atribut není dostupný. Avšak,

je důležité mít na paměti, že tyto atributy nejsou vždy unikátní, takže použití tohoto lokátoru může vést k nejednoznačnosti při identifikaci prvků. [38] [39]

7.1.3 Atribut elementu - Text

Tento typ lokátoru je užitečný, když potřebujete identifikovat element na základě textu, který obsahuje. Využití textu uvnitř elementu jako lokátoru je vhodné pro dynamické webové stránky, kde se atributy jako ID a název mohou měnit, ale text zůstává konzistentní. Může se jednat například o lokalizovaní tlačítka nesoucí text „Potvrdit“ či „Načíst“, nebo rozbalovací menu pro výběr měsíce v roce, kde nabízeným prvním prvkem je „leden“. [38] [39]

7.1.4 Atribut elementu - Class Name

Class Name lokátor využívá název třídy elementu pro jeho identifikaci, což je velmi užitečné, když elementy nemají jedinečné ID nebo jiné jednoznačné selektory. Tento lokátor je velmi rychlý a efektivní pro vyhledávání elementů, ale může narazit na problémy, pokud je třída použita u více elementů na stránce. Je důležité si uvědomit, že tento selektor by měl být používán právě v případě, když je název třídy hledaného prvku správně rozlišitelný od druhých. [38] [39]

7.1.5 Selektor CSS

CSS selektory poskytují mnohem robustnější způsob identifikace elementů na základě jejich CSS vlastností. Můžete vyhledávat elementy podle tříd, ID, atributů, hierarchie a dokonce i na základě jejich stavu jako například *:hover* nebo *:active*. CSS selektory jsou výkonné pro vyhledávání složitých a specifických vzorců ve struktuře stránky. Na druhou stranu, mohou být složité na psaní a mohou vyžadovat hlubší porozumění CSS a struktury DOM. [38] [39]

7.1.6 Selektor XPath

XPath je extrémně výkonný jazyk, který umožňuje identifikaci elementů pomocí jejich umístění v DOM (Document Object Model) stromu HTML dokumentu. XPath poskytuje flexibilitu při lokalizování prvků, které nelze snadno identifikovat jinými předešlými metodami. Umožňuje vyhledávání elementů na základě textového obsahu, atributů, hierarchie, pozice a mnoha dalšího. XPath poskytuje vysokou úroveň přesnosti, avšak jeho osvojení a aplikace může být pro neznalé dosti komplikovaná. Kromě toho může na rozsáhlých či komplexních webových stránkách trpět sníženým výkonem. [40]

7.2 Relativní vs Absolutní selektory

Selektory CSS a XPath se dále mohou dělit na takzvané relativní či absolutní. Tento typ se liší dle jejich definované cesty k elementu umístěném v DOM struktuře. Toto rozdělení je zásadní pro efektivní a udržitelný design testovacích skriptů, protože typ použitého selektoru může výrazně ovlivnit jak flexibilitu, tak křehkost testů.

7.2.1 Absolutní selektor

Absolutní selektor je takový, jehož specifikace udává přesnou cestu od kořene dokumentu, kterou se k hledanému webovému elementu krok po kroku dostaneme. Takovýto přístup není zcela vhodný pro dynamické aplikace s měnícími se prvky, jelikož takto „natvrdo“ definovaná pozice nemusí vždy odpovídat hledanému elementu. Další nevýhodou tohoto typu je nefunkčnost při jakékoliv manipulaci se strukturou DOM, jako je přidání, odebrání či přesun jednotlivých prvků. Ačkoliv se použití absolutního selektoru obecně nedoporučuje, v některých specifických situacích může být stále nezbytné. [40]

```
//XPath
/html/body/div[1]/section/main/form/input

//CSS
html > body > div:first-child > section > main > form > input
```

Zdrojový kód 17 Ukázky absolutních selektorů

7.2.2 Relativní selektor

Vždy, pokud to okolnosti dovolují, je doporučeno použít selektor relativní. Jedná se o mnohem více flexibilní selektor, jelikož je schopen k definovanému elementu přistoupit od jakéhokoliv místa ve stromu DOM. To znamená, že není třeba definovat celou cestu ke hledanému prvku stránky, ale pouze použít k identifikaci některý jedinečný identifikátor. Těmi mohou být například předem zmíněný název třídy, ID nebo jiný atribut, díky kterému bude element rozeznatelný od druhých. Vzhledem k tomu, že relativní selektory nezávisí na celkové cestě od kořene, jsou výrazně odolnější vůči změnám v struktuře DOM. [40]

```
//XPath
//form[@id='loginForm']/input

//CSS
form#loginForm > input
```

Zdrojový kód 18 Ukázky relativních selektorů

II. PRAKTICKÁ ČÁST

8 POUŽITÉ TECHNOLOGIE

Tato kapitola je zaměřena na přehled a popis technologií, které byly použity k vytváření a provádění testů v rámci mé diplomové práce. Kapitola poskytne ucelený pohled na nástroje, programovací jazyky, frameworky a další software, které byly klíčové pro efektivní testování webových aplikací. Účelem této kapitoly je nejenom zdokumentovat technologie, které byly použity, ale také vysvětlit, proč byly vybrány a jak přispěly k dosažení cílů testování.

8.1 Programovací jazyk Java

Java je programovací jazyk, který se již více než dvě desetiletí udržuje v čele technologického pokroku, to z něj činí jednu z nejpoužívanějších technologií ve světě vývoje software. Vyvinula ji společnost Sun Microsystems, současným vlastníkem je však nyní společnost Oracle. Jazyk Java je charakteristický především svou přenositelností napříč platformami, objektově orientovaným designem a bezpečnostními prvky, které z něj dělají ideální produkt pro vývoj široké škály aplikací. Pro psaní testů jsem tedy zvolil programovací jazyk Java, se kterým mám již nějaké předchozí zkušenosti. Obory, kterým se Java věnuje mohou být například:

1. **Vývoj her:** Java je populární volba pro vývoj videoher jak mobilních, tak počítačových.
2. **Cloud computing:** Díky schopnosti WORA (Write Once and Run Anywhere), neboli „napsat jednou, spustit kdekoliv“ je Java ideální pro vývoj cloudových aplikací.
3. **Big Data:** Běžně se také používá pro zpracování velkých objemů dat v reálném čase.
4. **Umělá inteligence:** Java podporuje množství knihoven pro strojové učení a je využívána pro vývoj AI aplikací.
5. **Internet věcí (IoT):** Java též nachází uplatnění v programování hardwarových zařízení a senzorů pro IoT. [41]

8.2 Vývojové prostředí IntelliJ IDEA

Dalším použitým nástrojem při vytváření této diplomové práce je integrované vývojové prostředí IntelliJ IDEA. Vyvinula jej mezinárodní softwarová společnost JetBrains s.r.o roku 2001. Ta se specializuje na tvorbu inteligentních nástrojů za účelem zvýšení produktivity vývojářů a celých týmů. Její hlavní sídlo se nachází v České republice -

v Praze, dalších 12 poboček se je situováno různě po světě. Toto vývojové prostředí jsem si vybral na základě mých předchozích zkušeností s jeho používáním, jelikož se jedná o hlavní vývojové prostředí ve firmě, kde pracuji.

IntelliJ IDEA je široce uznávaným produktem mezi vývojáři, kteří se zabývají vyvíjením softwarů v jazyce Java nebo Kotlin. Podporuje však také další jazyky založené na JVM (Java Virtual Machine) jak jsou Scala, Groovy nebo Clojure.

IntelliJ IDEA je dostupná ve dvou hlavních verzích: bezplatné Community Edition a placené Ultimate Edition. Ultimate Edition nabízí rozšířené funkcionality pro webový vývoj, podnikové frameworky a další nástroje. JetBrains poskytuje také devadesátidenní bezplatnou zkušební verzi pro Ultimate Edition, které vývojářům umožňuje otestovat plné možnosti IDE před jeho koupí. Cena individuální verze pro Českou republiku aktuálně začíná na 4,190 Kč bez DPH za první rok s postupným snižováním ceny v následujících letech. [42]

8.2.1 Použitá Rozšíření

Kromě standardních vývojových funkcí, IDEA nabízí různá rozšíření pro přidání dalších funkcionalit do aplikace. Pro vylepšení práce s vytvářením testů webových aplikací jsem využil rozšíření s názvem **Test Automation**[1)]. To přináší řadu nástrojů a funkcí, které usnadňují a automatizují procesy testování, jako jsou:

- Podpora pro populární testovací frameworky jako JUnit, TestNG, Selenium, Selenide a mnoho dalších.
- Zvýraznění a ověření XPath, CSS a JavaScriptu v rozhraní API pro testování uživatelského rozhraní.
- Vestavěný webový inspektor, který poskytuje možnost generovat jedinečné lokátory CSS nebo XPath pro prvky na webové stránce a následně je přidat přímo do kódu.

8.3 Selenide

Jaký nástroj použít pro vytváření automatizovaných testů webových aplikací bylo těžké zvolit. Za své favority jsem měl především Selenium a Selenide, jelikož mi byly doporučeny nadřízenými. Jejich podrobné specifikace popisuji v kapitolách [5.1] a [5.2]. Po dlouhém bádání jsem se rozhodl zvolit právě Selenide, a to z několika pro mě jasných důvodů. Jelikož do oblasti testování webových aplikací jsem zabrouzdal zcela poprvé, a tudíž jsem v tomto oboru úplným nováčkem, byla mi přívětivější jeho jednoduchost

[1)] Odkaz na plugin Test Automation: `https://plugins.jetbrains.com/plugin/20175-test-automation`

a přehlednost při psaní kódu. Na svých webových stránkách má Selenide velice vystihující citát, ten v překladu zní: „Špatný software nemá dokumentaci. Brilantní software dokumentaci nepotřebuje.“ S tímto názorem na nástroj Selenide se můžu ztotožnit a potvrdit, že pro většinu testů opravdu listovaní v dokumentaci nebylo nutné. Našlo se však pro mé potřeby pár výjimek, kdy jsem přesně nepochopil práci s konkrétní metodou a byl jsem tedy nucen si přesnou definici v dokumentaci dohledat. [28]

8.4 TestNG framework

TestNG je robustní open-source testovací framework pro jazyk Java. NG v názvu znamená „Next Generation“, tedy v českém jazyce „Nová generace“, což nám říká, že se jedná o vylepšení frameworků předešlých o nové a lepší funkcionality. Od svého vzniku, který má na svědomí pan Cedric Beust, se TestNG stal oblíbeným nástrojem mezi vývojáři a testery právě díky svým pokročilým funkcím a flexibilitě. Tento framework je ideální pro různé typy testování jako jsou například jednotkové testy, integrační nebo takzvané end-to-end testy.

TestNG zavádí koncept anotací, které slouží k definování a konfiguraci testů. Tyto anotace jsou klíčové pro správné fungování testů a jejich správu. Příklady některých jejich zástupců s příslušnou možností využití:

1. **@Test:** Jedná se o základní anotaci používanou k označení metody v kódu jakožto metody testovací, která by měla být spuštěna pomocí TestNG. Obsahuje atributy jako:
   - **priority:** Určuje pořadí, ve kterém by měly být testovací metody spuštěny.
   - **enabled:** Umožňuje nebo zakazuje spuštění testu. Užitečné pro dočasné vypnutí testu bez jeho odstraňování.
   - **dependsOnMethods:** Zajistí, že metoda bude spuštěna pouze po úspěšném dokončení závislých metod.
   - **dataProvider:** Určuje metodu, která poskytuje data pro test.
   - **groups:** Umožňuje klasifikaci testových metod do skupin, které poté lze pouštět paralelně.

```
@Test(priority=1, enabled=true, groups={"login"})
public void loginTest() {
    // Implementace testu
}
```

Zdrojový kód 19 TestNG - Anotace @Test

2. **@BeforeSuite a @AfterSuite:** Tyto anotace určují metody, které se mají spustit před nebo po všech testech v testovací sadě. Ideální pro nastavení nebo úklid zdrojů, které jsou sdílené mezi všemi testy.

```
@BeforeSuite
public void setupDatabaseConnection() {
    // Kod pro nastaveni databaze
}

@AfterSuite
public void closeDatabaseConnection() {
    // Kod pro uzavreni databaze
}
```

Zdrojový kód 20 TestNG - Anotace @BeforeSuite a @AfterSuite

3. **@BeforeTest a @AfterTest:** Spouští se před a po skupině testů, které jsou definovány v XML konfiguraci TestNG. Užitečné pro přípravu a úklid, které jsou specifické pro některé skupiny testů.

```
@BeforeTest
public void beforeTestSetup() {
    // Konfigurace potrebna vykonat pred testem
}

@AfterTest
public void afterTestCleanup() {
    // Uklid po skupine testu
}
```

Zdrojový kód 21 TestNG - Anotace @BeforeTest a @AfterTest

4. **@BeforeClass a @AfterClass:** Anotace, která slouží pro nutnou konfiguraci před spuštěním prvního testu dané třídy a pro následný úklid po vykonání všech testů třídy.

```
@BeforeClass
public void configureBrowser() {
    // Konfigurace weboveho prohlizece
}

@AfterClass
public void removeCookies() {
    // Vymazani cookies
}
```

Zdrojový kód 22 TestNG - Anotace @BeforeClass a @AfterClass

5. **@BeforeMethod a @AfterMethod:** Vhodné pro metody, které je třeba provést před každým testem a následně po každém provedeném testu.

```
@BeforeMethod
public void openWebBrowser() {
    // Otevreni weboveho prohlizece
}

@AfterMethod
public void closeWebBrowser() {
    // Zavreni weboveho prohlizece
}
```

Zdrojový kód 23 TestNG - Anotace @BeforeMethod a @AfterMethod

Jak můžeme vidět framework TestNG má mnoho anotací, které mají veliký rozsah použitelnosti. Je důležité si uvědomit, že každá ze zmíněných anotací má své dané pořadí, ve kterém se vykonají při hromadném spuštění testů. Pořadí přehledně zobrazuje obrázek č.(4).

Obrázek 4 Hierarchie spuštění TestNG anotací [44]

TestNG je v oblasti automatizace testů webových aplikací v kombinaci s nástrojem Selenide považovaný za výkonné duo a je to právě tato kombinace, která se mi nejvíce zalíbila, a kterou jsem tedy použil pro tvorbu a organizaci mých testů. TestNG, jako flexibilní testovací framework, přináší strukturu, organizaci a rozšířené možnosti správy testů. Na druhé straně, Selenide nabízí vývojářům vysoce abstraktní API pro interakci s webovými prohlížeči, což zjednodušuje psaní čitelnějších a udržovatelnějších testů. Společně tato kombinace poskytuje pevný základ pro rychlý a efektivní vývoj automatizovaných testů webových aplikací. [44] [43]

9 PŘEHLED TESTOVANÝCH APLIKACÍ

Kapitola představuje mnou testované webové aplikace. Byly vybrány a konzultovány na základě požadavků od nadřízených. Aplikace se dají rozdělit do tří typů, kterými jsou Vademecum, ProMuzeum a Spisová služba. Každá z nich se odlišuje svou implementací, funkcionalitou a specifickými informacemi, které jsou tak uchovávány pro dané instituce. Celkově lze říci, že se každá z testovaných aplikací zaměřuje na zprávu dokumentů různých organizací, jako jsou knihovny, muzea, galerie a jiných podobných archivních zařízení. Jejich hlavním cílem je uživateli zjednodušit práci se zpracováním potřebných dokumentů pomocí digitalizace činnosti archiváře.

9.1 VadeMeCum

Muzejní VadeMeCum je inovativní modulární systém navržený pro státní i soukromá muzea a galerie. Tento systém usnadňuje zpracování, ukládání a prezentaci muzejních sbírek, čímž podporuje efektivní správu unikátních historických materiálů. Využitím prezentační aplikace VadeMeCum se uložené sbírky stávají snadno dostupnými a vyhledatelnými, což zvyšuje jejich přístupnost jak pro odborníky tak veřejnost. Hlavním přínosem systému je eliminace duplicitních prací, dále šetření času a celkových zdrojů, také zároveň zajišťuje bezpečnost dat prostřednictvím centralizovaného ukládání. VadeMeCum je snadno implementovatelný kvůli svému webovému rozhraní, které zjednodušuje jeho obsluhu a správu. [45]

9.1.1 Národní galerie v Praze

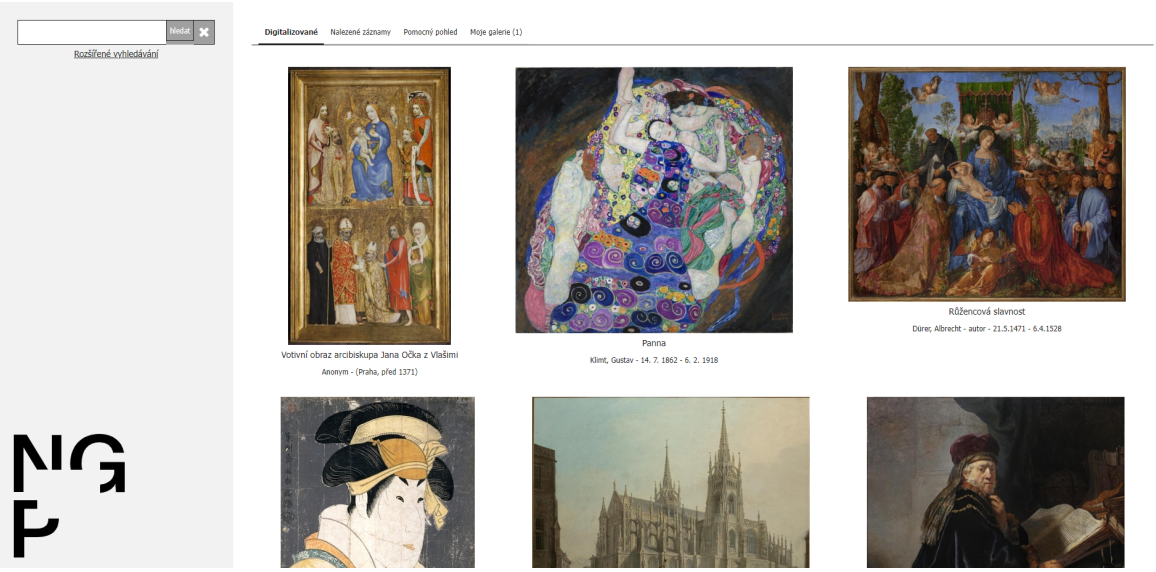
Jednou z prvních mnou testovaných aplikací je VadeMeCum Národní galerie města Prahy. Jedná se o velmi jednoduchou, a tedy veřejnosti přístupnou webovou aplikaci zaměřující se především na vyhledávání a průzkum uložených sbírek umění této galerie. V době psaní práce se v této digitální galerii nachází přes 45000 záznamů.

Uživatel může dílo vyhledávat několika způsoby, včetně fulltextového vyhledávání, vyhledávání podle inventárního čísla nebo jména autora. Kromě toho nabízí rozšířené vyhledávání možnost hledat dle názvu konkrétní sbírky, ve které se hledaný objekt nachází, nebo podle typu podsbírky, jako jsou Malba, Grafika nebo Kresba. Po nalezení díla je uživatel přenesen na kartu „Nalezené záznamy“, kde se výsledky zobrazují v defaultním mozaikovém zobrazení. Tento režim lze ale změnit na seznamový nebo detailní pohled. Z této stránky je možné prohlížet nalezené záznamy nebo je přidávat do virtuálního košíku, který se zobrazí až po první aktivaci funkce „Vložit do košíku“ s ikonou hvězdy.

Aplikace umožňuje zobrazení detailu záznamu přímým kliknutím na jeho tělo. V de-

tailním pohledu lze kliknutím na náhled díla zobrazit obrázek v plné kvalitě. To je umožněno díky nástroji Zoomify, který je schopen na webových stránkách prezentovat obrázky ve vysokém rozlišení. Dále u detailu díla je uživateli u každého záznamu, pokud definovány, zobrazeny tyto základní informace: [46]

- **Autor:** Jména autora či autorů daného díla.
- **Název:** Celý název.
- **Inventární číslo:** Jedná se o jednoznačný identifikátor každého uloženého záznamu.
- **Datace/Datování:** Datum vzniku díla.
- **Technika:** Technika, pomocí které bylo dílo vytvořeno.
- **Materiál:** Materiál, ze kterého dílo vzniklo.
- **Rozměry:** Výška, šířka, popřípadě hloubka díla.
- **Permalink:** Vytvořený permanentní odkaz na daný záznam.



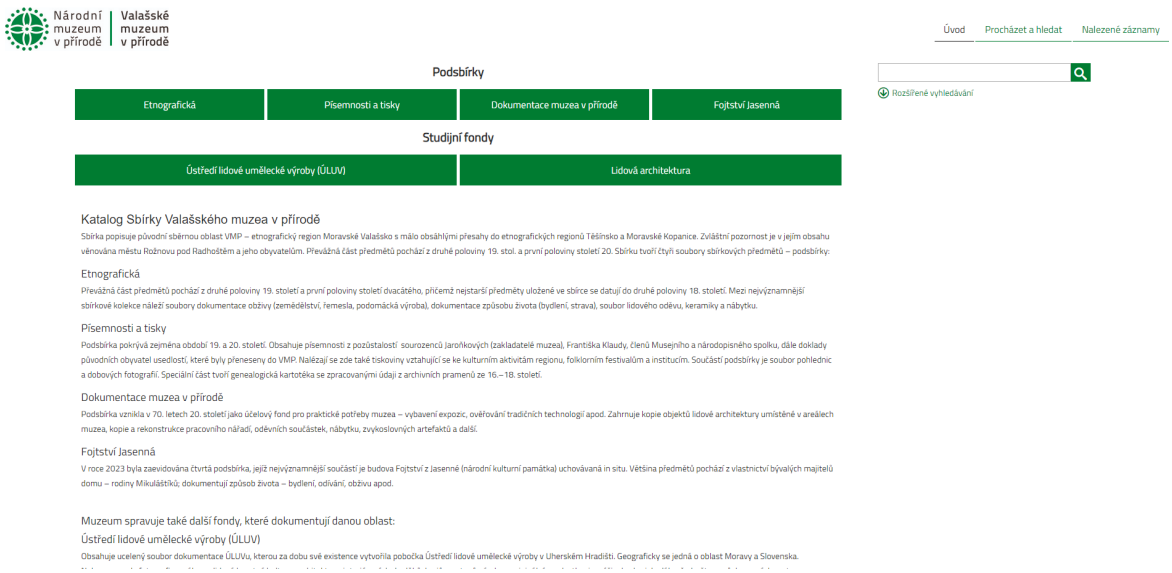
Obrázek 5 Ukázka vzhledu aplikace Národní galerie v Praze

9.1.2 Valašské muzeum v přírodě

Jedná se o online volně přístupný katalog sbírek Valašského muzea v přírodě jakožto Národní kulturní památky. Valašské muzeum v přírodě je živým svědkem bohaté lidové kultury a tradic, a tato webová stránka slouží jako digitální vstupní brána do jeho rozsáhlých sbírek. Zde mohou uživatelé objevovat a studovat sbírky uměleckých děl, nábytku, oděvů a dalších předmětů, které dokumentují život a řemesla regionu Moravského Valašska a Těšínského Slezska. V současné době je na webu prezentování přes 190 000 záznamů.

Stránka je strukturovaná do několika hlavních sekcí těmi jsou podsbírky „Etnografická“, „Písemnosti a tisky“, „Dokumentace muzea v přírodě“, „Fojtství Jasenná“. Muzeum spravuje také další fondy „Ústředí lidové umělecké výroby (ÚLUV)“ a „Lidová architektura“, které dokumentují danou oblast. Každá sekce vede ke záznamům nacházejících se v dané podsbírce.

Co se funkcí týče je tato webová stránka velice podobná předchozí zmíněné pražské Národní galerii, jelikož na její vývoj byla použita stejná vývojová šablona. Přichází tedy jen s některými upravenými funkcemi na míru a změněným vzhledem, dle požadavků zákazníka. Mezi ně patří například úprava rozšířeného hledání a možnost procházení jednotlivých sbírek a podsbírek pomocí jejich rozbalování. Také můžeme nalézt lehké vizuální změny v možnostech zobrazení seznamu, mozaiky a detailu. [47]



Obrázek 6 Ukázka vzhledu aplikace Valašského muzea v přírodě

9.1.3 Středočeské muzeum v Roztokách u Prahy

Tato testovaná veřejnosti dostupná sbírka katalogů prezentuje záznamy ze dvou zařízení současně. Konkrétně se jedná o údaje Středočeského muzea v Roztokách u Prahy a Sládečkova vlastivědného muzea v Kladně. Podsbírky se dělí na archeologické, botanické, etnografické, historické, písemnosti a tisky, zoologické a vzorkovnice keramiky ÚUŘ, to však pouze u sbírky Středočeského muzea. U druhého muzea se vyskytuje například podsbírka knihy, výtvarného umění, numizmatická nebo svatováclavské oslavy.

Vylepšené rozšířené vyhledávání, dostupné pod záložkou „Katalog sbírek“, umožňuje uživatelům nalézat záznamy z různých podsbírek. Vyhledávání začíná výběrem instituce, jako je muzeum, a následnou specifikací příslušné podsbírky. Další významnou novinkou na tomto webu je přidání tlačítka „Rezervovat“ do sekce košíku. Po kliknutí na toto tlačítko se uživateli zobrazí formulář, který umožňuje objednat si kopii dokumentu, nebo v případě jeho nedostupnosti dokonce i originál pro studium v badatelnách archivu.

Jelikož se také jedná o vytvořený web pomocí předem dané stále se opakují šablony, tak má velice podobnou strukturu jako již dříve zmíněné. Některé funkce jsou přemístěny nebo zaměněny, jako například přepínání mezi zobrazením mozaiky, detailu a seznamu nebo otevírání obrázku v plné kvalitě.

Používání stejné šablony pro vytváření aplikací typu VadeMeCum je velmi praktické z důvodu, že umožňuje archivním pracovníkům snadněji využívat online sbírky jiných zařízení, ale také badatelům usnadňuje práci díky konzistentnímu a známému uživatelskému rozhraní napříč různými aplikacemi. [48]

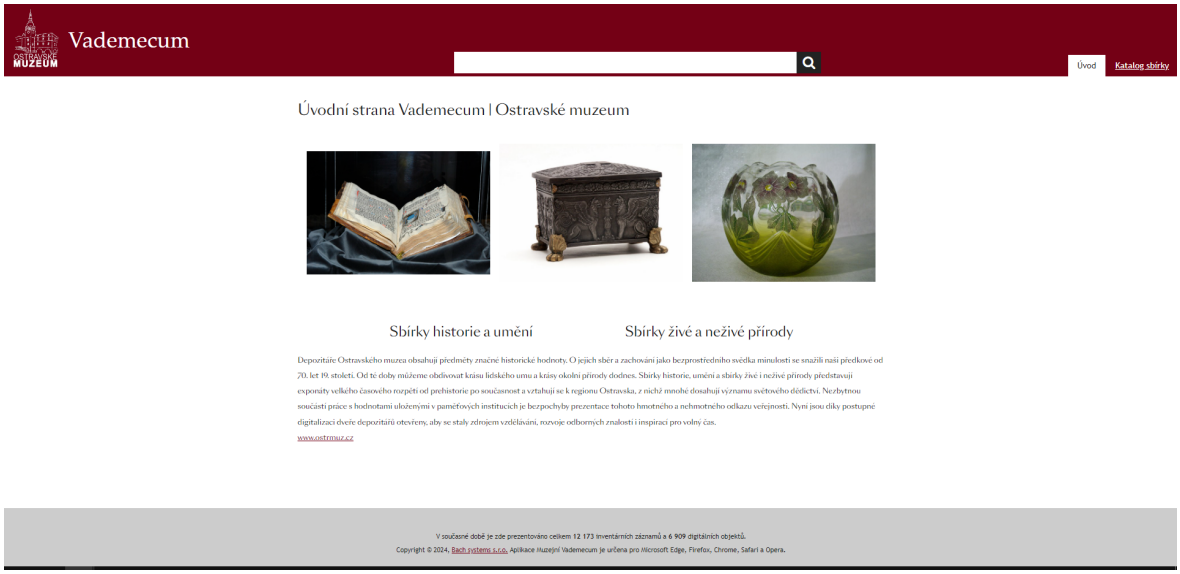
Obrázek 7 Ukázka vzhledu aplikace muzea Roztoky

9.1.4 Ostravské muzeum

„Depozitáře Ostravského muzea obsahují předměty značné historické hodnoty. O jejich sběr a zachování jako bezprostředního svědka minulosti se snažili naši předkové od 70. let 19. století. Od té doby můžeme obdivovat krásu lidského umu a krásy okolní přírody dodnes. Sbírky historie, umění a sbírky živé i neživé přírody představují exponáty velkého časového rozpětí od prehistorie po současnost a vztahují se k regionu Ostravska, z nichž mnohé dosahují významu světového dědictví. Nezbytnou součástí práce s hodnotami uloženými v paměťových institucích je bezpochyby prezentace tohoto hmotného a nehmotného odkazu veřejnosti. Nyní jsou díky postupné digitalizaci dveře depozitářů otevřeny, aby se staly zdrojem vzdělávání, rozvoje odborných znalostí i inspirací pro volný čas.“ [49]

Tento text se nachází na úvodní stránce posledního příkladu mnou testované webové stránky formátu VadeMeCum. Kompletně vystihuje její přesnou podstatu a zdůrazňuje také úděl a dostupnost široké veřejnosti.

Co se týče funkčnosti aplikace, je téměř identická s jejími předchůdci. Jedinou významnější změnou je úprava možnosti vyhledávání dle zvolené sbírky v rámci rozšířeného výběru. Nepřináší žádné nové změny v rámci funkcionality, pouze v malé míře upravuje uživatelské rozhraní k obrazu svému. Naopak některé funkce jako například rezervace v košíku nebo tab „Nalezené záznamy“, jsou v této implementaci VadeMeCum opomenuty.



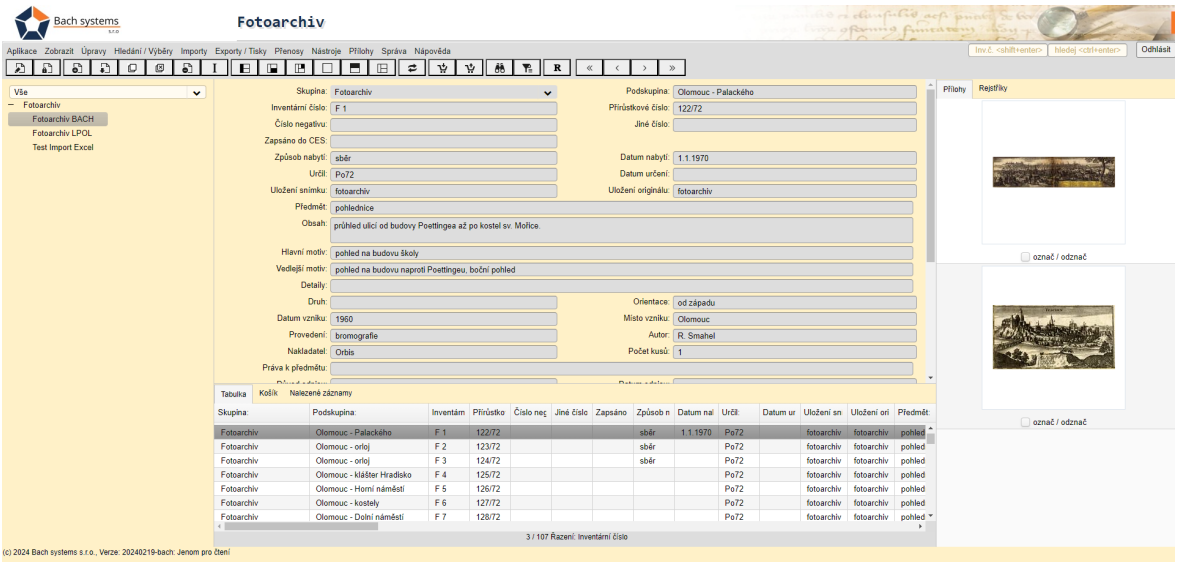
Obrázek 8 Ukázka vzhledu aplikace Ostravského muzea

9.2 ProMuzeum

Webové aplikace ProMuzeum a VadeMeCum společně tvoří jeden ucelený informační systém. Zatímco VadeMeCum slouží primárně k efektivní prezentaci vybraných muzejních sbírek veřejnosti na internetu, ProMuzeum je určené pro záznam těchto sbírek do evidence. Tyto dvě aplikace se tedy liší jak svými funkcionalitami, celkovým vzhledem nebo složitostí, tak hlavním účelem užití. Informační systém ProMuzeum není určen široké veřejnosti, ale pouze pro pracovníky daného archivu, který tento portál zastupuje. Jedná se o platformu používanou různými muzei, galeriemi a dalšími podobnými úředními institucemi, kde je potřeba digitalizovat zaznamenávání a správu velkého množství archivních záznamů.

9.2.1 FotoArchiv

FotoArchiv je konkrétní implementací webové aplikace šablony ProMuzeum. Jedná se o složitější aplikaci než pouze prezentační předchůdce VadeMeCum. Obsahuje mnoho funkcí pro práci s daty daného archivního ústavu. Tato konkrétní testovaná implementace s názvem FotoArchiv slouží pro evidenci a správu fotografií různých předmětů. Testovaný web představuje pouze demo verzi webové aplikace FotoArchiv. Má za úkol výhradně prezentovat zájemcům vzhled a funkcionality, které je možné pro potřebu odzkoušet. Z toho důvodu je ho možné přes zadanou adresu volně nalézt na internetu, obvykle totiž běží jen v lokálním prostředí daného ústavu, který je vnějšímu světu nedostupný. Testování této aplikace bylo iniciováno s cílem zjistit, zda je možné vytvořit definované testovací procedury dle požadavků nadřízených, a jak dlouho bude jejich implementace trvat.



Obrázek 9 Ukázka vzhledu dema aplikace FotoArchiv

Na základním vzhledu pracovní plochy webové aplikace FotoArchiv, který můžeme vidět na obrázku č. (9), lze nalézt grafické rozpoložení na tyto prvky:

- **Nabídkové menu**
- **Panel nástrojů**
- **Panel příloh**
- **Zobrazení příloh/rejstříků**
- **Vyhledávání dle inventárního čísla**
- **Fulltextové vyhledávání**
- **Panel navigátoru**
- **Zobrazení záznamu ve formě karty**
- **Zobrazení záznamů ve formě tabulky**
- **Košík**
- **Nalezené záznamy**

Nabídkové menu této webové aplikace je bohaté na různé funkce, které usnadňují práci s archivními záznamy. Uživatelé zde najdou nástroje pro import a export záznamů umožňující efektivní výměnu dat mezi různými systémy. Kromě toho aplikace nabízí rozšířené vyhledávací a výběrové možnosti, které uživatelům dávají přístup snadno najít a spravovat požadované informace. Další důležitou funkcí je možnost připojování příloh k záznamům, která podporuje lepší dokumentaci a archivaci souvisejících materiálů.

Pod nabídkovým menu se nachází panel nástrojů, obsahující klíčová tlačítka pro běžné operace v archivu. Ta umožňují uživatelům přecházet mezi režimy zápisu a čtení, které jsou pro správu dat zásadní. Uživatelé mohou také vytvářet nové záznamy, ukládat změny po jejich modifikaci nebo upravovat rozvržení panelů na stránce pro optimalizaci pracovního prostředí dle své potřeby.

Hlavní obrazovka pro zobrazení jednotlivých archivních sbírek zahrnuje navigační panel, který uživatelům umožňuje snadný přístup k různým sbírkám. Po výběru specifické sbírky jsou záznamy uživateli prezentovány ve dvou hlavních formátech a to jako karta nebo tabulka. Zobrazení ve formě karty poskytuje podrobný přehled o všech atributech zobrazeného záznamu, které je ideální pro hlubší analýzu jednotlivých položek. Na druhou stranu, řádky ve formě tabulky umožňují uživatelům efektivně prohlížet a porovnávat dílčí záznamy uložené ve zvolené sbírce, což usnadňuje rychlý přehled a správu velkého množství dat. [50]

9.3 Spisová služba WISPI

V dynamickém prostředí každé organizace je správné zpracování dokumentů zásadní pro její plynulý chod a efektivní komunikaci. WISPI, jako inovativní informační systém, přichází s řešením, které uceleně pokrývá celý životní cyklus dokumentu, tedy od jeho přijetí až po konečné uložení do archivu.

Webová aplikace WISPI je postavena na komplexním základě, který zahrnuje organizaci a sledování dokumentů v rámci celé struktury podniku. Ať už se jedná o příchozí poštu, interní dokumentaci nebo dokumenty generované samotnou organizací, WISPI to vše eviduje s precizností a transparentností. Základní funkce tohoto systému jsou přidělování jednacích čísel, efektivní distribuce mezi odděleními, tvorba a správa spisů, včetně jejich zpracování.

Design systému je vyvinut s ohledem na potřeby uživatele. Jednotné uživatelské rozhraní, které stojí na moderní třívrstvé architektuře, je přizpůsobeno pro obsluhu přes všechny běžné webové prohlížeče. Díky tomu redukuje dobu a náročnost školení uživatelů na minimum. Tato jednoduchost ovládání zaručuje, že WISPI je aplikace nejen intuitivní, ale i efektivní co se týče integrování jejího použití do existujících pracovních procesů. WISPI je navržena tak, aby byla otevřená integracím s dalšími aplikacemi, tato skutečnost umožňuje širokou škálu použití napříč různými systémy a platformami. [51]

9.3.1 Vzhled a hlavni funkce

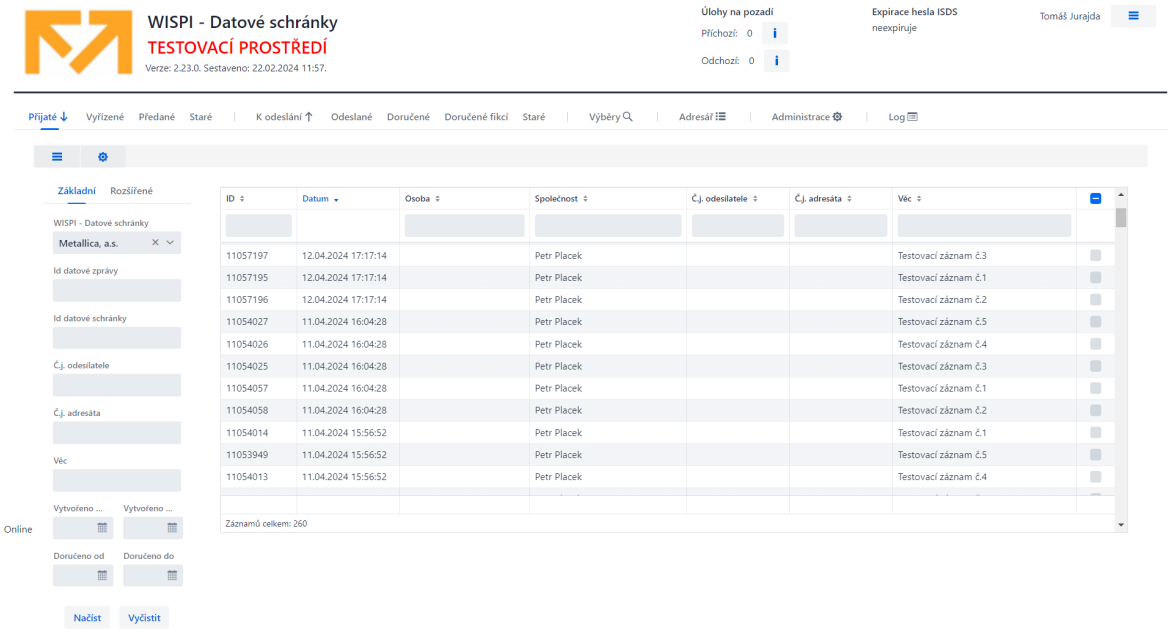
Samotný vzhled webové aplikace WISPI je navržen velmi čistě a propracovaně. Na stránce se nenachází mnoho funkcí, díky čemuž je struktura velice přehledná a uživatelsky přívětivá. Vzhledem patří mezi nejmodernější webové aplikace z předem zmíněných, a to je patrné zejména na použitých designových prvcích, které jsou aktuální a esteticky přitažlivé.

Aplikace je rozdělena do několika hlavních částí. První z nich představuje navigační panel, pomocí kterého uživatel může přepínat mezi různými stavy zpracování záznamů. Jako příklad lze uvést kartu „Přijaté“, kde jsou uživateli zobrazeny zprávy po jejich úspěšném přijetí datovou schránkou. Další stavy zahrnují karty jako „Vyřízené“, „Odeslané“ a „Doručené“, které zobrazují datové zprávy v různých fázích zpracování. V pozdější části tohoto panelu se nacházejí speciální funkce, jako je zobrazení aplikačního logu z vybraného časového rozmezí, který umožňuje uživateli hlouběji analyzovat a sledovat aktivitu chování aplikace.

Dalším v pořadí je panel představující funkce, které je možné uživatelem pro každý zvolený tab vykonávat. Nachází se pod navigačním panelem a jednotlivý tlačítka po zakliknutí zobrazí rozbalovací seznam možných akcí. Pokud se uživatel nachází v zá-

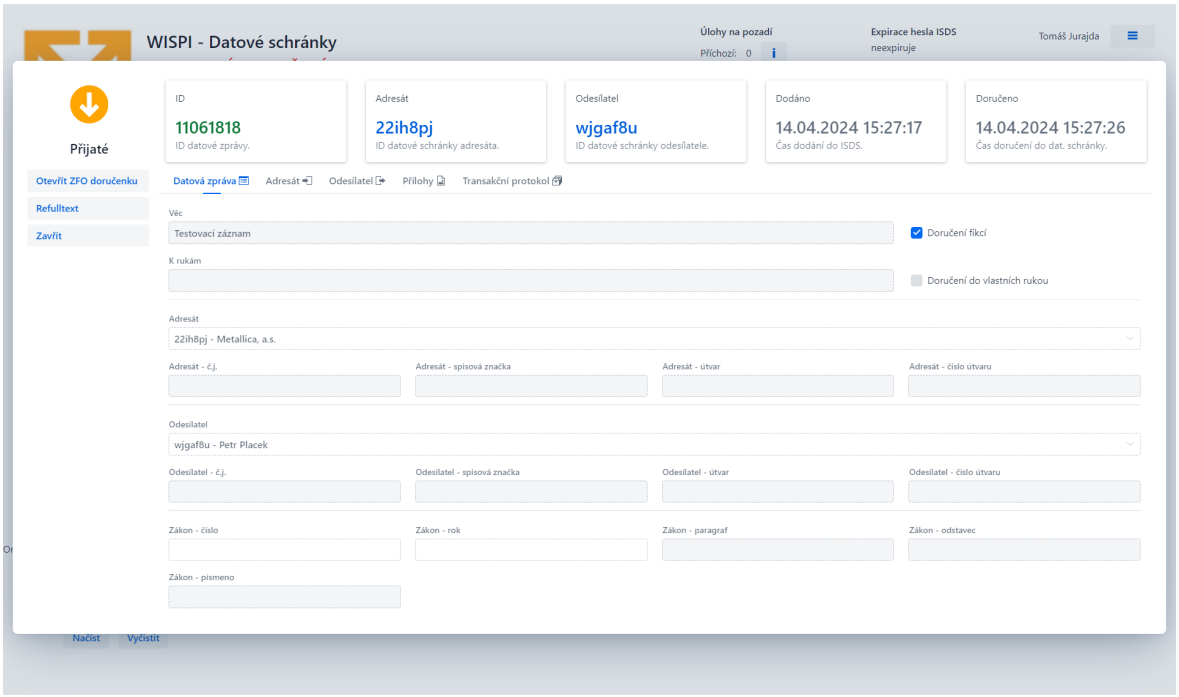
ložce „Přijaté“ může například zvolit akci přijmutí všech příchozích zpráv dané datové schránky, pouze však v případě že se v nějaké aktuálně nachází.

To nás přivádí k následujícímu prvku stránky, který obsluhuje funkcionalitu ohledně vybírání námi potřebné datové schránky nebo hledáním konkrétních záznamů dle jejich specifického jednoznačného identifikátoru ID.

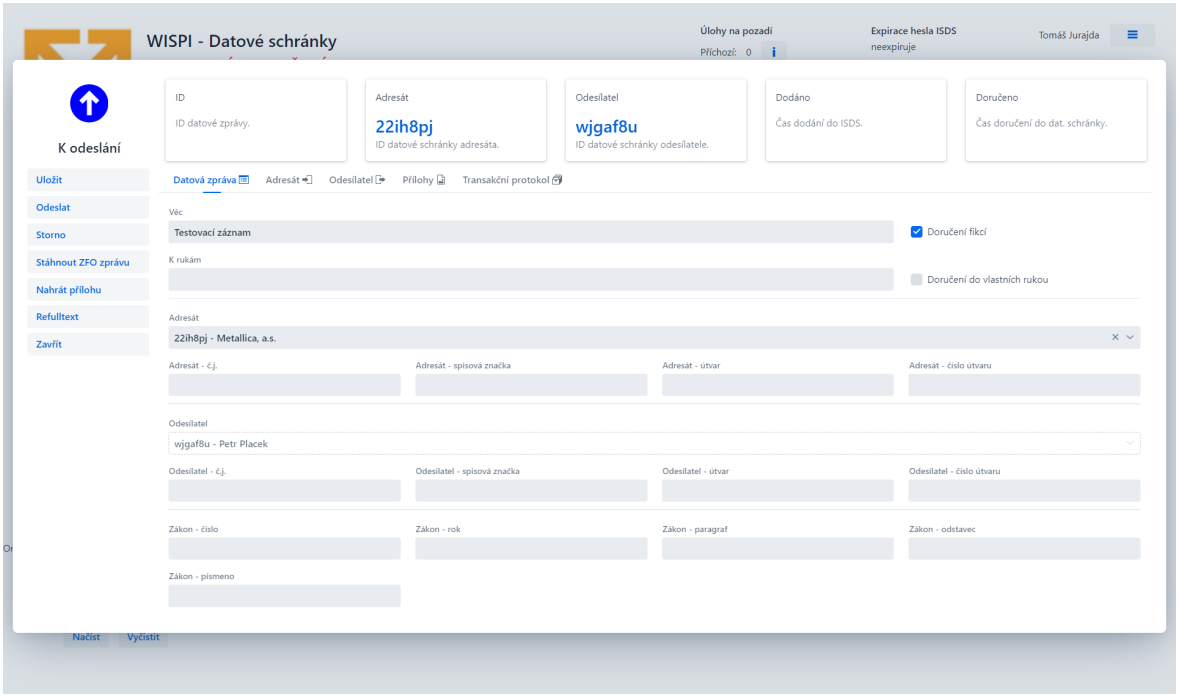


Obrázek 10 Ukázka vzhledu aplikace WISPI

Poslední z hlavních částí této webové aplikace tvoří tabulka, která obsahuje konkrétní záznamy. Ty jsou prezentovány svými základními údaji, které usnadňují jejich rychlé třídění. Uživatelé mají možnost každý záznam v tabulce otevřít a zobrazit tak všechna specifika, která byla do systému zadána. Pro jednodušší správu operací s daty je u každého záznamu v tabulce umístěn checkbox, díky kterému lze záznamy snadno označit a vykonávat s nimi další akce.



Obrázek 11 WISPI detail přijatého záznamu



Obrázek 12 WISPI detail záznamu k odeslání

10 ANALÝZA, NÁVRH A IMPLEMENTACE ŘEŠENÍ

V této kapitole se podrobně věnuji třem klíčovým fázím v procesu zajišťování kvality softwarových aplikací, které zahrnují analýzu požadavků, návrh řešení testování a implementaci řešení. Každý z těchto kroků je zásadní pro úspěšné provedení testování a zajistí, že systémy jsou spolehlivé a efektivní.

První fáze, analýza požadavků, se zaměřuje na důkladné pochopení a specifikaci toho, co má být testováno. Tento proces obnáší shromažďování důležitých informací od týmů zodpovědných za aplikace, identifikaci rizikových oblastí a klíčových funkcionalit, které je třeba ověřit.

Následuje fáze návrhu, kde se formují konkrétní testovací případy a strategie pro testování. Tato etapa rozhoduje o metodách testování, které budou použity k efektivnímu ověřování funkčnosti aplikací. Výběr správného přístupu a technologií je zásadní pro úspěch celého testování.

Poslední fází je podrobný popis mého přístupu k tvorbě testů pro specifické webové aplikace. Začal jsem s testováním jednodušších aplikací typu VadeMeCum a postupně jsem přešel k náročnějším projektům jako je software WISPI. V této části popisuji, jak jsem vnímal složitost tvorby testů a jaké metody jsem používal při jejich vytváření.

10.1 Analýza požadavků

Před samotnou tvorbou návrhu a implementace testů je nutné provedení podrobné analýzy testované aplikace. Ta byla prováděna v úzké spolupráci s nadřízenými a kolegy vývojáři, kteří mají dané aplikace na starost. Díky jejich důkladné znalosti dokumentů s požadavky na testované aplikace byl konzultován seznam specifických testovacích případů zaměřujících se na nejrizikovější oblasti, které jsou nejnáchylnější na vznik případných chyb v procesu postupného vývoje. Zaměřili jsme se tedy společnými silami na podrobné definování toho, co přesně má být testováno a jakým způsobem.

Dále bylo potřeba definovat jaký cíl má samotné testování mít. Zda je pro nás podstatné, aby aplikace běžela plynule pod velkou zátěží uživatelů, nebo aby fungovala správně dle definované specifikace požadavků na aplikaci. Díky však znalostem ohledně obsluhy aplikací víme, že pro naše potřeby je mnohem důležitější právě požadavek druhý, tedy správný chod. Bylo tedy rozhodnuto, že se využije automatického způsobu funkčního testování pro ověření hlavních funkcí jednotlivých testovaných webových aplikací. Jelikož se z finančního hlediska firmě nevyplatí testovat jejich celkovou funkcionalitu, tedy takzvaně end-to-end, byly dále v kapitole č. [10.2] navrženy testovací případy dle nejdůležitějších požadavků na každý testovaný software.

10.2 Návrh řešení testování

Dalším zásadním krokem k úspěšné implementaci automatizace testování webové aplikace je vypracování detailního návrhu řešení. V mém případě probíhala tvorba návrhu v úzké spolupráci s nadřízenými a kolegy, kteří testované aplikace spravují a tudíž mají důležité znalosti ohledně často se vyskytujících problémů při integraci nových verzí. Navržené testovací procedury nemají za úkol pokrýt ověřením veškeré funkcionality dané webové aplikace, nýbrž se spíše zaměřit na specificky určené funkce, které jsou pro chod aplikace zásadní a které se vyznačují svou vysokou úrovní poruchovosti.

V této kapitole jsou podrobně představeny konkrétní testovací procedury, které byly vyvinuty pro jednotlivé webové aplikace. Pro plné pochopení struktury a funkčnosti testovaných aplikací doporučuji navštívit kapitolu č. [9], kde jsou tyto aplikace podrobně popsány. Tato kapitola poskytne nezbytný základ pro lepší orientaci v testovacích procedurách a definuje jaké funkce jsou v aplikacích důležité.

10.2.1 Testovací procedury aplikace VadeMeCum

Testovací procedury pro prezentační aplikace, jako je VadeMeCum, jsou navrženy tak, aby definovaly a ověřily správnou funkčnost hlavních operací. Mezi ně patří funkce jako vyhledávání, zobrazování informací a přidávání záznamů do košíku. Vzhledem k tomu, že aplikace VadeMeCum jsou vyvíjeny na základě stejné šablony, byly testovací procedury pro tento typ aplikací sestaveny do jednoho sjednoceného seznamu. Některé scénáře se však specificky vztahují jen na aplikace, které obsahují upravené uživatelské rozhraní a funkce, odlišné od standardní šablony.

Výpis testovacích procedur pro aplikace VadeMeCum:

- **Ověření načtení aplikace**
- **Ověření zobrazení úvodní obrazovky**
- **Ověření funkčnosti přepínání mezi taby**
- **Ověření vyhledávání záznamů**
- **Ověření jednotlivých druhů zobrazení záznamu**
- **Ověření funkčnosti košíku**
- **Ověření funkcí při zobrazení detailu záznamu**

10.2.2 Testovací procedury aplikace ProMuzeum

Webová aplikace ProMuzeum slouží k odlišným účelům než zmíněné předchozí, proto se také zaměřujeme na návrh testovacích procedur odlišných funkcionalit, které prozkoumáváme podrobněji. Hlavním úkolem softwaru ProMuzeum je tvorba a správa archivních dat. V důsledku toho se testování této aplikace primárně soustředí na efektivitu a přesnost těchto klíčových funkcí, aby bylo zajištěno, že všechny operace s archivními daty jsou prováděny správně a bez chyb.

Výpis testovacích procedur pro aplikaci ProMuzeum:

- **Ověření načtení aplikace**
- **Ověření funkce přihlášení do aplikace**
- **Ověření zobrazení hlavní obrazovky**
- **Ověření funkcí toolbaru**
- **Ověření funkcí ohledně vytváření/mazaní záznamů**

10.2.3 Testovací procedury aplikace WISPI

Hlavním důvodem použití aplikace WISPI je pro zpravování dokumentů, konkrétní nejvíce používaná funkce je zasílání datových zpráv mezi jednotlivými datovými schránkami. Z toho důvodu se hlavní testovací procedury této aplikace právě na tuto funkcionalitu zaměřuje. Důkladné testování této funkce je nezbytné, protože jakékoliv selhání v procesu může vést k závažným problémům v komunikaci a operacích klientů, kteří aplikaci využívají. Z toho důvodu jsou testy navrženy tak, aby simulovaly různé reálné scénáře a zátěžové situace, které mohou během provozu nastat.

Výpis testovacích procedur pro aplikaci WISPI:

- **Ověření načtení aplikace**
- **Ověření funkce přihlášení do aplikace**
- **Ověření zobrazení hlavní obrazovky**
- **Ověření načtení záznamů dané datové schránky**
- **Ověření funkce vytváření nových záznamů**
- **Ověření funkcionality ohledně zasílání datových zpráv**

10.3 Návrh vzoru pro strukturu testů - POM

V následujícím kroku celkové tvorby testů byl vytvořen návrhový vzor pro testování jednotlivých komponent webové aplikace, který má za cíl zjednodušit správu vytvořených testů. Byl použit konkrétní vzor s názvem Page Object Model, zkráceně POM, který udává pravidla pro definování a zacházení s testovanými elementy stránek. Své využití uplatní výhradně při testování komplexnějších aplikací, které se dělí na mnoho obrazovek nebo stránek, ale lze jej použít i na aplikace jednodušší.

Page Object Model říká, že pro každou stránku testované aplikace musíme vytvořit separátní třídu, ve které definujeme jednotlivé elementy, které se na stránce nacházejí a metody pro jejich manipulaci. Tento přístup pomáhá minimalizovat duplicitu kódu a zvyšuje udržitelnost testovacích případů. [52]

Výhody Page Object Modelu:

- **Snadná údržba** - POM se ukazuje jako mimořádně užitečný při změnách v uživatelském rozhraní, kde se zamění definice elementu za jinou. V takových případech POM usnadňuje lokalizaci stránky nebo obrazovky, která potřebuje úpravu. Jak už bylo řečeno každá obrazovka má svůj vlastní soubor se třídou pro definici prvků na ní se vyskytujících, což zjednodušuje identifikaci a provádění změn.
- **Opakované využití kódu** - Díky nezávislosti stránek může být kód použitý pro testování jedné obrazovky snadno přizpůsoben a použit v jiném testovacím případě. To šetří čas a úsilí, které by jinak bylo vynaloženo na opakované psaní kódu.
- **Čitelnost a spolehlivost skriptů** - Nezávislé třídy pro každou stránku zlepšují čitelnost skriptů a usnadňují identifikaci akcí provedených na konkrétních obrazovkách. Pokud je třeba provést úpravy v konkrétní části kódu, lze ji provést efektivně bez rizika ovlivnění ostatních tříd a jejich elementů.

Na následující straně se nachází příklady zdrojových kódů. Nejprve se v kódů č. (24) nachází testovací třída *MyTest*, která nepoužívá návrhového vzoru Page Object Model. Můžeme si všimnout, že v testu *loadPageTest* jsou napřímo uvedeny selektory k daným prvkům. Naopak v kódu č.(25) se POM využívá a tudíž zde vidíme prvně uvedenou třídu *MainPage*, která představuje hlavní testovanou stránku a v ní metody vracející potřebné elementy. Následně v testovací třídě *MyTest* jsou pouze tyto metody volány.

```
public class MyTest {

    @BeforeClass
    public void setupAll() {
        //otevreni stranky
        open("https://example.com" );
    }

    @Test
    public void loadPageTest() {
        //kontrola zobrazeni loga stranky
        $x("//*[@id='logoImg']").shouldBe(visible);

        //kontrola zobrazeni uvodniho nadpisu
        $("div[id='hpNews'] h1").shouldBe(visible);
    }
}
```

Zdrojový kód 24 Ukázka testu loadPageTest bez využití POM

```
public class MainPage {

    //metoda vraci url testovane aplikace
    public String getAppURL() {
        return "https://example.com";
    }

    //metoda vraci nadpis stranky
    public SelenideElement getUvodText() {
        return $("div[id='hpNews'] h1");
    }

    //metoda vraci logo stranky
    public SelenideElement getLogo() {
        return $x("//*[@id='logoImg']");
    }
}


public class MyTest {

    MainPage mainPage = new MainPage();

    @BeforeClass
    public void setupAll() {
        //otevreni stranky
        open(mainPage.getAppURL());
    }

    @Test
    public void loadPageTest() {
        //kontrola zobrazeni loga stranky
        mainPage.getLogo().shouldBe(visible);

        //kontrola zobrazeni uvodniho nadpisu
        mainPage.getUvodText().shouldBe(visible);
    }
}
```

Zdrojový kód 25 Ukázka testu loadPageTest s využitím POM

10.4 Postup tvorby testů

Proces tvorby testovacích případů byl zpočátku složitý, jelikož jsem neměl žádné předchozí zkušenosti s webovým testováním, když mi byl nadřízenými přidělen tento úkol, avšak přes to jsem se ho beze strachu chopil. Prvním krokem bylo tedy provést důkladnou rešerši na téma testování webových aplikací. Ta zahrnovala výběr testovacího nástroje, který by mi nejlépe vyhovoval, naučení se syntaxi tohoto nástroje a pochopení, jak pomocí něj spouštět a vytvářet testy v mém používaném vývojovém prostředí IntelliJ, který je podrobněji představen v kapitole č. [8.2]. Tento přístup mi umožnil postupně získat potřebné dovednosti a zlepšit můj přístup k testování, neboli zvýšit mou efektivitu ve tvorbě samotných testů.

10.4.1 Začátky testování aplikace VadeMeCum

Webové aplikace typu VadeMeCum, které jsou podrobněji popsány v kapitole č. [9.1], byly prvními, které jsem měl za úkol otestovat podle připravených scénářů. Z počátku, jak už to u všeho bohužel bývá, mi tvorba testů zabrala hodně času a největším oříškem pro mě bylo definování metod pro hledání elementů na testované webové aplikaci, které jsou klíčové k provádění testů. Jelikož jsem však jako první testoval aplikace VadeMeCum, které obsahují jednoduché funkcionality hledání a zobrazovaní, šla mi práce poměrně dobrým tempem. Začal jsem na základních úkonech manipulace se stránkou jako její otevření, vyhledání elementu nadpisu a dále například klikání a přepínání mezi odlišnými obrazovkami (taby) až jsem se postupně propracoval ke samotným pokročilým funkcionalitám webu jako vyhledávání, přidávání do košíku a kontroly možnosti zobrazení záznamu.

Po úspěšném implementování testů hlavních funkcionalit první webové aplikace VadeMeCum jsem se pustil do práce na další aplikaci stejného typu. Zde bylo mým primárním úkolem zjistit, jak náročné bude, tyto vytvořené testy v nástroji Selenide pro první aplikaci, přepsat do jiné. Jednalo se totiž také o aplikaci typu VadeMeCum, tedy prezentačního portálu pro muzejní instituty, které jsou vytvářeny pomocí šablony s velmi obdobnými funkcemi a uživatelským rozhraním. Bylo tedy zkoumáno, jak složité bude testy přeformátovat pro chod na aplikaci druhé. Testovací případy nebylo nutné téměř vůbec upravovat až na některé výjimky. Úpravám podléhaly pouze třídy spravující vyhledávání elementů dané obrazovky. Konkrétně se tedy jednalo o přepsání velkého množství selektorů, jelikož se změnila struktura HTML kódu a některé identifikátory hledaných elementů. To ve výsledku bylo poměrně pracné a z celkového hlediska bylo potřeba přeformátovat více než 50% celkového kódu pro zprovoznění stejných testů na aplikaci podobného rázu. Nadřízení seznámeni s tímto výsledkem nebyli mnoho spokojeni, a tak mi byla zadána tvorba testů pro webové aplikace jiné struktury.

10.4.2 Přechod na pokročilejší aplikaci ProMuzeum

Jako další úkol v rámci mého projektu jsem se zaměřil na vytvoření testovacích případů pro webovou aplikaci ProMuzeum, konkrétně pro její variantu Fotoarchiv. Detailní popis této aplikace a její funkcionalit je k nalezení v samostatné kapitole č. [9.2], kde jsou všechny informace podrobně vyložené. Hlavní úlohou aplikace Fotoarchiv je umožnit efektivní vytváření, úpravy a správu archivních záznamů pro muzejní institut. Tato aplikace je navržena tak, aby výrazně ulehčila digitalizační práce archivním pracovníkům

ProMuzeum disponuje širokou škálou funkcionalit, které výrazně převyšují možnosti předchozí aplikace VadeMeCum. Z tohoto důvodu musel vzniknout seznam hlavních funkcí, které je potřeba důkladně otestovat. Ten mi byl předat nadřízenými, kteří mají s aplikací větší zkušenosti a tudíž pravomoc tyto podstatné funkce určit. Mezi nejdůležitější funkce, na které jsem se zaměřil patří například ověření procesu přihlášení do systému, schopnost aplikace přepínat mezi režimem pro čtení a zápis záznamů, a možnosti pro vytváření a mazání záznamů. Díky předchozím zkušenostem z testování aplikace VadeMeCum jsem mohl pracovat efektivně a svižně. Přesto jsem občas narazil na výzvy, jako bylo složitější časování určitých kroků testovacích nebo lokalizace specifických prvků na stránce, které byly náročnější na implementaci a tudíž i zabíraly více času.

Výsledkem je tedy implementace automatizovaných testů, který byly zvolené dle předem sestaveného seznamu hlavních funkcí aplikace. Cílem bylo zjistit zda tyto funkce bude vůbec možno testovat a jakým způsobem. Dalším zkoumaným prvkem byl rozsah použití mnou vybraného nástroje pro práci Selenide, které se ve výsledku ukázalo jako velmi příhodné pro mé potřeby tetování, jelikož práce s ním je jednoduchá a efektivní.

10.4.3 Finální testování aplikace WISPI

Poslední a nejtěžší pro mě byla tvorba automatizovaných testů pro webovou aplikaci spisové služby WISPI. Ve zkratce se jedná o software pro zpracování a následnou manipulaci různých datových dokumentů a jejich příloh, podrobné informace můžete nalézt v kapitole č. [9.3]. WISPI se jak vzhledem tak svými funkcemi nepodobá žádné předem zmíněné aplikaci. Obsahuje moderní uživatelské rozhraní přívětivé pro své uživatele.

Tato webová aplikace je vyvíjena a spravována na vedlejším oddělení, takže jsem s ní neměl žádné předchozí zkušenosti. Byla vybrána kolegou z tohoto oddělení z důvodu její vysoké náchylnosti na vznik chyb při zavádění nových verzí do provozu. Vznikl tedy seznam nejdůležitějších funkcionalit daného softwaru, které jsou podstatné pro základní práci uživatele. Celkově však tou nejpodstatnější je kompletní průběh životního cyklu datových zpráv. Ten probíhá od jejich vytvoření přes odeslání až po přijmutí, které je

provedeno u adresované datové schránky. Hlavním terčem bylo tedy ověření správné funkcionality tohoto procesu a na ni jsem se také nejvíce při testování zaměřil.

Celková tvorba testů pro software WISPI mi zabrala hodně času a byla oproti předchozím dosti náročná. Jednalo se tak hlavně kvůli vytváření složitějších selektorů pro potřebné elementy v HTML struktuře stránky. Aplikace je totiž moderního rázu a používá tak novější přístupy k vytváření těla webu. Nově jsem se zde setkal z dosud pro mě neznámým objektem s názvem „ShadowRoot“[1)], který má za úkol zapouzdřit určitou část HTML kódu neboli ji skrýt před okolním světem. Jeho hlavním úkolem je tedy skrýt tuto část před zbytkem stránky, čímž následně vytváří takzvaný „Shadow DOM“. Tato technika umožňuje izolovat styl a skripty v jedné části stránky, aniž by došlo k jejich ovlivnění nebo konfliktu s ostatními částmi dokumentu. Elementy uvnitř této skryté struktury nelze totiž nalézt obvyklým způsobem. Dosti jsem s tímto problémem bojoval, ale nakonec se mi podařila dohledat metoda implementovaná přímo v nástroji Selenide, která byla vytvořena konkrétně pro řešení tohoto problému. Umožňovala prohledávat konkrétní ShadowRoot objekt, pomocí zadaní atributu hledaného elementu a rodiče, ve kterém se skrytá struktura nachází. Díky jeho použití jsem dosáhnul zdárného konce ve vyhledání potřebných prvků stránky a mohl jsem se tak ponořit k tvorbě samotných testů.

Práce na implementaci testů pokračovala svižným tempem. To bylo zapříčiněno hlavně díky zkušenostem, které jsem získal při používání nástroje Selenide během testování předchozích webových aplikací. Vyskytly se jen menší problémy s načasováním určitých kroků testů, ale to nezabrzdilo celkový průběh práce, který směřoval k úspěšnému dokončení. Výsledkem jsou automatizované testy pro webovou aplikaci WISPI, které se ukázaly jako velmi užitečné pro mé kolegy zodpovědné za správu verzí této aplikace.

[1)] ShadowRoot - podrobnější informace viz. `https://developer.mozilla.org/en-US/docs/Web/API/Web_components/Using_shadow_DOM`

11 PŘEHLED VYTVOŘENÝCH TESTOVACÍCH PŘÍPADŮ

V této kapitole jsou prezentovány testovací případy, které jsem vytvořil pro automatizované testování vybraných webových aplikací ve firmě, kde pracuji. Kapitola je strukturovaná do několika podkapitol, přičemž každá se věnuje konkrétní testované aplikaci. V těchto podkapitolách poskytuji podrobný popis každého testovacího případu. Je zde také umístěna tabulka, která specifikuje detailní údaje o testovacích případech jako jsou například název, popis, předpoklady, výsledek nebo jednotlivé testovací kroky případu. Z důvodu bezpečnosti jsou u testovacích případů kontrolujících přihlášení uvedeny smyšlené přihlašovací údaje.

11.1 Testovací případy aplikace VadeMeCum

Tato podkapitola se zaměřuje na přehled mnou vytvořených testovacích případů pro aplikace typu VadeMeCum. Pro potřeby vyšší přehlednosti zde nejsou uváděny všechny případy, nýbrž pouze vybrané. A to z důvodu velké podobnosti aplikací VadeMeCum, jejichž základ je postaven na stejné šabloně, viz. kapitola č. [9.1]. Kapitola dále obsahuje konkrétní testovací případy se stručným popisem kroků jejich funkčnosti.

11.1.1 Testovací případy Národní galerie v Praze

Podkapitola obsahující vybrané testovací případy z webové aplikace Národní galerie v Praze.

- **loadPageTest -** Jedná se o nejzákladnější testovací případ, který má za úkol ověřit úspěšné načtení a zobrazení hlavní obrazovky dané aplikace. Jsou specificky tvořeny pro každý testovaný web zvlášť. Pro případ Národní galerie města Prahy zkontroluje načtení tak, že ověří zda je zobrazeno logo tohoto institutu, a také že je viditelná první ukázka záznamu v podobě obrazu na hlavní stránce.
- **vyhledavaniTest -** Dalším z hlavních testů vyskytujících se v každé aplikaci VadeMeCum je kontrola funkčnosti vyhledávání. Jelikož se každá aplikace v této implementaci drobně liší jsou tyto testovací případy různě složité. Celkově se však skládá z kroků jako selekce specifických fondů či sbírek pomocí rozšířeného hledání nebo samotné ověření nalezení hledaného záznamu.
- **zobrazeniTest -** V tomto testovacím případu se snažíme zkontrolovat funkčnost přepínání mezi různými druhy zobrazení nalezených záznamů. Jedná se o identickou funkcionalitu napříč celkovou implementací aplikací VadeMeCum.
- **detailTest -** DetailTest se zaměřuje na ošetření možné manipulace při zobrazení detailu záznamu. Konkrétně se jedná o funkce pro přesun na další záznam

v pořadí nebo o kontrolu přesunu na originální vzhled obrázku pomocí nástroje Zoomify.

- **kosikTest -** Tento případ má za úkol zkontrolovat veškerou funkcionalitu ohledně manipulace s košíkem aplikace. Zde je ošetřováno například přidávání záznamů do košíku nebo jejich následné odstranění.
- **rezervaceAOdeslaniOdkazuTest -** Dále je kontrolováno zobrazení okna pro možnou rezervaci záznamů v archivu pro následnou studii v zařízeních jako jsou například badatelny.

Testovací sada : TS-NGP_001 - Ověření funkcí hlavní stránky				
Testovací případ				
ID	TC_001	**Autor**	Marek Trefil	
Název	loadPageTest	**Datum**	20.10.2023	
Předpoklady	spuštěná aplikace	**Verze**	1.0	
Popis	Testovací případ má za úkol ověřit funkci správného načtení hlavní stránky webové aplikace			
Jednotlivé testovací kroky :				
Č. kroku	**Popis činnosti**	**Očekávaný výsledek**	**Aktuální výsledek**	**Výsledný status (PASS\FAIL)**
1	ověř výskyt loga	zobrazené logo	zobrazené logo	PASS
2	ověř výskyt prvního záznamu	zobrazený záznam	zobrazený zaznam	PASS

Tabulka 1 Ukázka testovacího případu kontroly načtení obrazovky

11.1.2 Testovací případy Ostravského muzea

Podkapitola obsahující vybrané testovací případy z webové aplikace Ostravského muzea. Jedná se o výpis testů, které přibyly při testování tohoto typu VadeMecum nebo bylo nutno je dle potřeby aplikace pozměnit.

- **katalogTest -** KatalogTest má za úkol se zaměřit na ověření přepnutí na tab Katalogu, kde požaduje k úspěchu jeho zobrazení.
- **uvodTest -** Další kontrola zobrazení tabu tentokrát však úvodního. Jelikož uvodTest následuje přímo po katalogTest, tak se ve výsledku jedná o kontrolu přesunu z tabu katalogu zpět na úvodní obrazovku.

- **katalogSearchTest -** Upravený test vyhledávání pro aplikaci Ostravského muzea. Obsahuje také navíc zrušení vyhledávání, které uživatele přesune zpět na úvodní obrazovku.
- **instituteSbirkyVyberTest -** Rozšířené vybrání specifických sbírek „Keramika“ a následné vrácení zpět na úvodní obrazovku.
- **detailTest -** Jedná se o přeformování metody pro kontrolu detailu pro potřeby testu této aplikace. Testuje veškeré funkcionality detailu jako přepínání mezi záznamy, zobrazení originálu v Zoomify, vkládání do košíku a následné otevření okna pro zobrazení permalinku na daný záznam.

11.1.3 Testovací případy Středočeského muzea v Roztokách

Podkapitola obsahující vybrané testovací případy z webové aplikace Středočeského muzea v Roztokách.

- **stredoceskeTest -** Testovací případ pro kontrolu přepnutí tabu pro zobrazení stránky o Středočeském muzeu.
- **sladeckovoTest -** Další případ kontroly změny tabu, tentokrát však na stránku o Sládečkově vlastivědném muzeu.

11.1.4 Testovací případy Valašského muzea v přírodě

Podkapitola obsahující vybrané testovací případy z webové aplikace Valašského muzea v přírodě.

- **podSbirkyFondyTest -** Tento testovací případ provádí ověření funkčnosti přepínaní mezi jednotlivými sbírkami a fondy, které se nachází na hlavní stránce při zapnutí.
- **zoomifyPermaLinkTest -** Pro potřeby aplikace přeformátovaný test pro kontrolu otevření originálního vzhledu obrázku a pro zobrazení permalinku daného záznamu.

11.2 Testovací případy aplikace ProMuzeum

Následující testovací případy se vztahují pro aplikaci ProMuzeum konkrétní implementaci FotoArchiv. Jde o aplikace pro práci, editaci a manipulaci s archivními záznamy, více v kapitole č. [9.2]. Testovací případy jsou rozděleny do testovacích sad podle části aplikace, na kterou se při ověřování zaměřují.

11.2.1 Kontrola obrazovky LoginPage

Podkapitola představující testovací případy pro kontrolu přihlašovací obrazovky, která je uživateli zobrazena po příchodu do aplikace. Bez přihlášení uživatelským jménem a heslem není možné pokračovat dále, proto se jedná o velmi podstatný test.

- **errorMessageWhileLogin -** Tento testovací případ má za úkol selhat. Jedná se totiž o test neúspěchu. Ověřuje vyskočení chybové hlášky v podobě okna, když uživatel aplikace nezadá žádné údaje do pole jméno a heslo.
- **errorMessageWhileWrongUser -** Dalším přihlašovacím testem je kontrola zobrazení chybového okna po zadání uživatele s nesprávnými údaji.
- **loginWithRealUser -** Posledním testovacím případem přihlašovací obrazovky je přihlášení s platnými údaji, které má vést k zobrazení hlavní stránky pro práci s archivními údaji.

11.2.2 Kontrola funkcí toolbaru

Tato testovací sada má za úkol ověřit vybrané funkce z toolbaru aplikace. Níže nalezneme testovací případy použité pro kontrolu funkčnosti těchto nástrojů.

- **logoutAndLoginBackTest -** První ověřovanou funkcí je odhlášení, které se nachází jako poslední prvek toolbaru, avšak pro uživatel velmi důležitý. Provede odhlášení z aplikace, které je provedeno úspěšně pokud jsme zpět na přihlašovací obrazovce.
- **savingModificationsTest -** Následující, velmi podstatná funkce pro práci se záznamy, je jejich samotné ukládání. To má za úkol prověřit tento testovací případ. Provedeme tedy modifikaci zvoleného záznamu a po uložení musí daný údaj stále v záznamu setrvat.
- **deletingModificationsTest -** Ruku v ruce jde tento testovací případ s předchozím, jelikož se v tomto případu ověřuje odstranění předem provedené modifikace.
- **readOnlyModeTest -** Další důležitou funkcí toolbaru pro uživatele je přepínání mezi módy k zápisu a ke čtení. Tento testovací případ zajišťuje přechod do pouze čtecího módu.
- **writeOnlyModeTest -** Kontrola spolupracující s předchozí zmíněnou, tentokrát se ověřuje přechod do módu k zápisu uživatelem, který povolí manipulaci se záznamy v daném archivu.

11.2.3 Manipulace se záznamy

Testovací případy této sady mají za úkol ošetřit další velmi podstatnou funkcionalitu, a to manipulaci se záznamy. Konkrétně se testují funkce na přidávání nových záznamů a jejich následné vymazání.

- **addRecordsTest -** Jedná se o komplexnější test než předchozí vytvořené. Hlavním prvkem je cyklus s proměnlivým počtem chodů, jehož úkolem je simulovat přidávání nových záznamů a následná kontrola správného uložení záznamu a jeho zobrazení.
- **removeRecordsTest -** Tento testovací případ následuje ihned po předchozím a má za úkol odstranit vytvořené záznamy z daného archivu. Funguje na stejném principu, který využívá cyklu pro opakované provádění činnosti odstranění.

11.3 Testovací případy aplikace WISPI

Poslední mnou testovanou webovou aplikací je spisová služka WISPI. Hlavním úkolem aplikace je zpracovávaní datových souborů, více o WISPI naleznete v kapitole č. [9.3]. Testovací případy jsou rozděleny do dvou sad, které testují nejdůležitější aspekty použití tohoto softwaru.

11.3.1 Základní testy

Testovací případy této sady spočívají v ověření funkčnosti základních operací jako je zobrazení stránky nebo načtení údajů konkrétní datové schránky.

- **loadPageTest -** Jedná se o klasický případ ověření správnosti načtení hlavní obrazovky po úspěšném přihlášení do aplikace. To provádí pro upřesnění v tomto příkladu pomocí zjištění, zda daný element nadpisu obsahuje předem definovaný text.
- **loadDatSchrankaTest -** Tento testovací případ kontroluje zobrazení přijatých záznamů v načítané datové schránce. Test předpokládá, že se ve schránce nachází jeden a více dokumentů.

11.3.2 Manipulace se záznamy

Následující testovací sada se zaměřuje na ověření funkcionalit ohledně samotné manipulace se záznamy. Jedná se konkrétně o činnosti jako vytváření nových záznamů, odesílání na datové schránky, přijmutí zpráv nebo kontrola doručení. Testovací případy jsou celkově velmi komplexní a skládají se z mnoha kroků, jelikož je nutné k otestovaní jednoho provést v aplikaci mnoho činností.

- **createNewSenderTest -** První testovací případ z této sady má za úkol vytvoření nového záznamu k odeslání a následné ověření jeho doručení u adresáta. Testuje pouze zaslání jednoho specifického záznamu, a proto se jedná o jednodušší případ z této sady.

- **vyrizeniZaznamuTest -** Následující test je spojen z předchozím, jelikož dále po přijetí předem vytvořeného záznamu ověřuje funkci jeho vyřízení. To vyžaduje vybrání požadovaného záznamu k vyřízení ze seznamu a následné spuštění funkce pro spuštění vyřízení.

- **createMultipleFilesToSendTest -** Jedná se o dosud nejkomplexnější vytvořený testovací případ, který se skládá z mnoho prováděných kroků. Jeho hlavní činností je vytvoření libovolného počtu záznamů pomocí cyklu a jejich následné odeslání adresované datové schránce. Dále ověřuje jejich správné přijetí u adresáta. Celkově je tento test koncipovaný jako zátěžové testování, kde zkoumáme časovou náročnost nutnou pro vykonání tohoto cyklu životního cyklu záznamu nebo některé jeho části.

- **doruceneCheckTest -** DoruceneCheckTest je další z případů, kdy je test závislí na předchůdci. Má totiž za úkol ověřit správnost funkce kontroly doručení, která se vykonává po přijetí zprávy u adresáta. Následně jsou tedy u datové schránky odesílatele v tabu „Doručené“ zobrazeny veškeré správně přijaté záznamy.

- **tryCreatingMessageWithoutKnownAdresatIdTest -** Posledním testovacím případem této webové aplikace je kontrola funkce vyhledávání adresáta při vytváření nového záznamu. Ten může být vyhledám pomocí jednoznačné Id nebo jména a vždy se nám v reálném čase zobrazí výsledky v rolovacím seznamu. Specifické zaměření toho případu je na vyhledání pomocí části jména cílové datové schránky místo využití jeho Id. Je totiž důležité, aby si uživatel aplikace tyto identifikátory nebyl nucena pamatovat.

Testovací sada : TS-NGP_001 - Ověření funkcí hlavní stránky				
Testovací případ				
ID	TC_002	**Autor**	Marek Trefil	
Název	vyhledavaniTest	**Datum**	20.10.2023	
Předpoklady	spuštěná aplikace	**Verze**	1.0	
Popis	Testovací případ má za úkol ověřit funkci vyhledání záznamu			
Jednotlivé testovací kroky :				
Č. kroku	**Popis činnosti**	**Očekávaný výsledek**	**Aktuální výsledek**	**Výsledný status (PASS\FAIL)**
1	ověření výskytu elementu input pro hledání	zobrazený input	zobrazený input	PASS
2	kliknutí na tlačítko pro rozšířené hledání	otevření rozšířeného hledání	otevření rozšířeného hledání	PASS
3	ověření výskytu tlačítka "Pouze s obrázkem"	zobrazení tlačítka	zobrazení tlačítka	PASS
4	kliknutí na tlačítko "Pouze s obrázkem"	označení tlačítka jako zakliknutého	označení tlačítka jako zakliknutého	PASS
5	kliknutí na tlačítko "Sbírka umění Asie"	označení tlačítka jako zakliknutého	označení tlačítka jako zakliknutého	PASS
6	kliknutí na tlačítko "Asie - malba a grafika"	označení tlačítka jako zakliknutého	označení tlačítka jako zakliknutého	PASS
7	ověření výskytu tlačítka pro hledání	zobrazení tlačítka	zobrazení tlačítka	PASS
8	kliknutí na tlačítko pro hledání	provedení hledání	provedení hledání	PASS
9	ověření nalezeného záznamu	výskyt záznamu	výskyt záznamu	PASS
10	ověření názvu záznamu obsahující text "Krevety"	správný název záznamu	správný název záznamu	PASS
11	kliknutí na tlačítko pro resetování hledání	vymazání paramterů pro hledání	vymazání paramterů hledání	PASS
12	ověření výskytu zpět na původním záznamu před hledáním	zobrazení prvního záznamu	zobrazení prvního záznamu	PASS

Tabulka 2 Ukázka testovacího případu ověření funkce vyhledávání

12 PREZENTACE VÝSLEDKŮ TESTŮ

Reportování neboli podávání hlášení o výsledcích testů, je jednou z dalších důležitých součástí procesu automatizovaného testování. Poskytuje přesné a podrobné údaje o výsledcích testování a umožňuje týmům informovaně rozhodovat o dalších krocích ve vývoji softwaru. Vybrání kvalitního reportovacího nástroje je nezbytné pro identifikaci problémů, zlepšení komunikace v týmu a úsporu času, který by jinak musel být vynaložen na manuální analýzy jednotlivých testů.

Přehledné reporty umožňují rychle identifikovat kritické chyby v rané fázi vývojového cyklu. Tím přispívají k vyšší efektivitě analýzy a poskytují důležitou zpětnou vazbu pro řízení kvality produktu. Jasná reportáž také podporuje sledování pokroku produktu, jenž je nezbytné pro určení, zda je produkt připraven na uvedení na trh.

Při výběru nástroje pro reportování je nezbytné zvážit následující faktory:

- **Funkčnost a vlastnosti:** Nástroj by měl poskytovat relevantní a podrobné výsledky, které jsou užitečné pro tým.
- **Jednoduchost použití:** Intuitivní rozhraní snižuje náročnost na zaškolení a usnadňuje každodenní práci s nástrojem.
- **Integrace:** Flexibilita v integraci s různými nástroji pro automatizaci testování.
- **Cena:** Nákladově efektivní řešení, které pokrývá různé potřeby a požadavky organizace.

Následující kapitoly popisují příklady jakým způsobem je v mých vytvořených testech zaimplementováno jejich reportování. Vetšinou se jedná o pouze využití už existujících nástrojů, které jsou schopny samy vygenerovat výsledné hlášení o celém průběhu jednotlivých testovacích případů. Je tedy pouze na nás, které využijeme a které se nám bude zdát nejpřívětivější pro naše aktuální potřeby. [53] [54]

12.1 Reportování pomocí nástroje Selenide

Hlavní vývojář nástroje Selenide tvrdí, že pro nejobyčejnější hlášení výsledků testů bohatě stačí výpis z buildu aplikace pomocí nástrojů Gradle nebo Maven. [28] Ty poskytují detailní výpis chyby, ukazují kde nastala a jaký byl například očekávaný stav daného elementu. Tento způsob reportování však není příliš přívětivý pro uživatele bez technických znalostí, protože vygenerovaný kód může být složitý a pro laiky těžko srozumitelný. Hodí se tedy spíše pro samotné programátory testů, kteří již přesně dle daného výpisu vědí, kde se problém nachází.

```
[INFO]
[INFO] Results:
[INFO]
[ERROR] Failures:
[ERROR]   MainPageTest.loadPageTest:31 Element not found {By.xpath: //*[@id='logoImage']}
Expected: visible
Screenshot: file:/N:/user/src-git/tests/ngp/build/reports/tests/1714063907339.0.png
Page source: file:/N:/user/src-git/tests/ngp/build/reports/tests/1714063907339.0.html
Timeout: 4 s.
Caused by: NoSuchElementException: no such element: Unable to locate element: {"method":"xpath","selector":"//*[@id='logoImage']"}
[INFO]
[ERROR] Tests run: 4, Failures: 1, Errors: 0, Skipped: 0
[INFO]
[ERROR] There are test failures.
```

Obrázek 13 Ukázka části výpisu buildu nástroje Maven při chybě

Na obrázku č. (13) můžeme vidět ukázku vypsání chybové hlášky v reportu u samotného buildu aplikace. Vidíme zde výpis elementů, u kterých nastala chyba. Konkrétně v tomto případě nebyl nalezen element nesoucí obrázek, který nesplnil ověření, jelikož měl být dle nadefinovaného testovacího případu „visible“, tedy viditelný na aktuálně se nacházející stránce aplikace. Můžeme si dále v kódu všimnou definice cesty, kde se nachází fotografie aktuálního stavu aplikace, kterou samotný nástroj pro vytváření testů Selenide automaticky při chybě generuje. Hned pod fotografií se vyskytuje další cesta tentokrát však k zaznamenanému kódu HTML testované stránky. Zda tyto dva soubory má Selenide automaticky generovat či nikoliv a mnoho dalšího, je možné nastavit v jeho rozsáhlé konfiguraci. [55]

12.2 Reportovací třída TextReport

Pokud nám ovšem pouhé vypsání průběhu buildu do konzole nestačí nebo nevyhovuje našim požadavkům, můžeme zvolit možnosti jiné. Další způsob je sice podobný předešlému, jelikož se také jedná o text v konzoli vývojového prostředí, ale je vzhledově upraven, aby byl přehlednější. Je jím pomocná třída s názvem TextReport implementovaná přímo vývojáři nástroje Selenide. Jedná se o jednoduchý nástroj na vypsání reportů jednotlivých testovacích případů. Než začneme TextReport používat, je nutné jej definovat jako posluchače pro třídu, kterou chceme testovat. To zajišťuje použití anotace frameworku TestNG, který byl již v práci představen v kapitole č. [8.4]. Příklad jak se taková anotace k třídě přidává vidíme na následujícím zdrojovém kódu č. (26).

```
import com.codeborne.selenide.testng.TextReport;

@Listeners({TextReport.class})
public class MainPageTest {
  //...
  //...
  //...
}
```

Zdrojový kód 26 Přidání TextReport hlášení ke třídě

Reportovací nástroj TextReport poskytuje detailní popis činnosti každé metody v testované třídě. Tyto metody představují jednotlivé testovací případy. Jak můžeme vidět na obrázku č. (14), výsledky jsou zobrazeny v tabulce, kde každý řádek odpovídá jednomu kroku testu. Údaje v tabulce jsou organizovány do čtyř sloupců: [55]

- **Element:** Element představuje konkrétní prvek na stránce, se kterým je v testu manipulováno.
- **Subject:** Subject obsahuje činnost, kterou s daným elementem provádíme.
- **Status:** Status udává, zda testovací krok proběhl úspěšně (PASS) nebo selhal (FAIL).
- **ms.** Zobrazuje dobu trvání testovacího kroku v milisekundách.

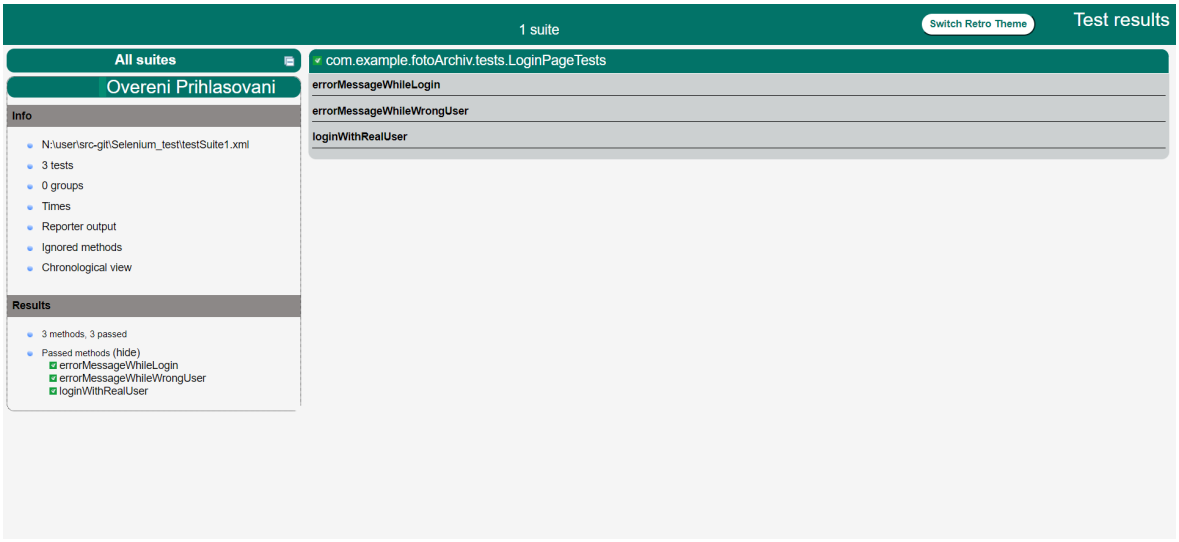
```
[main] INFO com.codeborne.selenide.logevents.SimpleReport - Report for loadPageTest
+--------------------------------+--------------------+------------+------------+
| Element                        | Subject            | Status     | ms.        |
+--------------------------------+--------------------+------------+------------+
| By.xpath: //*[@id='logoImage'] | should be(visible) | FAIL       | 4334       |
+--------------------------------+--------------------+------------+------------+

[main] INFO com.codeborne.selenide.logevents.SimpleReport - Report for vyhledavaniTest
+-------------------------------------------------------------+----------------------------------+------------+------------+
| Element                                                     | Subject                          | Status     | ms.        |
+-------------------------------------------------------------+----------------------------------+------------+------------+
| By.xpath: //input[@name='patternFulltext']                  | should be(visible)               | PASS       | 38         |
| By.xpath: //*[@id='searchAdvanced1']                        | click()                          | PASS       | 82         |
| By.xpath: //label[@for='digiCheck']                         | should be(visible)               | PASS       | 25         |
| By.xpath: //label[@for='digiCheck']                         | click()                          | PASS       | 58         |
| By.xpath: //label[@for='tema_checkbox_SUA']                 | click()                          | PASS       | 53         |
| By.xpath: //label[@for='entity_checkbox_10033']             | scroll into view(true)           | PASS       | 16         |
| By.xpath: //label[@for='entity_checkbox_10033']             | click()                          | PASS       | 57         |
| By.xpath: //input[@value='Provést výběr']                   | should be(visible)               | PASS       | 21         |
| By.xpath: //input[@value='Provést výběr']                   | click()                          | PASS       | 1169       |
| By.xpath: //div[@class='listArticle active'][1]/a/div/span[1] | should be(visible)               | PASS       | 23         |
| By.xpath: //div[@class='listArticle active'][1]/a/div/span[1] | should have(inner text "Krevety") | PASS       | 22         |
| By.xpath: //*[@id='resetButton']                            | click()                          | PASS       | 575        |
| By.xpath: //a[contains(@tooltip, 'Jana')]                   | should be(visible)               | PASS       | 22         |
+-------------------------------------------------------------+----------------------------------+------------+------------+
```

Obrázek 14 Ukázka části výpisu využitím nástroje TextReport

12.3 Reportování pomocí frameworku TestNG

Využití frameworku TestNG při implementaci testů má jako další svou výhodu automatické vytváření výstupních reportů. V tomto případě generování hlášení o jednotlivých testovacích případech nedochází ke pouhému výpisu chyb do vývojářské konzole, nýbrž jde o vytvoření kompletního reportu s přehlednějším uživatelským rozhraním v podobě webové stránky. TestNG automaticky vytváří v adresáři projektu report, pokud není nastaveno jinak, který zahrnuje různé typy dat. Najdeme zde například XML soubor, který detailně popisuje kroky testů a jejich výsledky, podobně jako když používáme

třídu TextReport. Kromě toho se v reportu nachází soubor index.html, který po otevření ukáže vizuálně upravené informace o průběhu testů, jak můžeme vidět na obrázku č. (15). V adresáři projektu je také složka obsahující historii TestNG reportů.



Obrázek 15 Ukázka vzhledu reportu TestNG

Tento nástroj se celkově ukázal jako ne zcela vhodný pro mé potřeby. Sice nabízí uživatelsky přehlednější výpis testů, nicméně ty neobsahují dostatečné detaily a vzhled webové stránky reportu působí zastarale. Z těchto důvodů jsem se rozhodl, že tuto funkci frameworku TestNG nebudu ve svých projektech využívat. [55]

12.4 Nástroj pro reportování Allure

Jako nejpřívětivějším jak pro mě, tak pro mé kolegy byl vybrán právě nástroj Allure, nebo také Allure Report. Jedná se o moderní framework navržený pro vytváření efektivních a přehledných testovacích zpráv, díky tomu se stal velmi oblíbeným jak mezi QA inženýry tak samotnými vývojáři testů. Jeho integrace do kódu je velmi jednoduchá, stačí na něj pouze vytvořit závislost v dependencies projektu a není potřeba žádné další konfigurace. Dále při použití nástroje Selenide je důležité, aby na začátku každé testovací třídy byl jako posluchač přidán Allure. Toto nastavení umožňuje, že všechny testy v třídě jsou sledovány a dokumentovány pomocí Allure. Příklad implementace v ukázce zdrojového kódu č. (27).

```
@BeforeClass
public void setUpAll() {
    Configuration.browserSize = "1280x800";

    //Pridani Allure reportu
    SelenideLogger.addListener("allure", new AllureSelenide());
    open("https://example.com");
}
```

Zdrojový kód 27 Přidání Allure hlášení ke třídě

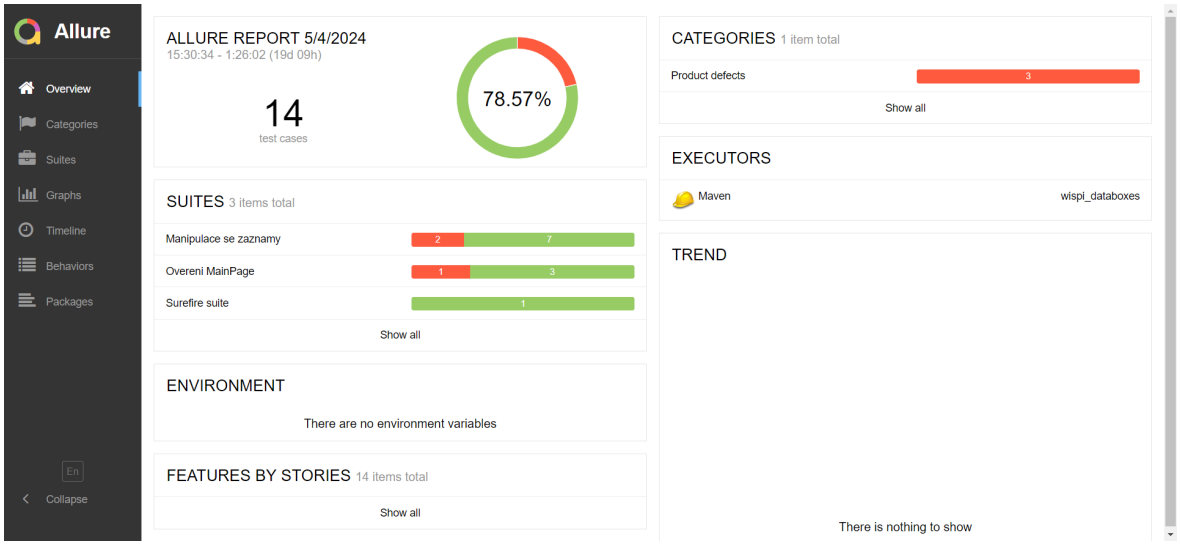
Přináší s sebou kromě uživatelsky přívětivého vzhledu také nové anotace pro detailnější popis jednotlivých testů. Konkrétně například anotace „Description“ pro slovní popis co daný test dělá nebo „Severity“, která určuje závažnost správné funkčnosti testu pro chod testované aplikace. Také se využívá anotace „Owner“ uvádějící osobu, která nese zodpovědnost za stabilitu daného testu. Přehled využitých příkladů anotací je možné nalézt na obrázku č. (28).

```
@Test
@AllureId("001") //Nastavi ID testu
@Description("Test pro kontrolu nacteni hlavni stranky")
@Severity(CRITICAL) //Kriticky pozadavek na spravnost testu
@Owner("Jan Voprsalek")
public void mainPageLoadTest() {
    //...
    //...
}
```

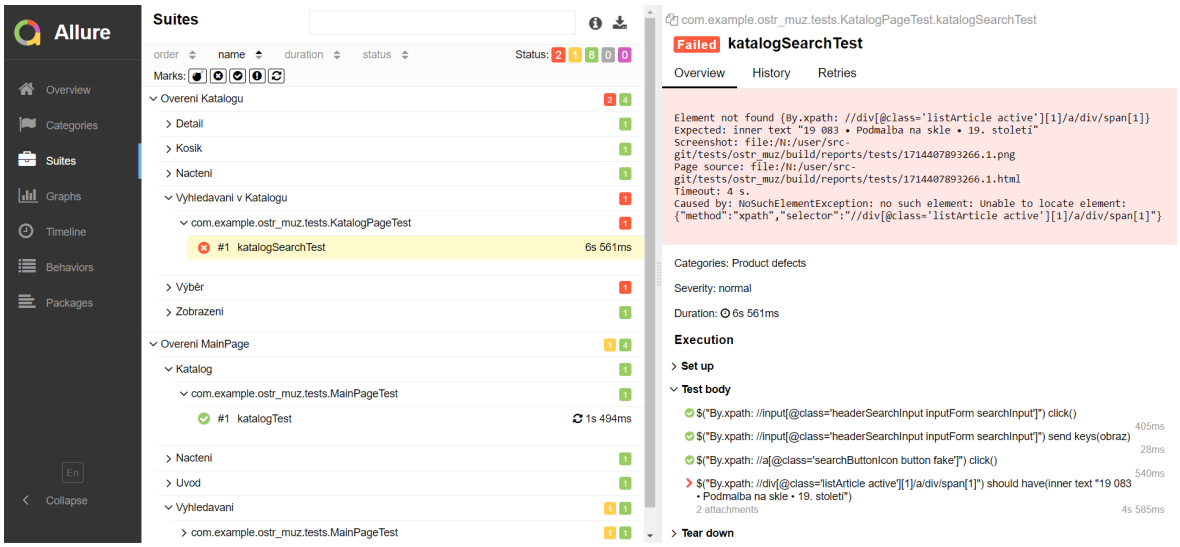
Zdrojový kód 28 Příklad anotací Allure u testu

Allure report se generuje využitím buildovacího nástroje Maven. Když Maven spustí testy během fáze buildu, Allure sbírá testovací data v reálném čase. Po dokončení fáze testování spolupracují k vygenerování reportu z těchto dat. Po provedení testů lze spustit Maven příkazy *allure:serve* nebo *allure:report*, které vygenerují a následně otevřou interaktivní Allure report. Obrázek č. (16) představuje samotný vzhled tohoto hlášení o průběhu testování.

Vizuální prezentace reportů pomocí Allure, stejně jako u reportů generovaných TestNG frameworkem, se provádí prostřednictvím webové stránky, kterou lze otevřít v jakémkoli webovém prohlížeči. Na úvodní stránce, která se automaticky zobrazuje pod záložkou s názvem „Overview“, lze nalézt klíčové informace. Zobrazuje se zde počet spuštěných testů, procentuální míra jejich úspěšnosti a seznam testovacích sad, které byly provedeny. V reportu Allure se dále nachází například grafy a časové osy, které zobrazují uživatelsky přívětivý přehled o testech a jejich výsledcích. Můžeme zde nalézt také detailní popis testu, ve kterém je možno prozkoumat jeho každý provedený krok a zjistit tak přesně, při které činnosti nastala chyba. U každého testu lze zobrazit jeho historická data, a tak sledovat opakovaná selhání a detekovat tím nestabilní části aplikace. [56]



Obrázek 16 Ukázka vzhledu reportu Allure



Obrázek 17 Výpis chyby pomocí Allure

13 SHRNUTÍ VYTVOŘENÉ AUTOMATIZACE

V závěrečné kapitole této diplomové práce se zaměřuji na zhodnocení projektu vytváření automatizovaných testů pro vybrané webové aplikace. Tato kapitola nabízí přehled hlavních přínosů, které mi automatizace přinesla, popisuje výzvy, kterým jsem čelil během vývoje testů a nastiňuji možné směry budoucího rozvoje tohoto projektu.

Má práce na projektu ukázala významné výhody, které automatizované testování přináší v oblasti úspory času a zvýšení spolehlivosti aplikací. Na druhou stranu jsem se setkal s řadou technických a procedurálních výzev, které jsem musel překonat. Díky těmto zkušenostem jsem získal cenné poznatky, které mohou sloužit jako základ pro další zlepšování testovacích procesů.

13.1 Přínosy práce

Hlavním přínosem práce je vytvoření automatizovaných testů pro vybrané webové aplikace, které jsou vyvíjeny v organizaci, kde pracuji. Díky těmto testům došlo ke snížení času, který byl dříve potřebný na manuální testování jednotlivých funkcí těchto aplikací. Kolegové, kteří se starají o obsluhu těchto aplikací, se tak mohou více věnovat jiným důležitým úkolům spojeným s údržbou softwaru. Automatizace testování také přinesla snížení chybovosti, která při manuálním testování často vznikala.

Dalším přínosem je, že automatizace testů umožnila nadřízeným lepší odhad časových a finančních nákladů potřebných pro plánování dalších testovacích aktivit. To vede k efektivnějšímu rozdělení zdrojů a lepšímu plánování rozpočtu na software. Kromě toho automatizované testy zvyšují důvěru v kvalitu a stabilitu aplikací, jenž může přispět k lepšímu vnímání produktu uživateli a potenciálně vést k vyšší uživatelské spokojenosti.

13.2 Problémy při tvorbě

Jak už to při vytváření čehokoliv bývá nelze se obejít bez vzniklých problémů. Moje práce tedy není výjimkou tohoto pravidla a spíše jej potvrzuje. Prvním hlavním úskalím, kterému jsem čelil, bylo získaní veškerých vědomostí k tvorbě automatizovaných testovacích případů samostudiem, jelikož jsem předchozí zkušenosti s touto problematikou neměl. Díky rozsáhlé literatuře a neomezenému počtu zdrojů, ze kterých bylo možné čerpat, mi tvorba šla dobrým tempem.

Dále při tvorbě samotných testů jsem narazil na problémy zcela nové. Hlavní výzvou pro mě bylo vytváření selektorů pro identifikaci prvků v testované webové aplikaci. Atributy těchto prvků byly často chaotické a chyběla jim jednoznačná a přehledná struktura. To bylo způsobeno tím, že aplikace nebyly původně navrženy s předpokladem, že

bude potřeba takové selektory vytvářet. Hledané prvky tak například neobsahovaly jednoznačné identifikátory ID, pomocí kterých je vytváření selektorů velmi rychlé a přesné. Naopak při některých prvcích nebyla jiná možnost než použít selektor absolutní, což není ideální z hlediska budoucí údržby testů.

Při spouštění testů ve vývojovém prostředí jsem čelil potížemi s často se vyskytující chybou o nenalezení elementu na stránce. Nejčastější příčina toho problému byla velmi rychlé vykonávání kroků daného testu, kdy některé operace stránka prováděla rychleji než jiné. Nakonec jsem na vyřešení této překážky použil metody *shouldBe()*, které v nástroji Selenide řeší ověření dostupnosti daného elementu. Konkrétně lze této metodě zadat parametr *visible*, který říká, že prvkem musí být na stránce zobrazen. Pro další informace o metodách nástroje Selenide doporučuji navštívit kapitolu č.[5.2].

13.3 Další možný vývoj

Možné budoucí rozšíření práce na automatizovaných testech zahrnuje vytvoření dalších testovacích případů, aby bylo pokryto co největší spektrum funkcí webové aplikace. Kromě toho se uvažuje o vytváření nových testovacích sad pro další typy softwaru, které firma spravuje a vyvíjí. Hlavním vylepšením bude integrace těchto testů do procesů CI/CD prostřednictvím softwaru Jenkins, což umožní snazší spouštění testů a automatizaci generování reportů.

Aktuálně není Jenkins připraven na integraci automatizovaných testů, neboť dosud nebyly součástí vývojového procesu firmy. Změny v konfiguraci Jenkins, které jsou nezbytné pro zahrnutí testů, vyžadují získání dalších oprávnění, která v současné době nevlastním. Tuto implementaci jsem projednal s kolegou, který vyjádřil plnou podporu myšlence začlenění testů do vývojového cyklu a je ochoten se podílet na jejím dalším rozvoji.

ZÁVĚR

Diplomová práce se zabývala návrhem a realizací automatizovaných testů pro webové aplikace vyvíjené v místní firmě. Cílem bylo zefektivnit proces testování a snížit potřebu lidských zdrojů pro pravidelné ověřování funkčnosti aplikací. Význam tohoto úsilí spočívá v neustálé potřebě zvyšování kvality a spolehlivosti softwaru, zvláště v kontextu současného rychlého technologického rozvoje.

Práce systematicky prošla klíčovými aspekty softwarového testování od teoretických základů, přes analýzu životního cyklu testování, až po praktickou implementaci testů. Teoretická část poskytla pevný základ pro pochopení metod a nástrojů používaných v automatizovaném testování, zatímco praktická část demonstrovala aplikaci těchto metod a nástrojů na konkrétní případy z praxe.

Při realizaci projektu jsem narazil na řadu výzev, které zahrnovaly technické obtíže s identifikací prvků na testovaných webových aplikacích a občasné problémy s nestabilitou aplikací. Tyto problémy byly postupně řešeny s využitím pokročilých funkcí testovacího nástroje Selenide, které vedlo k výraznému zlepšení efektivity testovacích procesů.

Zpětně lze konstatovat, že stanovené cíle byly úspěšně dosaženy. Automatizace testů vedla k podstatnému snížení času potřebného na testování a zvýšení přesnosti detekce chyb. Tento přístup nejen že zlepšil kvalitu vývoje v rámci firmy, ale také poskytl model, který je možno aplikovat na další vývojové projekty podobné struktury. Výsledky této práce poskytují komplexní obraz o významu a efektivitě automatizovaného testování v moderním softwarovém inženýrství.

V závěru práce byly uvedeny možnosti dalšího vývoje a rozšíření testovacích případů, které by pomohly k pokrytí většího spektra funkcí testovaných aplikací. Následně byla navržena integrace do systémů kontinuální vývoje a kontinuálního doručování, která by dále zvýšila flexibilitu a adaptabilitu testů do vývojových procesů softwaru.

SEZNAM POUŽITÉ LITERATURY

[1] SKILLMEA. Co je testování softwaru? Online. 2021. Dostupné z: https://skillmea.cz/blog/co-je-testovanie-softveru. [cit. 2024-02-27].

[2] KOĎUSKOVÁ, Barbora. Jak na testování webů a webových aplikací. Online. 2024. Dostupné z: https://www.rascasone.com/cs/blog/7-tipu-jak-na-testovani-webu-internetovych-stranek-a-mobilnich-aplikaci. [cit. 2024-02-27].

[3] SKILLMEA. TPOINT TECH. Software Testing Principles. Online. C2011-2021. Dostupné z: https://www.javatpoint.com/software-testing-principles. [cit. 2024-04-08].

[4] HAMILTON, Thomas. 7 Principles of Software Testing with Examples. Online. 2021. Dostupné z: https://www.guru99.com/software-testing-seven-principles.html. [cit. 2024-04-08].

[5] HAMILTON, Thomas. STLC (Software Testing Life Cycle). Online. 2024. Dostupné z: https://www.guru99.com/software-testing-life-cycle.html. [cit. 2024-04-20].

[6] TPOINT TECH. Software Testing Life Cycle (STLC). Online. C2011-2021. Dostupné z: https://www.javatpoint.com/software-testing-life-cycle. [cit. 2024-04-08].

[7] TPOINT TECH. Types of Software Testing. Online. C2011-2021. Dostupné z: https://www.javatpoint.com/types-of-software-testing. [cit. 2024-02-27].

[8] TPOINT TECH. Manual Testing. Online. C2011-2021. Dostupné z: https://www.javatpoint.com/automation-testing. [cit. 2024-04-08].

[9] TPOINT TECH. Automation Testing. Online. C2011-2021. Dostupné z: https://www.javatpoint.com/automation-testing. [cit. 2024-04-08].

[10] SUDEN, G. Automated Web Testing: Step by Step Automation Guide. CreateSpace Independent Publishing Platform, 2016. ISBN 978-1535285988.

[11] TPOINT TECH. Static Testing. Online. C2011-2021. Dostupné z: https://www.javatpoint.com/manual-testing. [cit. 2024-04-08].

[12] TPOINT TECH. Dynamic Testing. Online. C2011-2021. Dostupné z: https://www.javatpoint.com/static-testing. [cit. 2024-04-08].

[13] B., Anton. Static vs. Dynamic Testing: Definitions, Differences, and Business Considerations. Online. 2021. Dostupné z: https://testfort.com/blog/static-vs-dynamic-testing-definitions-differences-and-business-considerations. [cit. 2024-04-08].

[14] HLAVA, Tomáš. Testování bílé a černé skříňky (white box, black box, grey box). Online. 2011. Dostupné z: http://testovanisoftwaru.cz/metodika-testovani/druhy-typy-a-kategorie-testu/testovani-bile-a-cerne-skrinky-white-box-black-box-grey-box/. [cit. 2024-04-08].

[15] TPOINT TECH. Black box testing. Online. C2011-2021. Dostupné z: https://www.javatpoint.com/black-box-testing. [cit. 2024-04-08].

[16] TPOINT TECH. White Box Testing. Online. C2011-2021. Dostupné z: https://www.javatpoint.com/white-box-testing. [cit. 2024-04-08].

[17] TPOINT TECH. GreyBox Testing. Online. C2011-2021. Dostupné z: https://www.javatpoint.com/grey-box-testing. [cit. 2024-04-08].

[18] HAMILTON, Thomas. Functional Vs Non-Functional Testing – Difference Between Them. Online. 2024. Dostupné z: https://www.guru99.com/functional-testing-vs-non-functional-testing.html. [cit. 2024-04-08].

[19] Funkční vs nefunkční testování. Online. 2024. Dostupné z: http://test.swtestovani.cz/index.php?option=com_content&view=article&id=22:funkni-vs-nefunkni-testovani&catid=3:zaklady&Itemid=11. [cit. 2024-04-08].

[20] KINSBRUNER, Eran a BAHMUTOV, Gleb. A Frontend Web Developer's Guide to Testing: Explore leading web test automation frameworks and their future driven by low-code and AI. Packt Publishing, 2022. ISBN 978-1803238319.

[21] HLAVA, Tomáš. Funkční a nefunkční testy. Online. 2011. Dostupné z: http://testovanisoftwaru.cz/tag/nefunkcni-testy/. [cit. 2024-04-08].

[22] SHARMA, Pallavi. Selenium with Java – A Beginner's Guide: Web Browser Automation for Testing using Selenium with Java. BPB Publications, 2022. ISBN 978-9391392680.

[23] RICHARDSON, Alan. Selenium Simplified. 0002-Revised. Compendium Developments, 2012. ISBN 978-0956733238.

[24] SELENIUM. Selenium History. Online. C2024. Dostupné z: https://www.selenium.dev/history/. [cit. 2024-04-20].

[25] RUNGTA, Krishna. What is Selenium? Introduction to Selenium Automation Testing. Online. 2024. Dostupné z: https://www.guru99.com/introduction-to-selenium.html. [cit. 2024-04-20].

[26] BROWSERSTACK. What is Selenium? Online. 2024. Dostupné z: https://www.browserstack.com/selenium. [cit. 2024-04-20].

[27] SELENIDE. What is Selenide. Online. 2013. Dostupné z: https://selenide.org/2013/04/23/what-is-selenide/. [cit. 2024-04-20].

[28] SELENIDE. WHAT IS SELENIDE? Online. Dostupné z: https://selenide.org/index.html. [cit. 2024-04-20].

[29] APPLIEDTECH. Choosing tools for UI testing: Selenium or Selenide? Online. 2019. Dostupné z: https://www.appliedtech.ru/en/web-tools-for-ui-testing-selenium-or-selenide.html. [cit. 2024-04-20].

[30] GITHUB. Selenide vs Selenium. Online. 2015. Dostupné z: https://github.com/selenide/selenide/wiki/Selenide-vs-Selenium. [cit. 2024-04-20].

[31] JANOVSKÝ, Dušan. Iframe. Online. Dostupné z: https://www.jakpsatweb.cz/iframe.html. [cit. 2024-04-30].

[32] PALANI, Narayanan. Automated Software Testing with Cypress. Auerbach Publications, 2021. ISBN 978-0367759681.

[33] CYPRESS.IO. Why Cypress? Online. C2024. Dostupné z: https://docs.cypress.io/guides/overview/why-cypress. [cit. 2024-04-22].

[34] UNADKAT, Jash. Cypress vs Selenium: Key Differences. Online. 2023. Dostupné z: https://www.browserstack.com/guide/cypress-vs-selenium. [cit. 2024-04-22].

[35] BOSE, Shreya. How to write Test Cases in Software Testing? (with Format & Example). Online. 2023. Dostupné z: https://www.browserstack.com/guide/how-to-write-test-cases. [cit. 2024-04-25].

[36] TESTOVANISOFTWARU. Test Case – Testovací případ. Online. Dostupné z: http://testovanisoftwaru.cz/dokumentace-v-testovani/test-case/. [cit. 2024-04-25].

[37] KITAKABEE. What is a Test Suite & Test Case? (with Examples). Online. 2022. Dostupné z: https://www.browserstack.com/guide/what-is-test-suite-and-test-case. [cit. 2024-04-25].

[38] MEDIUM. Using Web Element Locators in Test Automation More Effectively. Online. 2019. Dostupné z: https://medium.com/ranorex-webtestit/using-web-element-locators-in-test-automation-more-effectively-12c25c09d0d9. [cit. 2024-04-08].

[39] SELENIUM. Locator strategies. Online. C2024. Dostupné z: https://www.selenium.dev/documentation/webdriver/elements/locators/. [cit. 2024-04-08].

[40] DAITYARI, Shaumik. How to use XPath in Selenium? (using Text, Attributes, Logical Operators). Online. 2023. Dostupné z: https://www.browserstack.com/guide/xpath-in-selenium. [cit. 2024-04-25].

[41] AMAZON WEB SERVICES. What is Java? Online. C2024. Dostupné z: https://aws.amazon.com/what-is/java/. [cit. 2024-04-25].

[42] JETBRAINS. IntelliJ IDEA overview. Online. 2024. Dostupné z: https://www.jetbrains.com/help/idea/discover-intellij-idea.html#IntelliJ-IDEA-supported-languages. [cit. 2024-04-25].

[43] TESTNG. TestNG Documentation. Online. 2024. Dostupné z: https://testng.org/. [cit. 2024-04-25].

[44] TPOINT TECH. What is TestNG Annotation? Online. C2011-2021. Dostupné z: https://www.javatpoint.com/testng-annotations. [cit. 2024-04-25].

[45] BACH SYSTEMS. Co je Muzejní VadeMeCum? Online. C1994-2024. Dostupné z: http://www.bach.cz/produkty/muzejni-vademecum/. [cit. 2024-04-18].

[46] BACH SYSTEMS. Národní galerie v Praze. Online. C2021. Dostupné z: https://ng.bach.cz/ng_vade/. [cit. 2024-04-18].

[47] BACH SYSTEMS. Valašské muzeum v přírodě. Online. C2014. Dostupné z: https://vademecum.vmp.cz/vademecum/. [cit. 2024-04-18].

[48] BACH SYSTEMS. Středočeské muzeum v Roztokách u Prahy. Online. C2024. Dostupné z: https://vademecum.muzeum-roztoky.cz/vademecum/. [cit. 2024-04-18].

[49] BACH SYSTEMS. Ostravské muzeum. Online. C2024. Dostupné z: https://sbirkyonline.ostrmuz.cz/vademecum_muzeum/. [cit. 2024-04-18].

[50] BACH SYSTEMS. FotoArchiv. Online. C2024. Dostupné z: https://demo.bach.cz/web-bach-foto/. [cit. 2024-04-18].

[51] BACH SYSTEMS. Co je spisová služba WISPI? Online. C1994-2024. Dostupné z: http://www.bach.cz/produkty/spisova-sluzba-wispi/. [cit. 2024-04-18].

[52] RUNGTA, Krishna. Page Object Model (POM) & Page Factory in Selenium. Online. 2024. Dostupné z: https://www.guru99.com/page-object-model-pom-page-factory-in-selenium-ultimate-guide.html. [cit. 2024-04-20].

[53] SON, Hannah. How to Report On Test Automation (Tools + Metrics). Online. 2023. Dostupné z: https://www.testrail.com/blog/report-test-automation/. [cit. 2024-04-30].

[54] BHARATI, Neha. Top Selenium Reporting Tools. Online. 2023. Dostupné z: https://www.browserstack.com/guide/top-selenium-reporting-tools. [cit. 2024-04-30].

[55] SELENIDE. Documentation Reports. Online. C2024. Dostupné z: https://selenide.org/documentation/reports.html. [cit. 2024-04-30].

[56] QAMETA SOFTWARE. Allure TestNG. Online. C2024. Dostupné z: https://allurereport.org/docs/testng/. [cit. 2024-04-30].

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

STLC	Software Testing Life Cycle
QA	Quality Assurance
RTM	Requirements Traceability Matrix
API	Application Programming Interface
RC	Remote Control
IDE	Integrated Development Environment
HTTP	Hypertext Transfer Protocol
XML	Extensible Markup Language
AJAX	Asynchronous JavaScript and XML
HTML	Hypertext Markup Language
UI	User Interface
DOM	Document Object Model
URL	Uniform Resource Locator
CSS	Cascading Style Sheets
XPath	XML Path Language
WORA	Write Once, Run Anywhere
IoT	Internet of Things
JVM	Java Virtual Machine
DPH	Daň z Přidané Hodnoty
POM	Page Object Model
CI/CD	Continuous Integration / Continuous Delivery

SEZNAM OBRÁZKŮ

SEZNAM TABULEK

SEZNAM ZDROJOVÝCH KÓDŮ

SEZNAM PŘÍLOH

PŘÍLOHA P I. OBSAH PŘILOŽENÉHO CD/DVD

Struktura adresářů v přiloženém CD/DVD:

src/

Kompletní zdrojové kódy testů v rámci jednotlivých projektů.

ngp/ - Projekt Národní galerie v Praze.

ostr_muz/ - Projekt Ostravské muzeum.

smrp/ - Projekt Středočeské muzeum v Roztokách u Prahy.

vmp/ - Projekt Valašské muzeum v přírodě.

fotoArchiv/ - Projekt fotoArchiv.

wispi_databoxes/ - Projekt Spisová služba WISPI.

tab/

Tabulky všech podrobně popsaných testovacích případů dle jednotlivých aplikací.

FotoArchiv/ - Testovací případy webové aplikace FotoArchiv.

VadeMeCum/ - Testovací případy webových aplikací VadeMeCum.

WISPI/ - Testovací případy webové aplikace WISPI.

doc/

Text práce ve formátu PDF/A.