



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

DEPARTMENT OF COMPUTER SYSTEMS

AUTOMATICKÉ TESTY PRO SYSTÉM K-DISPATCH

AUTOMATED TESTING AND CONTINUOUS INTEGRATION FOR THE K-DISPATCH SYSTEM

BAKALÁŘSKÁ PRÁCE

BACHELOR'S THESIS

AUTOR PRÁCE

AUTHOR

SIMONA VALKOVSKÁ

VEDOUCÍ PRÁCE

SUPERVISOR

Ing. MARTA JAROŠ, Ph.D.

BRNO 2024

Zadání bakalářské práce



161562

Ústav: Ústav počítačových systémů (UPSY)
Studentka: **Valkovská Simona**
Program: Informační technologie
Název: **Automatické testy pro systém k-Dispatch**
Kategorie: Analýza a testování softwaru
Akademický rok: 2024/25

Zadání:

1. Seznamte se s druhy automatických testů (jednotkové testy, regresní testy, integrační testy).
2. Seznamte se s metodikou návrhu software ve výzkumné skupině SC@FIT.
3. Navrhněte demonstrační aplikaci, pro kterou vytvoříte krátké ukázky testů na různých úrovních.
4. Seznamte se se systémem k-Dispatch a jeho moduly.
5. Navrhněte sadu různých testů pro ověření funkcionality vybraných modulů a nasazení systému k-Dispatch.
6. Navržené testy implementujte a řešení otestujte.
7. Navrhněte postup pro automatické spouštění navržených testů a implementujte.
8. Vytvořené řešení zdokumentujte a demonstруйте.

Literatura:

- Dle pokynů vedoucího.

Při obhajobě semestrální části projektu je požadováno:

- Body 1 až 3 zadání.

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

Vedoucí práce: **Jaroš Marta, Ing., Ph.D.**
Vedoucí ústavu: Sekanina Lukáš, prof. Ing., Ph.D.
Datum zadání: 1.11.2024
Termín pro odevzdání: 14.5.2025
Datum schválení: 31.10.2024

Abstrakt

Táto bakalárska práca sa zameriava na testovanie softvéru, s dôrazom na praktickú implementáciu testovacích metód a nástrojov. V práci je predstavená demonštračná aplikácia knižničného systému, ktorá slúži na ukážku rôznych typov testovania vrátane jednotkových, integračných a end-to-end testov. Následne práca analyzuje systém k-Dispatch, pre ktorého kľúčové moduly boli navrhnuté komplexné testovacie scenáre. Praktická časť zahŕňa implementáciu jednotkových testov pre databázové metódy, výkonnostné testovanie SQL dotazov a automatizované testy webového rozhrania pomocou Selenium. Súčasťou riešenia je aj kontajnerizácia aplikácie technológiou Docker a nastavenie CI/CD pipeline pre automatizáciu testovacieho procesu. Práca demonštruje význam systematického testovania pri vývoji softvéru a poskytuje metodologický základ pre budúce rozšírenia testovacieho rámca.

Abstract

This bachelor thesis focuses on software testing, with emphasis on practical implementation of testing methods and tools. The thesis presents a demonstrative library system application, which serves to showcase various types of testing including unit, integration, and end-to-end tests. Subsequently, the thesis analyzes the k-Dispatch system, for which complex testing scenarios were designed for key modules. The practical part includes implementation of unit tests for database methods, performance testing of SQL queries, and automated testing of the web interface using Selenium. The solution also includes containerization of the application using Docker technology and setting up a CI/CD pipeline for test process automation. The thesis demonstrates the importance of systematic testing in software development and provides a methodological foundation for future extensions of the testing framework.

Kľúčové slová

Testovanie softvéru, životný cyklus testovania softvéru (STLC), životný cyklus vývoja softvéru (SDLC), funkčné testovanie, nefunkčné testovanie, jednotkové testovanie, integračné testovanie, regresné testovanie, Python, pytest, Selenium, Docker, kontinuálna integrácia a nasadenie (CI/CD), k-Dispatch, SQL optimalizácia, vysokovýkonná výpočtová infraštruktúra (HPC)

Keywords

Software testing, Software Testing Life Cycle (STLC), Software Development Life Cycle (SDLC), functional testing, non-functional testing, unit testing, integration testing, regression testing, Python, pytest, Selenium, Docker, Continuous Integration and Continuous Deployment (CI/CD), k-Dispatch, SQL optimization, High-Performance Computing infrastructure (HPC)

Citácia

VALKOVSKÁ, Simona. *Automatické testy pro systém k-Dispatch*. Brno, 2024. Bakalárska práca. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Marta Jaroš, Ph.D.

Automatické testy pro systém k-Dispatch

Prehlásenie

Prehlasujem, že som túto bakalársku prácu vypracovala samostatne pod vedením pani Ing. Marty Jaroš, Ph.D. Uviedla som všetky literárne pramene, publikácie a ďalšie zdroje, z ktorých som čerpala.

.....
Simona Valkovská
13. mája 2025

Podakovanie

Rada by som sa poďakovala vedúcej práce, pani Ing. Marte Jaroš, Ph.D. za jej výnimočné vedenie počas celého procesu vypracovania tejto práce. Jej odborné znalosti, cenné rady a konštruktívna spätná väzba významne prispeli ku kvalite výsledného diela.

Obsah

1	Úvod	4
2	Testovanie software	5
2.1	Životný cyklus testovania softvéru	5
2.2	Životný cyklus vývoja softvéru (SDLC)	9
3	Funkčné testovanie	12
3.1	Proces funkčného testovania	12
3.2	Jednotkové testovanie	12
3.3	Integračné testovanie	18
3.4	Regresné testovanie	21
4	Nefunkčné testovanie	23
4.1	Charakteristika nefunkčného testovania	23
4.2	Význam v procese testovania	23
4.3	Bezpečnostné testovanie	24
5	Testovacie metódy	31
5.1	Manuálne testovanie	31
5.2	Automatizované testovanie	31
5.3	Porovnanie metód	31
6	Testovanie webových aplikácií	33
6.1	Najlepšie nástroje na testovanie webových aplikácií	34
7	Kontajnerizácia a Monitorovanie	35
7.1	Fluent Bit	35
8	Testovacie frameworky v jazyku Python	37
8.1	Unittest	37
8.2	Pytest	37
8.3	Hypothesis	37
8.4	Porovnanie rozhraní	38
9	Praktická časť	39
9.1	Demonštračná aplikácia a implementácia testov	39
9.2	K-Dispatch aplikácia	44
9.3	Návrh testovacích scenárov	54

10 Záver	66
10.1 Použité implementačné prístupy a technológie	66
10.2 Možnosti ďalšieho rozšírenia	66
Literatúra	68
A Obsah odovzdaných súborov	71

Zoznam obrázkov

2.1	STLC	6
2.2	SDLC	10
3.1	Typy funkčného testovanie	12
3.2	Typy jednotkového testovania, zľava <i>White box</i> , <i>Grey box</i> a <i>Black box</i> testovanie	14
9.1	Architektúra systému	40
9.2	Dátový model systému	40
9.3	End-to-End (E2E) testy pre knižničný systém. Diagram ilustruje interakcie medzi tromi hlavnými komponentami: Používateľ, Knižničný systém a Databáza	42

Kapitola 1

Úvod

Kľúčovou fázou vo vývoji akéhokoľvek softvérového riešenia je dôkladné testovanie, ktoré zohráva nezastupiteľnú úlohu pri zabezpečovaní jeho kvality, spoľahlivosti a bezpečnosti. Jeho význam spočíva v schopnosti včas odhaliť chyby ešte pred dodaním produktu klientovi, čím sa nielen predchádza komplikáciám a nespokojnosti, ale aj výrazne šetrí finančné prostriedky keďže oprava problémov po spustení aplikácie je vo všeobecnosti nákladnejšia. Nemenej dôležité je odhaľovanie bezpečnostných zraniteľností, ktoré môžu v nezabezpečenom kóde predstavovať vážne riziko, najmä v online aplikáciách často vystavených kybernetickým útokom. Testovanie tiež pomáha pri overovaní škálovateľnosti, teda schopnosti aplikácie zvládať narastajúce pracovné zaťaženie, ako je objem dát či počet súbežných používateľov, a identifikovať body, kedy by mohla prestať optimálne fungovať.

S ohľadom na uvedené kritické aspekty sa táto práca zameriava na demonštráciu testovania softvéru na implementovanej základnej aplikácii. Ukážky demonštračnej aplikácie 1.0.1 poskytujúce rôzne typy testovania a využitia v praxi vás budú prevádzať celou prácou.

V rámci tejto práce sa vyvinula demonštračná aplikácia na prezentáciu rôznych typov a metód testovania na zjednodušenom príklade pre rýchle pochopenie. Práca zahŕňa aj analýzu testovacích prípadov pre moduly aplikácie **k-Dispatch**, s následnou implementáciou vybraných testov. Aplikácia je vyvinutá v jazyku **Python**, rovnako ako **k-Dispatch**. Databáza **k-Dispatch** bude otestovaná jednotkovými testami pre metódy a atribúty modulov, ako aj výkonnosťnými testami na optimalizáciu SQL príkazov, ktoré môžu významne zlepšiť odozvu pri väčšom objeme dát. Pre webové rozhranie **k-Dispatch** bol vytvorený testovací prípad pomocou **Selenium**, ktorý slúži ako vzor pre ďalšie testy tohto typu.

Ukážka

Príklad jednoduchého knižničného systému predstavuje ideálnu prípadovú štúdiu, pretože zahŕňa bežné komponenty moderných aplikácií:

- Správu používateľov (registrácia čitateľov)
- Správu zdrojov (katalóg kníh)
- Biznis logiku (pravidlá požičiavania)

Ukážka 1.0.1: Príklad

Kapitola 2

Testovanie software

Softvérové testovanie je metóda ktorou kontrolujeme, či existujúci produkt spĺňa očakávané požiadavky, ktoré môžu byť explicitné či implicitné. Touto metódou taktiež zaistujeme čo najnižšiu chybovosť softvérového produktu a minimalizáciu množstva ďalších investícií, ktoré by spoločnosť investovala do riešenia problémov.

Samotná dôležitosť testovania tkvie v jeho komplexnom prínose pre celý životný cyklus softvéru. Umožňuje nielen včasnú identifikáciu chýb, čím sa výrazne znižujú náklady na ich opravu v neskorších fázach, ale prispieva aj k zlepšeniu celkovej kvality a robustnosti produktu. Výsledkom je nárast dôvery a spokojnosti koncových používateľov. Okrem toho, je dôsledné testovanie kľúčové, pre odhaľovanie bezpečnostných zraniteľností, ktoré by mohli byť zneužitú, a pre overenie schopnosti aplikácie škálovať sa a efektívne zvládať očakávanú prevádzkovú záťaž.

Tento spôsob zahŕňa spúšťanie softvérových a manuálnych komponentov pomocou manuálnych alebo automatizovaných nástrojov. Teraz si predstavíme ako by to mohlo vyzeráť pre našu demonštračnú aplikáciu 2.0.1. V tejto kapitole som čerpala zo zdrojov [37] [20] [13] [3] [33] [34] [6] [32].



V kontexte nášho knižničného systému môžeme identifikovať niekoľko kritických oblastí, kde je testovanie nevyhnutné:

- Správnosť pravidiel požičiavania kníh
- Integrita dát používateľov a kníh
- Výkon pri vyhľadávaní v katalógu

Ukážka 2.0.1: Testovanie softvéru

2.1 Životný cyklus testovania softvéru

Životný cyklus testovania softvéru, ďalej ako STLC (Software testing life cycle) je systematický prístup k testovaniu, zahŕňajúci overovanie aplikácie s cieľom skontrolovať, či spĺňa

požiadavky a či neobsahuje chyby. Sleduje sériu/fázy krokov, kde každá fáza má svoj cieľ a výstupy a softvér musí tieto ciele splniť, aby mohol prejsť do ďalšieho kroku. Cieľom STLC pri testovaní, je poskytnúť kvalitnú, spoľahlivú a stabilnú aplikáciu, ktorá zostane verná požiadavkám zákazníkov. Informácie v sekcii sú čerpané z [16]. V nasledujúcich častiach sa budeme podrobne zaoberať každou z fáz STLC 2.1.



Obr. 2.1: STLC

2.1.1 Analýza požiadaviek

Prvým a kľúčovým požiadavkom v STLC je analýza požiadavok 2.1.1. Základom celej analýzy je podrobné preskúmanie dokumentu s požiadavkami na softvér, ďalej ako SRD (Software requirements document) spolu s ďalšou relevantnou dokumentáciou. Dôležitou súčasťou je tiež identifikácia možných rizík, ktoré by mohli ovplyvniť priebeh testovania. Pre lepšiu organizáciu a prehľadnosť vytvára tím maticu sledovateľnosti požiadaviek, ďalej ako RTM (Requirements traceability matrix). Táto matica slúži ako most medzi požiadavkami a testovacími prípadmi, čo umožňuje efektívne sledovanie pokrytia všetkých požiadaviek testami.



V prípade nášho knižničného systému zahŕňa analýza požiadaviek nasledujúce zainteresované strany:

- Knihovníkov - požiadavky na správu kníh a používateľov
- Čitateľov - funkcionality výpožičiek a prezerania katalógu
- IT administrátorov - technické požiadavky na prevádzku systému
- Vedenie knižnice - administratívne funkcie

Ukážka 2.1.1: Analýza požiadavok

2.1.2 Plánovanie testov

Proces začína stanovením jasných cieľov testovania a definovaním jeho rozsahu. Tím následne vypracováva dôkladnú stratégiu testovania, ktorá zahŕňa výber najvhodnejších metód a techník pre daný projekt. Dôležitou súčasťou plánovania je určenie testovacieho prostredia a identifikácia všetkých potrebných zdrojov. Tím tiež definuje konkrétne testovacie prípady a pripravuje testovacie údaje, ktoré sa budú počas testovania využívať [2.1.2](#). V rámci plánovania sa vypracováva aj časový harmonogram a rozpočet projektu. Stanovujú sa kľúčové milníky a očakávané výsledky testovania.



Pre knižničný systém je potrebné naplánovať testovanie z pohľadu:

- Používateľského rozhrania - testy pre knihovníkov a čitateľov
- Biznis logiky - testy pravidiel výpožičiek a správy katalógu
- Dátovej vrstvy - testy integrity a perzistencie dát
- Výkonu - testy odozvy systému pri viacerých používateľoch

Ukážka 2.1.2: Plánovanie testov

2.1.3 Fáza vývoja testovacích prípadov

Na začiatku procesu tím identifikuje konkrétne testovacie prípady, ktoré je potrebné vyvinúť [2.1.3](#). Pri ich písaní sa kladie dôraz na jasnosť, stručnosť a zrozumiteľnosť, aby boli ľahko pochopiteľné a vykonateľné. Súčasťou procesu je aj príprava testovacích údajov a scenárov, ktoré budú použité v jednotlivých testovacích prípadoch. Pre každý testovací prípad sa jasne definujú očakávané výsledky, čo umožňuje jednoznačné vyhodnotenie úspešnosti testov.



Pre knižničný systém sa vytvárajú testovacie prípady pokrývajúce:

- Procesy správy kníh a používateľov
- Scenáre výpožičiek a vrátení
- Prístupové práva rôznych rolí
- Spracovanie chybových stavov

Ukážka 2.1.3: Vývoj testovacích prípadov

2.1.4 Nastavenie testovacieho prostredia

Prostredie vytvára základné podmienky, v ktorých bude prebiehať samotné testovanie softvéru, a preto má zásadný vplyv na kvalitu a spoľahlivosť testovacieho procesu. Umožňuje simulovať reálne podmienky používania softvéru a tým zabezpečiť, že testovanie odhalí potenciálne problémy ešte pred nasadením do produkcie [2.1.4](#).



Vyžaduje sa príprava:

- Testovacej databázy so vzorovou množinou dát
- Konfigurácie pre rôzne role používateľov
- Simulácie reálnej prevádzky knižnice
- Monitorovacích nástrojov pre sledovanie výkonu

Ukážka 2.1.4: Testovacie prostredie

2.1.5 Vykonávanie testov

Testovací tím v tejto fáze systematicky realizuje všetky pripravené testovacie prípady podľa vopred stanovených postupov. Vykonávanie testov je iteratívny proces, často je potrebné opakovať testovacie cykly viackrát, najmä po úpravách kódu, pridávaní novej funkcionality, kým sa neodstránia všetky identifikované chyby a softvér nedosiahne požadovanú kvalitu pre vydanie.

Každá identifikovaná chyba sa zaznamenáva spolu s kľúčovými informáciami ako závažnosť problému, jeho priorita a detailný popis. Tím priebežne analyzuje výsledky testov, hodnotí výkonnosť softvéru a identifikuje problematické oblasti. Po oprave chýb nasleduje opätovné testovanie, ktoré má potvrdiť úspešnosť vykonaných opráv. Všetky výsledky tes-

tov sa dôkladne dokumentujú a komunikujú zainteresovaným stranám. Táto komunikácia zabezpečuje, že všetci účastníci projektu majú aktuálne informácie o stave testovania.

2.1.6 Uzavretie testovania

Uzavretie testovania predstavuje záverečnú fázu v ktorej sa finalizujú a dokumentujú všetky testovacie aktivity s hlavným cieľom potvrdiť pripravenosť softvéru na vydanie 2.1.5. Tím pripravuje komplexnú súhrnnú správu o testovaní, ktorá zahŕňa všetky podstatné metriky - počet vykonaných testov, identifikované chyby a celkovú úspešnosť testovania.



Záverečná fáza zahŕňa:

- Vyhodnotenie splnenia požiadaviek všetkých zainteresovaných strán
- Dokumentáciu nájdených a opravených chýb
- Odporúčania pre budúce vylepšenia systému
- Prípravu systému na nasadenie do produkcie

Ukážka 2.1.5: Uzavretie testovania

2.2 Životný cyklus vývoja softvéru (SDLC)

Životný cyklus vývoja softvéru (Software Development Life Cycle - SDLC) predstavuje systematický prístup k tvorbe softvérových riešení, ktorého primárnym cieľom je zabezpečenie optimálnej kvality výsledného produktu pri efektívnom využití dostupných zdrojov a času. Tento metodologický rámec poskytuje štruktúrovaný sled fáz, prostredníctvom ktorých organizácie dokážu vytvárať robustné softvérové riešenia pripravené na nasadenie v produkčnom prostredí. Informácie v sekcii sú čerpané z [15]. Následne si predstavíme jednotlivé fázy SDLC 2.2.



Obr. 2.2: SDLC

2.2.1 Analýza požiadaviek

Iniciálna fáza SDLC zahŕňa komplexný zber a analýzu požiadaviek od všetkých relevantných zainteresovaných strán, vrátane koncových používateľov, dodávateľov, doménových expertov a vývojového tímu. V rámci tejto fázy sa realizuje detailná evaluácia existujúcich systémov s cieľom identifikácie priestoru pre optimalizáciu.

2.2.2 Projektové plánovanie

Táto fáza sa zameriava na detailnú špecifikáciu potrebných zdrojov a stanovenie rozpočtu pre implementáciu identifikovaných požiadaviek. Súčasťou je aj komplexná analýza rizík a návrh stratégií ich mitigácie.

2.2.3 Systémový návrh

V tejto fáze dochádza k transformácii softvérových špecifikácií do formálneho návrhového dokumentu. Tento dokument podlieha dôkladnému pripomienkovaniu všetkými zainteresovanými stranami. Kvalitný proces zberu a integrácie spätnej väzby je kritický pre úspech projektu, keďže nedostatky v tejto fáze môžu viesť k významným prekročeniam rozpočtu alebo až k zlyhaniu celého projektu.

2.2.4 Implementácia

Implementačná fáza predstavuje realizáciu samotného vývoja softvéru. Fundamentálnym aspektom je striktné dodržiavanie stanovených návrhových princípov a implementačných štandardov. Esenciálnou súčasťou je definícia a dodržiavanie konvencií pre písanie zdrojového kódu, vrátane jednotného systému pomenovania súborov a premenných (napríklad *camelCase*).

2.2.5 Verifikácia a validácia

Táto fáza zahŕňa systematické testovanie softvérového produktu s cieľom identifikácie a eliminácie defektov. Proces verifikácie a validácie zabezpečuje súlad implementácie s pôvodnými špecifikáciami.

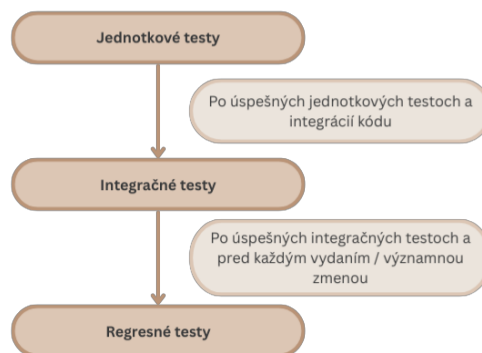
2.2.6 Nasadenie a údržba

Finálna fáza životného cyklu predstavuje nasadenie softvérového produktu do produkčného prostredia. Moderné organizácie často implementujú viacstupňový proces nasadenia využívajúci vývojové, testovacie a produkčné prostredia. Tento prístup umožňuje bezpečnú validáciu produktu v kontrolovanom prostredí pred finálnym nasadením, čím sa minimalizuje riziko výskytu kritických chýb v produkčnej prevádzke.

Kapitola 3

Funkčné testovanie

Funkčné testovanie predstavuje špecifický druh testovania softvéru, ktorý sa zameriava na overenie, či systém správne vykonáva všetky požadované funkcie podľa stanovených špecifikácií. Pri tomto type testovania sa simuluje reálne používanie systému koncovými užívateľmi **3.1**. V tejto kapitole čerpám zo zdrojov [12] [21].



Obr. 3.1: Typy funkčného testovanie

3.1 Proces funkčného testovania

Proces funkčného testovania možno zhrnúť do troch hlavných krokov:

1. Zadanie testovacích vstupných údajov
2. Definovanie očakávaného výstupu
3. Porovnanie skutočného výstupu s očakávaným výsledkom

Tento systematický prístup umožňuje dôkladne overiť, či každá funkcia aplikácie pracuje presne tak, ako bolo navrhnuté v špecifikácii. Testovanie sa pritom sústreďuje výhradne na funkčnosť systému z pohľadu užívateľa, nie na technické detaily jeho implementácie.

3.2 Jednotkové testovanie

Jednotkové testovanie predstavuje fundamentálny koncept v oblasti vývoja softvéru. Ide o metódu overovania najmenších testovateľných častí kódu voči ich účelu a špecifikova-

ným požiadavkám. Testovanou jednotkou môže byť napríklad individuálna funkcia, objekt, metóda, procedúra alebo samostatný modul 3.2.1. [22]

👁️ Jednotkové testy pridania knihy

V našom knižničnom systéme predstavuje jednotkové testovanie overenie základných operácií jednotlivých komponentov. Napríklad, testovanie pridania novej knihy:

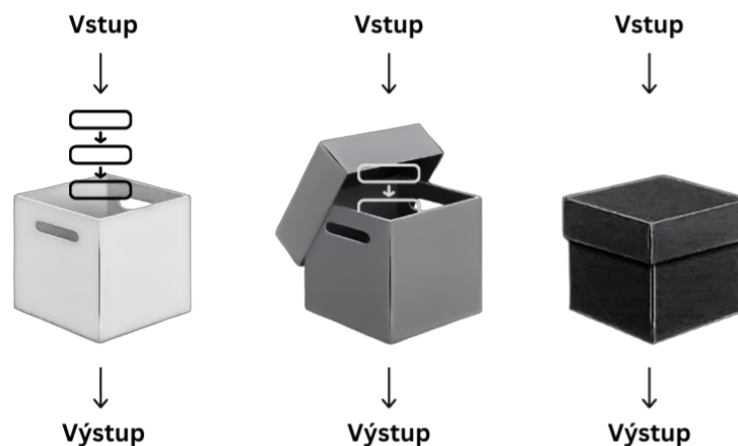
```
1 def testAddBookAllRequiredFields(self, setupLibrary):
2     """Test adding book with all required fields"""
3     success, message = setupLibrary.addBook(
4         bookId="book1",
5         title="Test Book",
6         author="Test Author"
7     )
8     assert success == True
9     assert "successfully" in message
10
11 def testAddBookMissingId(self, setupLibrary):
12     """Test adding book with missing ID"""
13     success, message = setupLibrary.addBook(
14         bookId="",
15         title="Test Book",
16         author="Test Author"
17     )
18     assert success == False
19     assert "required" in message
```

Ďalšími príkladmi jednotkového testovania v našej aplikácii môže byť overenie dostupnosti neexistujúcej knihy, testovanie úpravy údajov používateľa, či kontrola formátu identifikátora knihy.

Ukážka 3.2.1: Jednotkové testovanie

3.2.1 Techniky jednotkového testovania

Techniky jednotkového testovania sa líšia najmä v prístupe k testovanému kódu a na základe ich transparentnosti môžeme rozlíšiť tri hlavné typy, viď Obr. 3.2.



Obr. 3.2: Typy jednotkového testovania, zľava *White box*, *Grey box* a *Black box* testovanie

White Box testovanie

Testovanie typu White Box sa zameriava na analýzu vnútorných štruktúr kódu. Pri tomto type testovania má tester prístup k zdrojovému kódu a využíva tieto znalosti na návrh testovacích prípadov [3.2.2.](#) [\[4\]](#) [\[31\]](#) Poznáme nasledujúce typy white box testovania:

- **Mutačné testovanie** - Predstavuje metódu, pri ktorej sa zámerne vkladajú chyby do testovaného programu prostredníctvom vytvárania mutantov - verzií s drobnými modifikáciami pôvodného kódu. Využíva tri základné typy mutácií: mutácie rozhodnutí (zmeny operátorov), mutácie hodnôt a mutácie príkazov.
- **Testovanie toku dát** - Zameriava sa na sledovanie premenných a ich hodnôt počas behu programu. Rozlišuje tri kľúčové koncepty: definíciu premennej, výpočtové použitie (c-use) a predikátové použitie (p-use). Hlavným cieľom je identifikácia problémov v manipulácii s premennými.
- **Testovanie riadiaceho toku** - Analyzuje poradie vykonávania príkazov v programe pomocou grafovej reprezentácie. Graf obsahuje uzly (príkazy), hrany (tok riadenia), rozhodovacie uzly (vetvenie) a spojovacie uzly (spájanie vetiev programu). Slúži na overenie správnosti postupnosti vykonávania inštrukcií.

👁 White box testovanie procesu vrátenia knihy

V našom knižničnom systéme by white box testing vykonával vývojový tím, keďže má najhlbšie znalosti internej štruktúry a implementačných detailoch systému. Napríklad, uvažujme metódu `returnBook` v triede `LibrarySystem`. Vývojár by mohol napísať white box test na overenie internej logiky procesu vrátenia knihy:

```
1 def testReturnBookInternalLogic(self, setupLibrary):
2     """Test of internal logic for the book return process"""
3     library = setupLibrary
4     # Preparing test data
5     library.addUser("user1", "Test User", "test@test.com")
6     library.addBook("book1", "Test Book", "Test Author")
7     library.borrowBook("user1", "book1")
8
9     # Verifying internal state before return
10    book = library.database.getBook("book1")
11    user = library.database.getUser("user1")
12    assert book['available'] == False
13    assert book['borrowedBy'] == "user1"
14    assert "book1" in user['borrowedBooks']
15
16    # Executing book return
17    success, _ = library.returnBook("user1", "book1")
18    assert success == True
19
20    # Verifying internal state after return
21    book = library.database.getBook("book1")
22    user = library.database.getUser("user1")
23    assert book['available'] == True
24    assert book['borrowedBy'] is None
25    assert "book1" not in user['borrowedBooks']
```

Tento test využíva vývojárove znalosti interných častí systému, ako je štruktúra objektov `book` a `user` v databáze, na vytvorenie presných tvrdení o zmenách stavu počas procesu vrátenia knihy.

Ukážka 3.2.2: Transparentné testovanie

Black Box Testovanie

Black Box testovanie predstavuje typ testovania softvéru, pri ktorom sa tester nezaobrá internými znalosťami ani detailami implementácie softvéru. Namiesto toho sa zameriava na overovanie funkčnosti na základe poskytnutých špecifikácií alebo požiadaviek 3.2.3. [7] [30]

Výhody Black Box testovania:

- Testovanie je efektívne pri implementácii vo väčších systémoch, kde je komplexná štruktúra kódu.

- Testy sa vykonávajú z perspektívy koncového používateľa alebo klienta, čo zabezpečuje reálne testovanie užívateľskej skúsenosti.
- Testovacie prípady sú jednoducho reprodukovateľné a môžu byť opakovane použité.
- Pomáha identifikovať nejednoznačnosti a rozpory vo funkčných špecifikáciách systému.

👁 Black box testing požičiavania kníh

Tu je príklad black box testu pre funkčnosť požičiavania:

```

1 def testBorrowBook(self, setupLibrary):
2     """Testing book borrowing"""
3     library = setupLibrary
4     # Preparing test data
5     library.addUser("user1", "Test User", "test@test.com")
6     library.addBook("book1", "Test Book", "Test Author")
7
8     # Borrow available book
9     success, message = library.borrowBook("user1", "book1")
10    assert success == True
11    assert "successfully" in message
12
13    # Borrow unavailable book
14    success, message = library.borrowBook("user1", "book1")
15    assert success == False
16    assert "not available" in message
17
18    # Borrow nonexistent book
19    success, message = library.borrowBook("user1", "nonexistentBook")
20    assert success == False
21    assert "not found" in message

```

Tento test sa zameriava na externé správanie metódy `borrowBook`, na základe požiadaviek. Nerobí žiadne predpoklady o internej implementácii.

Vývojár píšuci white box test by mal prístup k tomuto pseudokódu a mohol by robiť špecifické tvrdenia o zmenách interného stavu (napr. o príznaku dostupnosti knihy, atď.). Tester píšuci black box test by však vedel iba o očakávaných vstupoch a výstupoch na základe požiadaviek (napr. pokus o požičanie dostupnej knihy). Toto rozlíšenie vedomostí a zamerania je kľúčový rozdiel medzi white box a black box testovaním v kontexte nášho knižničného systému.

Ukážka 3.2.3: Black-box testovanie

Gray Box Testing

Gray Box testovanie predstavuje prístup, pri ktorom má tester, na rozdiel od Black Box testovania (kde znalosť vnútornej štruktúry chýba) a White Box testovania (kde je štruktúra plne známa), iba čiastočné znalosti o vnútornej štruktúre testovaného systému. Táto čiastočná znalosť však poskytuje dostatočný prístup k interným dátovým štruktúram a algoritmom, čo umožňuje efektívny návrh relevantných testovacích prípadov 3.2.4. [5] [29] [1]

Výhody Gray Box testovania:

- Zabezpečuje jasné testovacie ciele pre používateľov aj vývojárov systému.
- Nevyžaduje rozsiahle programátorské zručnosti od testerov, čo rozširuje okruh potenciálnych testerov.
- Predstavuje nerušivý prístup k testovaniu, ktorý minimálne zasahuje do existujúcej štruktúry systému.

👁 Gray box testovanie procesu vrátenia knihy

V našom knižničnom systéme môžeme demonštrovať gray box testing pri testovaní procesu vrátenia knihy:

```
1 def testReturnBookFlow(self, setupLibrary):
2     """Test of returning the book"""
3     library = setupLibrary
4     # Preparing test data
5     library.addUser("user1", "Test User", "test@test.com")
6     library.addBook("book1", "Test Book", "Test Author")
7     library.borrowBook("user1", "book1")
8     # Verifying status before returning the book
9     book = library.database.getBook("book1")
10    assert book['available'] == False
11    # Return the book
12    success, message = library.returnBook("user1", "book1")
13    assert success == True
14    assert "successfully" in message
15    # Verify state after return
16    book = library.database.getBook("book1")
17    assert book['available'] == True
```

V tomto príklade tester využíva znalosť o interných dátových štruktúrach (dostupnosť knihy v objekte `book`), ale zároveň testuje proces vrátenia knihy primárne cez verejné rozhranie (metóda `returnBook`).

Ukážka 3.2.4: Grey-box testovanie

3.2.2 Prínosy jednotkového testovania

- **Agilný vývojový proces**

Jednotkové testovanie podporuje agilný vývoj tým, že zjednodušuje implementáciu zmien v existujúcom kóde a urýchľuje pridávanie nových funkcií. Podporuje kontinuálnu integráciu a umožňuje flexibilné úpravy bez obáv z narušenia existujúcej funkcionality.

- **Zlepšenie kvality kódu**

Proces písania jednotkových testov núti vývojárov lepšie premyslieť návrh kódu. Chyby sú identifikované a opravené v skoršej fáze vývoja.

- **Jednoduchšie zmeny a integrácia**

Testy zjednodušujú proces reštrukturalizácie a údržby kódu. Vývojári môžu bezpečne upravovať existujúci kód, pretože testy okamžite odhalia prípadné nežiadúce vedľajšie účinky zmien.

- **Dostupná dokumentácia**

Jednotkové testy slúžia ako aktuálna dokumentácia kódu. Poskytujú praktické príklady použitia, ktoré pomáhajú novým vývojárom rýchlo pochopiť funkcionality existujúceho kódu.

- **Nižšie náklady na vývoj**

Včasná detekcia chýb znižuje celkové náklady na vývoj. Oprava problémov v skorých fázach je podstatne lacnejšia než riešenie chýb objavených v produkcii.

- **Demonštrácia kompletnosti kódu**

Testy poskytujú merateľný dôkaz o pokrytí kódu a jeho funkčnosti. V agilnom vývoji umožňujú sledovať progres a potvrdzujú pripravenosť kódu na integráciu.

- **Merateľné pokrytie kódu**

Jednotkové testy umožňujú kvantitatívne meranie kvality testovania. Pomáhajú identifikovať netestované časti aplikácie a poskytujú objektívne hodnotenie kvality softvéru.

3.2.3 Osvedčené postupy

Pri vytváraní jednotkových testov je potrebné dodržiavať nasledujúce princípy: (1) Testovanie jednej jednotky kódu, (2) Písanie malých a izolovaných testov, (3) Zabezpečenie rýchleho vykonávania testov, (4) Opakovateľnosť testov, (5) Zrozumiteľné pomenovanie testov, (6) Pravidelné spúšťanie testov

3.3 Integračné testovanie

Tento typ testovania sa zameriava na systematické spájanie a testovanie jednotlivých modulov systému, ktoré boli predtým samostatne otestované pomocou jednotkových testov. Hlavným cieľom je overiť, či tieto komponenty dokážu efektívne spolupracovať a komunikovať v rámci celého systému [3.3.1](#).

V modernom vývoji softvéru sa aplikácie zvyčajne skladajú z množstva vzájomne prepojených komponentov. Integračné testovanie vzniklo ako odpoveď na potrebu zabezpečiť, že tieto samostatne vyvíjané časti budú po spojení fungovať ako harmonický celok. Na

rozdiel od jednotkového testovania, ktoré sa zameriava na izolované časti kódu, integračné testovanie skúma vzájomné interakcie medzi komponentmi a ich schopnosť spolupracovať v reálnych podmienkach.

V praxi často nastávajú situácie, kedy moduly, ktoré fungujú bezchybne samostatne, zlyhávajú pri vzájomnej interakcii. Toto môže byť spôsobené rozdielmi v interpretácii špecifikácií medzi vývojármi, nekompatibilitou vo formátoch dát, alebo nezohľadnením všetkých možných scenárov pri vývoji jednotlivých komponentov.

Významným faktorom je aj častá transformácia dát medzi modulmi. Integračné testovanie pomáha odhaliť prípadné problémy v týchto transformáciách skôr, než sa dostanú do produkčného prostredia. [36]

🕒 Integračný test sekvenčného vypožičiavania

V kontexte nášho knižničného systému predstavuje integračné testovanie overenie spolupráce medzi hlavnými komponentmi `LibrarySystem` a `Database`. Napríklad, testovanie sekvenčného vypožičiavania tej istej knihy viacerými používateľmi:

```
1 def testSequentialBookBorrowing(self, setupLibrary):
2     """Test sequential borrowing of the same book by different users"""
3     library = setupLibrary
4     # Setup
5     users = ["user5", "user6", "user7"]
6     for user in users:
7         library.addUser(user, f"User {user}", f"{user}@example.com")
8     library.addBook("popularBook", "Popular Book", "Famous Author")
9     # Sequential borrowing and returning
10    for user in users:
11        success, _ = library.borrowBook(user, "popularBook")
12        assert success == True
13        book = library.database.getBook("popularBook")
14        assert book['borrowedBy'] == user
15        assert not book['available']
16        success, _ = library.returnBook(user, "popularBook")
17        assert success == True
18        book = library.database.getBook("popularBook")
19        assert book['borrowedBy'] is None
20        assert book['available']
```

Ukážka 3.3.1: Integračné testovanie

3.3.1 Metodologické prístupy

V oblasti integračného testovania rozlišujeme dva základné metodologické prístupy k testovaniu softvérových komponentov. Každý z nich ponúka špecifické výhody a prináša vlastné výzvy pre testovacie tímy. V praxi sa často využíva aj ich kombinácia.

Bottom-up prístup

Bottom-up metodológia začína testovanie na najnižšej úrovni systému a postupuje smerom nahor. Základné komponenty sa postupne integrujú do väčších celkov, pričom sa využívajú testovacie ovládače (*drivers*) na simuláciu správania vyšších vrstiev systému.

Hlavnými výhodami tohto prístupu sú skorá detekcia chýb v základných komponentoch a možnosť paralelného testovania modulov. Medzi nevýhody však patrí neskoršie overovanie celkovej funkcionality systému a náročnejšia príprava aj údržba potrebných testovacích ovládačov.

Top-down prístup

Top-down metodológia začína na najvyššej úrovni aplikácie a postupne integruje nižšie vrstvy. Využíva zástupcov (*stubs*) na simuláciu správania ešte neintegrovaných nižších modulov.

Výhodami tohto prístupu sú skoré testovanie hlavnej logiky aplikácie, rýchla identifikácia architektonických problémov a jednoduchšia demonštrácia funkcionality. Nevýhodami sú však komplexné vytváranie a údržba zástupcov, ako aj vyššie časové a zdrojové nároky pri simulácii komplexných modulov.

Kombinovaný prístup

V praxi sa často využíva kombinovaný prístup (sendvičové testovanie), ktorý spája výhody oboch metodológií. Umožňuje súbežné testovanie z oboch smerov, čím optimalizuje proces integrácie. Je vhodný najmä pre rozsiahle projekty, ale vyžaduje dôkladné plánovanie a väčšie množstvo zdrojov na koordináciu testovacích tímov.

Big Bang prístup

Big Bang prístup predstavuje metódu, pri ktorej sa všetky moduly integrujú naraz, bez postupnej integrácie jednotlivých častí. Systém sa testuje ako celok až po spojení všetkých komponentov. Tento prístup je jednoduchý na implementáciu, ale prináša určité riziká pri odhaľovaní a riešení problémov.

Hlavnou výhodou tohto prístupu je jeho vhodnosť pre malé systémy s jednoduchšou architektúrou. Nevýhody však zahŕňajú náročnú identifikáciu problematických modulov, obmedzený čas na testovanie z dôvodu dlhej integračnej fázy, ako aj nemožnosť dôkladne otestovať jednotlivé moduly v izolácii.

3.3.2 Proces implementácie

Úspešná implementácia integračného testovania predstavuje komplexný proces, ktorý vyžaduje systematický prístup a dôkladné plánovanie.

Plánovacia fáza

V plánovacej fáze sa vykonáva analýza systému a jeho komponentov. Testerský tím identifikuje všetky moduly určené na integráciu a vytvára mapu ich vzájomných závislostí. Na základe tejto analýzy sa stanovuje optimálne poradie integrácie jednotlivých komponentov a definujú sa kritériá úspechu pre každý integračný krok. Súčasťou plánovania je aj identifikácia potenciálnych rizík a stanovenie časového harmonogramu.

Prípravná fáza

Prípravná fáza sa zameriava na vytvorenie testovacieho prostredia a prípravu potrebných zdrojov. Kľúčovou aktivitou je vybudovanie prostredia, ktoré simuluje reálne podmienky prevádzky systému. V tejto fáze sa pripravujú testovacie dáta a scenáre, ktoré pokrývajú štandardné prípady použitia aj hraničné situácie. Dôležitou súčasťou je aj príprava nástrojov pre monitorovanie a záznam priebehu testov.

Vykonávacia fáza

Počas vykonávacej fázy prebieha samotná integrácia modulov a ich testovanie podľa pripraveného plánu. Každý integračný krok zahŕňa spojenie príslušných modulov, vykonanie pripravených testov a dokumentáciu výsledkov. Dôraz sa kladie na systematické monitorovanie priebehu testovania a zaznamenávanie všetkých odchýlok od očakávaného správania.

Analytická a záverečná fáza

V záverečných fázach sa vyhodnocujú výsledky testov a pripravuje sa potrebná dokumentácia. Testerský tím analyzuje zaznamenané problémy, klasifikuje ich podľa závažnosti a pripravuje reporty pre manažment. Súčasťou je aj vypracovanie odporúčaní pre budúce integračné projekty a finalizácia dokumentácie potrebnej pre nasadenie systému do produkčného prostredia.

3.4 Regresné testovanie

Regresné testovanie je technika testovania softvéru, ktorá opakovane vykonáva funkčné **3** a nefunkčné testy **4**, aby sa zabezpečilo, že softvérová aplikácia funguje tak, ako má, po akýchkoľvek zmenách, aktualizáciách, revíziách, vylepšeniach alebo optimalizáciách kódu. Z technického hľadiska môže zmena kódu spôsobiť zmenu závislostí, čo vedie k chybám alebo nefunkčnosti softvéru. Regresné testovanie sa zavádza s cieľom minimalizovať takéto riziká. [35] [26]

3.4.1 Techniky regresného testovania

Techniky regresného testovania zahŕňajú niekoľko prístupov. Úplné regresné testovanie spočíva v spustení všetkých dostupných testov po vykonaní zmien, čo poskytuje najvyššiu úroveň istoty, avšak za cenu časovej náročnosti a potenciálnej neefektívnosti pri rozsiahlych aplikáciách. Selektívne regresné testovanie predstavuje prístup, pri ktorom sa vyberajú len testy relevantné pre vykonané zmeny, čo vyžaduje dôkladnú analýzu závislostí v kóde, no výrazne znižuje čas potrebný na testovanie. Prioritizácia testov organizuje testy podľa ich dôležitosti a pravdepodobnosti odhalenia chýb, pričom kritické testy sú spúšťané ako prvé, umožňujúc skorú identifikáciu potenciálnych problémov **3.4.1**.

🕒 Regresný test po pridaní restore funkcionality

V našom knižničnom systéme sme implementovali novú funkcionality pre obnovu (restore) vymazaných entít. Táto funkcionality úzko súvisí s existujúcou funkciou mazania (delete). Pridanie novej funkcionality vyžaduje dôkladné regresné testovanie na zabezpečenie, že pôvodná funkcionality zostáva nedotknutá.

```
1 def test_delete_restore_relationship(self, setup_library):
2     """Test, that delete works properly after adding restore"""
3     library = setup_library
4
5     # Test book creation
6     library.addBook("test_book", "Test Book", "Test Author")
7
8     # Verify delete functionality
9     delete_success, _ = library.deleteBook("test_book")
10    assert delete_success == True
11    assert library.database.getBook("test_book") is None
12
13    # Restore book
14    library.restoreBook("test_book")
15
16    # Test delete after restoration
17    delete_success, _ = library.deleteBook("test_book")
18    assert delete_success == True
19    assert library.database.getBook("test_book") is None
```

Tento test demonštruje regresné testovanie v praxi - overuje, že pôvodná funkcionality mazania zostáva plne funkčná aj po implementácii novej súvisiacej funkcionality obnovy.

Ukážka 3.4.1: Regresné testovanie

3.4.2 Regresné testovanie pri rozširovaní funkcionality

Pri pridávaní novej funkcionality do existujúceho systému je regresné testovanie zvlášť dôležité. Vo všeobecnosti sú potrebné tri typy testov:

1. Testy nových funkcionality - overujú správnu implementáciu pridanej funkcionality
2. Testy existujúcich funkcionality - potvrdzujú, že existujúce funkcie neboli narušené
3. Integračné testy - overujú správnu spoluprácu medzi novými a existujúcimi funkcionality

Kapitola 4

Nefunkčné testovanie

Nefunkčné testovanie predstavuje kľúčový typ testovania softvéru, ktorý sa zameriava na overenie nefunkčných požiadaviek aplikácie. Jeho hlavným cieľom je kontrola, či správanie systému zodpovedá stanoveným nefunkčným parametrom a požiadavkám. [12] [27]

4.1 Charakteristika nefunkčného testovania

Tento typ testovania pokrýva všetky aspekty systému, ktoré nie sú súčasťou funkčného testovania. Zameriava sa na overenie kvalitatívnych charakteristík softvéru, ako sú výkon, bezpečnosť, použiteľnosť a spoľahlivosť 4.2.1.

4.2 Význam v procese testovania

Nefunkčné testovanie je rovnako dôležité ako funkčné testovanie, pretože:

- Zabezpečuje kvalitu systému nad rámec základnej funkcionality
- Hodnotí systém z pohľadu výkonu a spoľahlivosti
- Kontroluje aspekty, ktoré priamo ovplyvňujú používateľskú skúsenosť

Nefunkčné testovanie tak tvorí neoddeliteľnú súčasť komplexného procesu zabezpečenia kvality softvéru, pričom sa zameriava na aspekty, ktoré priamo ovplyvňujú celkovú kvalitu a použiteľnosť aplikácie.



V našom knižničnom systéme si ukážeme praktické príklady rôznych typov nefunkčného testovania. Naš knižničný systém musí spĺňať niekoľko nefunkčných požiadaviek:

- Výkonnostná efektívnosť pri práci s veľkým katalógom kníh
- Bezpečné spracovanie používateľských údajov
- Škálovateľnosť pri zvyšujúcom sa počte používateľov a kníh

Ukážka 4.2.1: Nefunkčné testovanie

4.3 Bezpečnostné testovanie

Bezpečnostné testovanie predstavuje špecializovaný typ testovania, ktorý vykonáva špeciálny tím s cieľom overiť odolnosť systému voči rôznym typom bezpečnostných hrozieb. Zameriava sa na hodnotenie zabezpečenia softvéru, aplikácií alebo webových stránok proti interným aj externým hrozbám 4.3.1.

Bezpečnostné testovanie sa zameriava na niekoľko kľúčových oblastí, ktoré zahŕňajú odolnosť voči škodlivému softvéru a vírusom, hodnotenie bezpečnosti autorizačných a autentifikačných procesov, a správanie systému pri pokusoch o neoprávnený prístup. Dôraz sa kladie tiež na ochranu dát po prípadnom bezpečnostnom incidente a na zachovanie integrity systému pri čelení rôznym typom útokov. [19]

🔍 Bezpečnostný test kontroly prístupu

V kontexte nášho knižničného systému sa bezpečnostné testovanie môže zameriavať najmä na:

- Správu prístupových práv neaktívnych používateľov
- Ochranu pred neautorizovaným prístupom k zdrojom

Príklad implementácie bezpečnostných testov:

```
1 def testInactiveUserRestrictions(self, setupLibrary):
2     """Test that inactive user cannot perform operations"""
3     library = setupLibrary
4     # Get inactive user from test data
5     inactiveUser = 'user4' # Known inactive user
6     # Attempt to borrow book with inactive account
7     success, message = library.borrowBook(inactiveUser, 'book6')
8     assert success == False
9     assert "inactive" in message.lower()
10    # Attempt to edit user details
11    success, message = library.editUser(
12        inactiveUser,
13        name='New Name'
14    )
15    assert success == True # Edit should still work
16    # Reactivate account
17    success, _ = library.editUser(inactiveUser, active=True)
18    assert success == True
19    # Verify user can now borrow
20    success, message = library.borrowBook(inactiveUser, 'book6')
21    assert success == True
```

Ukážka 4.3.1: Bezpečnostné testovanie

4.3.1 Penetračné testovanie

Penetračné testovanie (Pen testing) predstavuje špecifickú formu bezpečnostného testovania, pri ktorej sa vykonáva autorizovaný kybernetický útok na systém. Jeho hlavným cieľom je identifikácia bezpečnostných slabín systému.

Penetračné testy typicky vykonávajú externí dodávatelia (etickí hackeri) na základe písomného povolenia, pričom zahŕňajú rôzne typy testovacích operácií ako `SQL injection`, manipuláciu URL, eskaláciu privilégií a testovanie expirácie relácie.

Pri realizácii penetračného testovania je potrebné dodržiavať prísne bezpečnostné pravidlá: (1) Testovanie sa môže vykonávať len v autorizovanom prostredí, (2) Vyžaduje sa písomný súhlas zodpovedných osôb, (3) Testovanie musí prebiehať podľa vopred stanovených postupov a všetky výsledky testovania musia byť dôkladne zdokumentované.

4.3.2 Výkonnostné testovanie

Výkonnostné testovanie predstavuje proces overovania stability a reakčného času aplikácie pri aplikovaní záťaže 4.3.2. Stabilita v tomto kontexte znamená schopnosť aplikácie udržať funkčnosť pod záťažou, zatiaľ čo reakčný čas vyjadruje rýchlosť odozvy aplikácie pre používateľov. Toto testovanie sa realizuje pomocou špecializovaných nástrojov. Zdroje pre výkonnostné testovanie [18] [14] [17]. Medzi najpodstatnejšie typy výkonnostného testovania patria:

- **Záťažové testovanie** - Overuje stabilitu a reakčný čas aplikácie pri záťaži, ktorá zodpovedá alebo je nižšia ako projektovaný počet používateľov. Cieľom je overiť, či aplikácia dodržiava stanovené parametre odozvy pri očakávanej záťaži.
- **Záťažové stresové testovanie** - Skúma správanie aplikácie pri záťaži prekračujúcej projektovaný počet používateľov. Zameriava sa na: (1) Overenie stability systému pri extrémnej záťaži, (2) Analýzu správania pri prekročení bežných prevádzkových podmienok, (3) Identifikáciu limitov systému
- **Testovanie škálovateľnosti** - Zameriava sa na identifikáciu hraničných bodov výkonu systému postupným zvyšovaním záťaže. Sleduje sa: (1) Zmena v reakčnom čase pri rastúcej záťaži, (2) Bod zlyhania systému, (3) Maximálnu kapacitu pred degradáciou výkonu

Typický príklad sledovaných parametrov:

- Štandardná záťaž - optimálna odozva
 - Stredná záťaž - akceptovateľná odozva
 - Vysoká záťaž - degradácia výkonu
 - Kritický bod - zlyhanie systému
- **Objemové testovanie** - Zameriava sa na overenie stability a reakčného času aplikácie pri prenose veľkého objemu dát do databázy. Hlavným cieľom je testovanie kapacity databázy a jej schopnosti spracovať veľké množstvo dát.
 - **Vytrvalostné testovanie** - Overuje stabilitu a reakčný čas aplikácie pri dlhodobom zaťažení. Zameriava sa na: (1) Detekciu pamäťových únikov, (2) Overenie stability pri dlhodobej prevádzke, (3) Identifikáciu degradácie výkonu v čase, (4) Analýzu správania systému pri nepretržitej prevádzke

🕒 Pseudo-kód výkonnostného testu knižničného systému pomocou Locust

Pre náš knižničný systém je kritické efektívne spracovanie operácií s knihami a používateľmi. Implementujeme výkonnostné testy na overenie: (1) Rýchlosti vyhľadávania v katalógu, (2) Výkonu pri hromadných operáciách, (3) Správania systému pri veľkom počte záznamov

Príklad možnej implementácie výkonnostných testov:

```
1 from locust import HttpUser, task, between
2 class LibraryLoadTest(HttpUser):
3     """Load simulation for the library system
4     wait_time = between(1, 3) # Pause between requests
5     def on_start(self):
6         """User initialization"""
7         self.client.post("/login", {
8             "username": "test_user",
9             "password": "test_pass"
10        })
11    @task(3) # Weight 3 - most frequent operation
12    def search_books(self):
13        """Searching the catalog"""
14        self.client.get("/api/books?query=python")
15    @task(2)
16    def borrow_book(self):
17        """Borrowing a book"""
18        self.client.post("/api/books/borrow", {
19            "book_id": "book1",
20            "user_id": self.user_id
21        })
22    @task(1)
23    def return_book(self):
24        """Returning a book"""
25        self.client.post("/api/books/return", {
26            "book_id": "book1",
27            "user_id": self.user_id
28        })
29 # Usage: locust -f load_test.py --users 100 --spawn-rate 10
30 # Simulates 100 concurrent users with a ramp-up rate of 10 per second
```

Tento príklad používa nástroj Locust na simuláciu viacerých súbežných používateľov, ktorí vykonávajú rôzne operácie v knižničnom systéme. Test meria odozvu systému pri rôznych úrovniach záťaže a sleduje výkonnostné metriky ako priemernú dobu odozvy a počet požiadaviek za sekundu.

Ukážka 4.3.2: Výkonnostné testovanie

4.3.3 Testovanie použiteľnosti

Testovanie použiteľnosti predstavuje proces hodnotenia aplikácie z perspektívy používateľa, so zameraním na užívateľský zážitok, intuitívnosť ovládania a celkovú prívetivosť rozhrania. Tento typ testovania sa zameriava na zabezpečenie optimálnej interakcie medzi používateľom a aplikáciou 4.3.3. Medzi typy testovania použiteľnosti patria:

- **Prieskumné testovanie** Prieskumné testovanie (Exploratory Testing) predstavuje neformálny prístup k testovaniu, ktorý vykonáva testovací tím. Charakteristické črty zahŕňajú: (1) Využitie znalostí biznis domény pri testovaní, (2) Flexibilný prístup k objavovaniu defektov, (3) Použitie testovacích plánov ako vodítka, (4) Systematické skúmanie rôznych aspektov aplikácie.
- **Testovanie medzi prehliadačmi** Cross-browser testovanie overuje funkčnosť, výkon a vzhľad aplikácie naprieč rôznymi platformami:
 - Rôzne webové prehliadače
 - Rôzne operačné systémy
 - Rôzne mobilné zariadenia

Hlavné ciele tohto testovania:

- Zabezpečenie konzistentného používateľského zážitku
 - Overenie kompatibility na rôznych platformách
 - Identifikácia problémov špecifických pre jednotlivé prehliadače
 - Optimalizácia výkonu na rôznych zariadeniach
- **Testovanie prístupnosti** Testovanie prístupnosti (Accessibility Testing) sa zameriava na overenie dostupnosti softvéru pre používateľov s rôznymi formami zdravotného znevýhodnenia. Hlavné oblasti testovania:
 - Zrakové postihnutie - je potreba sa zamerať najmä a veľkosť písma, kontrast farieb a kompatibilitu s čítačkami obrazovky
 - Sluchové postihnutie - je potreba si všímať alternatívne textové popisy pre zvukový obsah a vizuálne signály pre zvukové upozornenia
 - Motorické obmedzenia - zamerať sa hlavne na ovládanie klávesnicou a prispôbenie časových limitov
 - Kognitívne obmedzenia - produkt by mal mať jednoduchú navigáciu a zrozumiteľné používateľské rozhranie

Kľúčové aspekty testovania použiteľnosti

Pri testovaní použiteľnosti sa hodnotia nasledujúce prvky: (1) Intuitívnosť používateľského rozhrania, (2) Efektívnosť navigácie, (3) Konzistencia dizajnu, (4) Čitateľnosť a zrozumiteľnosť obsahu, (5) Responzivnosť a prispôsobivosť rôznym zariadeniam, (6) Dostupnosť kľúčových funkcií, (7) Celková užívateľská spokojnosť

Testy použiteľnosti knižničného systému

```
1 def testUserInterfaceFlows(self):
2     """Simulation of user scenarios for usability testing"""
3     # Test 1: Book search
4     usabilityMetrics = {
5         "clicksToFindBook": 0,
6         "timeToComplete": 0,
7         "navigationErrors": 0
8     }
9     def simulateUserBookSearch():
10         # 1. Track metrics: clicks needed, completion time, navigation
            errors
11         # 2. Click search button
12         # 3. Enter search term "Python programming"
13         # 4. Simulate user error (clicking wrong button)
14         # 5. Navigate back from error
15         # 6. Submit search
16         # 7. Record total clicks and completion time
17     # Test 2: Borrowing process
18     def simulateBorrowingProcess():
19         # 1. Measure time of each borrowing step: find_book, select_book,
            confirm_borrow, etc.
20         # 2. Record duration of each step
21         # 3. Check if any step takes more than 30 seconds (too long)
22         # 4. Fail test if any step is too slow
23     # Test 3: Accessibility for different devices
24     def testResponsiveDesign():
25         # 1. Test on multiple screen sizes: desktop, tablet, mobile
26         # 2. Verify search box is visible on all devices
27         # 3. Confirm borrow button is clickable
28         # 4. Check text font size meets minimum requirements (12px
            mobile, 14px others)
```

Tento príklad demonštruje:

- Meranie efektivity používateľského rozhrania (počet kliknutí, čas)
- Sledovanie a analýzu používateľských chýb
- Testovanie prístupnosti na rôznych zariadeniach
- Vyhodnocovanie plynulosti procesov vypožičiavania

Ukážka 4.3.3: Testovanie použiteľnosti

Hoci je možné automatizovať niektoré aspekty testovania použiteľnosti, ako je demonštrované v ukážke vyššie, je dôležité poznamenať, že skutočné testovanie použiteľnosti je

najefektívnejšie pri manuálnom testovaní s reálnymi používateľmi. Automatizované testy nedokážu zachytiť skutočné kognitívne procesy používateľov, ich emócie, frustrácie či neočakávané spôsoby interakcie so systémom. Manuálne testovanie použiteľnosti umožňuje sledovať neverbálne reakcie používateľov, získavať kvalitatívnu spätnú väzbu a identifikovať problémy s použiteľnosťou, ktoré by automatizované testy mohli prehliadnúť.

4.3.4 Testovanie kompatibility

Testovanie kompatibility predstavuje nefunkčné testovanie zamerané na overenie schopnosti softvérovej aplikácie fungovať v rôznych prostrediach a na rôznych platformách. Cieľom je zabezpečiť konzistentné správanie aplikácie naprieč rôznymi konfiguráciami a zariadeniami

4.3.4. Najpoužívanéjšie typy testovania kompatibility:

- **Testovanie kompatibility prehliadačov** - Overuje funkčnosť aplikácie v rôznych webových prehliadačoch a ich verziách. Kľúčové aspekty:
 - Testovanie na populárnych prehliadačoch (Chrome, Firefox, Safari, Edge)
 - Kontrola konzistentnosti používateľského rozhrania
 - Overenie funkčnosti všetkých interaktívnych prvkov
 - Testovanie responzívneho dizajnu
- **Hardvérová kompatibilita** - Zameriava sa na overenie kompatibility s rôznymi hardvérovými konfiguráciami:
 - Rôzne procesorové architektúry
 - Pamäťové konfigurácie
 - Grafické adaptéry
 - Periférne zariadenia
- **Sieťová kompatibilita** - Testuje správanie aplikácie v rôznych sieťových podmienkach:
 - Rôzne typy pripojenia (Wi-Fi, 3G, 4G, 5G)
 - Rôzne rýchlosti pripojenia
 - Stabilita pri nestabilnom pripojení
- **Kompatibilita mobilných zariadení** - Overuje fungovanie aplikácie na rôznych mobilných platformách:
 - Android zariadenia
 - iOS zariadenia
 - Rôzne veľkosti obrazoviek
 - Rôzne verzie operačných systémov
- **Verzie softvéru** - Testovanie kompatibility verzií sa delí na dve hlavné kategórie:
 - **Spätná kompatibilita:** Testovanie funkčnosti v starších verziách
 - **Dopredná kompatibilita:** Testovanie funkčnosti v novších verziách

Časté problémy kompatibility

Pri testovaní kompatibility sa najčastejšie identifikujú nasledujúce typy problémov: (1) Problémy s veľkosťou a typom písma, (2) Problémy so zarovnaním prvkov, (3) Zmeny v štýloch CSS, (4) Problémy s posuvníkmi, (5) Prekrývanie obsahu, (6) Narušené tabuľky alebo rámy

Implementácia testovania

Pri implementácii testovania kompatibility je potrebné: (1) Analyzovať požiadavky zákazníka na podporované platformy, (2) Využívať analytické nástroje pre identifikáciu najpoužívanejších platforiem, (3) Testovať v reálnom, nie virtuálnom prostredí, (4) Pravidelne vykonávať testy na nových verziách podporovaných platforiem

🔗 Testy kompatibility prehliadačov

V našom knižničnom systéme testovanie kompatibility môže zahŕňať napr. overenie správnej funkčnosti systému v rôznych webových prehliadačoch ako je ilustrované v príklade:

```
1 def test_browser_compatibility(self, setup_library):
2     """Test of system compatibility in different browsers"""
3     browsers = {
4         "Chrome": {"version": "120.0"},
5         "Firefox": {"version": "122.0"},
6         "Safari": {"version": "17.0"},
7         "Edge": {"version": "120.0"}
8     }
9     test_scenarios = [
10         "book_search",
11         "book_detail",
12         "book borrow"
13     ]
14     for browser, info in browsers.items():
15         for scenario in test_scenarios:
16             result = library.test_scenario(
17                 browser=browser,
18                 version=info["version"],
19                 scenario=scenario
20             )
21             assert result.is_successful,
22                 f"Scenario '{scenario}' failed in {browser}
23                     {info['version']}"
```

Ukážka 4.3.4: Testovanie kompatibility

Kapitola 5

Testovacie metódy

V oblasti testovania softvéru rozlišujeme dva základné prístupy k testovaniu: manuálne a automatizované. Každý z týchto prístupov má svoje špecifické použitie a výhody v procese zabezpečenia kvality softvéru.

5.1 Manuálne testovanie

Manuálne testovanie predstavuje proces, pri ktorom tester priamo interaguje s aplikáciou a vykonáva testy podľa preddefinovaných testovacích prípadov. Hlavné výhody manuálneho testovania:

- Presná vizuálna spätná väzba pri testovaní používateľského rozhrania
- Nižšie vstupné náklady na implementáciu
- Nevyžaduje znalosti programovania
- Vysoká flexibilita pri neplánovaných zmenách v aplikácii

5.2 Automatizované testovanie

Automatizované testovanie využíva špecializované nástroje a skripty na vykonávanie testov. Tento prístup je vhodný najmä pre opakované a časovo náročné testovacie scenáre. Kľúčové výhody automatizovaného testovania:

- Efektívne vykonávanie opakovaných testov
- Vyššia spoľahlivosť a konzistentnosť výsledkov
- Rozsiahlejšie pokrytie testami
- Minimalizácia chýb spôsobených ľudským faktorom

5.3 Porovnanie metód

Pre lepšie pochopenie rozdielov medzi oboma prístupmi uvádzame porovnanie kľúčových parametrov:

Tabuľka 5.1: Porovnanie testovacích metód

Parameter	Manuálne testovanie	Automatizované testovanie
Vykonávanie	Realizované ľudským testerom	Vykonávané pomocou testovacích nástrojov
Časová náročnosť	Vyššia časová náročnosť	Rýchlejšie po počiatočnej implementácii
Potrebné zdroje	Primárne ľudské zdroje	Automatizačné nástroje a vyškolený personál
Prieskumné testovanie	Možné a efektívne	Obmedzené možnosti

Každá z metód má svoje špecifické využitie v procese testovania. Manuálne testovanie je nenahraditeľné pri testovaní používateľského rozhrania a exploratívnom testovaní, zatiaľ čo automatizované testovanie je ideálne pre regresné testy a opakované testovacie scenáre.

Kapitola 6

Testovanie webových aplikácií

Testovanie webových aplikácií je proces overovania, či webové aplikácie a stránky fungujú podľa očakávaní. Testerí webových aplikácií používajú rôzne metódy ich testovanie, vrátane manuálneho testovania, automatizovaného testovania a záťažového testovania.

Overovaním správneho fungovania webových aplikácií môžu testerí pomôcť zabezpečiť, aby používatelia mali pozitívnu skúsenosť pri ich používaní. Časté problémy pri testovaní webových aplikácií:

- **Interakcia s firewall** - Problémy s pripojením s firewall vznikajú, pretože port môže blokať webovú aplikáciu kvôli problémom s dodržiavaním bezpečnostných certifikátov.
- **Autentifikácia webových služieb** - Moderné webové aplikácie vo veľkej miere využívajú vrstvy webových služieb, ako sú XML/SOAP alebo JSON/REST, na výmenu údajov medzi aplikáciami alebo systémami. Preto sa výrazne zvýšila potreba autentifikácie týchto webových služieb.
- **Konzistencia medzi prehliadačmi** - Zákazníci vždy uprednostňujú škálovateľné a vysoko interaktívne webové aplikácie. Ak zákazníci narazia na nekonzistentnosť pri prehliadaní v rôznych prehliadačoch, môže to mať negatívny vplyv na rast podniku. Preto vývojári a testerí musia pri testovaní použiteľnosti sledovať škálovateľnosť a interaktivitu aplikácie v rôznych prehliadačoch a hardvérových konfiguráciách.
- **Riešenie problémov s výkonom** - Používatelia opúšťajú každú pomaly sa načítavajúcu webovú aplikáciu, čo v konečnom dôsledku väčšinou ovplyvňuje príjmy spoločnosti. Na prekonanie tejto prekážky by vývojári mali identifikovať faktory ovplyvňujúce testovanie výkonu, ako sú škálovateľnosť funkcií aplikácie, kompatibilita a integračné obmedzenia.
- **Zabezpečenie webovej aplikácie proti únikom údajov** - Počet kybernetických hrozieb, ktoré môžu uniknúť citlivé údaje používateľov, neustále rastie. Je nevyhnutné monitorovať procesy testovania bezpečnosti, aby sa zabránilo strate informácií a únikom údajov. To pomôže predchádzať DDoS útokom a iným kybernetickým hrozbám.

Následne si predstavíme najpoužívanější nástroje na testovanie webových aplikácií:

6.1 Najlepšie nástroje na testovanie webových aplikácií

- **Cypress** - Cypress je komplexný testovací framework, ktorý umožňuje vývojárom písať a spúšťať testy priamo v prehliadači. Cypress poskytuje funkciu automatického čakania, ktorá čaká, kým aplikácia dokončí vykresľovanie pred vykonaním ďalšieho príkazu, čím sa eliminuje potreba časových limitov. Cypress tiež ponúka ladenie cestovania v čase, ktoré umožňuje vývojárom vidieť stav aplikácie v ľubovoľnom bode počas vykonávania testu. Okrem toho Cypress podporuje zaslepovanie a mockovanie siete, čo uľahčuje testovanie hraničných prípadov a scenárov zlyhania.
- **Playwright** - Playwright je open-source knižnica Node.js na automatizáciu webových prehliadačov. Jednou z najlepších funkcií Playwright je podpora viacerých prehliadačov vrátane Chromium, Firefox a WebKit, čo umožňuje testovanie medzi prehliadačmi. Playwright tiež ponúka zachytávanie mriežky, čo umožňuje vývojárom upravovať dotazy a odpovede počas vykonávania testu.
- **Katalon Studio** - Katalon Studio je **open-source** framework pre automatizované testovanie, ktorý podporuje testovanie webov, mobilných aplikácií a API. Katalon Studio podporuje vizuálny editor testov, ktorý umožňuje netechnickým používateľom vytvárať a spúšťať testy. Tiež podporuje testovanie založené na kľúčových slovách, čo umožňuje používateľom písať testy pomocou syntaxe prirodzeného jazyka.
- **Tosca** - Tosca je nástroj na automatizované testovanie, ktorý podporuje komplexné testovanie webových aplikácií. Tosca prichádza s prístupom k testovaniu založeným na modeloch, čo umožňuje používateľom vytvárať testovacie prípady bez akýchkoľvek programovacích zručností. Tosca tiež ponúka automatizovanú analýzu rizík, ktorá identifikuje najkritickejšie oblasti aplikácie na testovanie.
- **Selenium** - Selenium je populárny open-source nástroj na automatizované testovanie, ktorý podporuje komplexné testovanie webových aplikácií. Jednou z jedinečných funkcií Selenia je podpora viacerých programovacích jazykov vrátane Javy, Pythonu a Ruby. Selenium tiež ponúka automatické čakanie, kde čaká, kým aplikácia dokončí vykresľovanie pred vykonaním ďalšieho príkazu. [2]

Zdroje použité v kapitole [28].

Kapitola 7

Kontajnerizácia a Monitorovanie

Kontajnerizácia predstavuje významný pokrok v stratégiách vývoja a nasadzovania softvéru. Na rozdiel od tradičnej virtualizácie, ktorá zahŕňa spúšťanie kompletných operačných systémov, kontajnerizácia zapuzdruje iba aplikáciu a jej závislosti, pričom zdieľa jadro hostiteľského počítača. Výsledkom sú ľahké, prenosné a konzistentné prostredia naprieč rôznymi fázami vývoja.

Pre knižničný systém v tejto práci bol **Docker** zvolený ako platforma pre kontajnerizáciu vďaka jeho širokému prijatiu, komplexnému ekosystému a vynikajúcej podpore nástrojov. Kontajnerizovaný prístup poskytuje pre našu aplikáciu niekoľko kľúčových výhod: [23] [38]

- **Konzistentnosť** - Zabezpečuje, že aplikácia beží identicky vo vývojovom, testovacom a produkčnom prostredí
- **Izolácia** - Zapuzdruje aplikáciu s jej závislosťami, čím zabraňuje konfliktom s iným softvérom
- **Škálovateľnosť**: Uľahčuje horizontálne škálovanie pri nasadení v orchestračných systémoch ako **Kubernetes**
- **Efektívnosť zdrojov** - Vyžaduje menej zdrojov v porovnaní s tradičnými virtuálnymi strojmi
- **Zjednodušené nasadenie** - Zefektívňuje proces nasadenia prostredníctvom štandardizovaných obrazov kontajnerov

7.1 Fluent Bit

Fluent Bit je odľahčený a vysoko výkonný nástroj na zber a spracovanie logov, ktorý slúži na riešenie problému efektívneho manažmentu logov v moderných infraštruktúrach. Jeho hlavným účelom je zbierať, filtrovať a presmerovávať logy z rôznych zdrojov do centrálnych úložísk pre ďalšiu analýzu.

Fluent Bit sa primárne využíva na:

- **Zber logov** z kontajnerizovaných prostredí, najmä v **Kubernetes** a **Docker** ekosystémoch, kde dokáže efektívne zbierať logy zo všetkých kontajnerov.
- **Spracovanie telemetrických údajov** pre monitorovacie riešenia, kde plní úlohu spoľahlivého kolektora metrík.

- **Centralizované loggovanie** v distribuovaných systémoch, čo umožňuje konsolidovať logy z rôznych služieb na jednom mieste.
- **Transformáciu formátov dát** medzi rôznymi systémami, čím zjednodušuje integráciu heterogénnych prostredí.

Nasadzuje sa ako agent na každom uzle infraštruktúry, kde zbiera lokálne generované logy a odosiela ich do centrálného agregáčného bodu. Tento prístup umožňuje efektívne škálovanie bez potreby centralizovaného kolektora, ktorý by mohol predstavovať úzke hrdlo systému. V praxi sa tak vytvára distribuovaná sieť Fluent Bit agentov, ktorá zabezpečuje robustný systém pre manažment logov. [8]

Kapitola 8

Testovacie frameworky v jazyku Python

Python poskytuje niekoľko testovacích rozhraní, ktoré sa líšia prístupom a funkcionalitou. Každé z nich má svoje špecifické využitie a výhody.

8.1 Unittest

Unittest je zabudovaný framework inšpirovaný JUnit, ktorý je súčasťou štandardnej knižnice Pythonu. Využíva triednu štruktúru pre organizáciu testov a poskytuje základné metódy pre nastavenie a upratanie (`setUp` a `tearDown`). Je vhodný predovšetkým pre jednoduché projekty a štandardné testovanie. Framework obsahuje vlastný mechanizmus pre objavovanie testov, pričom sa vyznačuje stabilitou, avšak s obmedzenými možnosťami rozšírení. [10]

8.2 Pytest

Pytest predstavuje moderný framework s rozšírenými možnosťami [9]:

- Vyžaduje samostatnú inštaláciu
- Ponúka jednoduchý a intuitívny syntax
- Podporuje pokročilé funkcie ako parametrizované testy
- Obsahuje výkonný systém `fixtures` pre správu testovacieho prostredia
- Poskytuje detailné správy o zlyhaní testov
- Má rozsiahly ekosystém pluginov
- Podporuje paralelné spúšťanie testov

8.3 Hypothesis

Hypothesis je knižnica pre `property-based` testovanie [11]:

- Automatizuje generovanie testovacích dát na základe definovaných vlastností

- Používa systém dekorátorov pre špecifikáciu testov
- Poskytuje širokú škálu stratégií pre generovanie dát
- Umožňuje definovať vlastnosti kódu miesto konkrétnych testovacích prípadov
- Obsahuje dekorátor `@given` pre parametrizáciu argumentov
- Ponúka dekorátor `@example` pre špecifikáciu povinných testovacích prípadov
- Vhodná pre objavovanie hraničných prípadov a neočakávaného chovania

8.4 Porovnanie rozhraní

Aspekt	Pytest	Unittest	Hypothesis
Typ testovania	Jednotkové a funkčné testy	Jednotkové testy	Property-based testy
Inštalácia	Externý balík	Štandardná knižnica	Externý balík
Syntax	Jednoduchá a intuitívna	Založená na triedach, rozvláčna	Dekorátory a stratégie
Generovanie dát	Základná parametrizácia	Manuálne	Automatické generovanie
Rozšíriteľnosť	Pluginy a hooks	Obmedzená	Vlastné stratégie
Použitie	Všeobecné testovanie	Základné jednotkové testy	Testovanie vlastností a invariantov
Hlavná výhoda	Flexibilita a ekosystém	Štandardná súčasť Pythonu	Automatické hľadanie hraničných prípadov

Tabuľka 8.1: Porovnanie testovacích frameworkov v Pythone

Kapitola 9

Praktická časť

Praktická časť sa skladá z dvoch častí, ako bolo spomenuté na začiatku. Najskôr bolo potrebné spraviť demonštračnú aplikáciu pre jednoduché ukážky rôznych testov a ich jednoduché pochopenie.

Následne som sa zamerala na aplikáciu k-Dispatch. K-Dispatch je systém, ktorý zabezpečuje efektívne riadenie vzdialených výpočtov. Používa princíp Plug & Execute, čo znamená, že používateľ len zadá požiadavku a systém sa postará o všetko ostatné. K-Dispatch automaticky optimalizuje využitie HPC zdrojov a pomocou grafu závislostí úloh určuje najefektívnejší spôsob spracovania. [25] [24]

Nad systémom bolo potreba spraviť analýzu poskytnutých modulov a pripraviť vhodné typy testov a testovacích scenárov. Potom sa vybrané testy implementovali a aplikácia sa kontajnerizovala pre jednoduché lokálne spustenie. Teraz sa pozrieme na obe z týchto častí.

9.1 Demonštračná aplikácia a implementácia testov

Pre demonštráciu rôznych typov testovania som implementovala jednoduchý knižničný systém. Tento systém predstavuje vhodnú testovaciu aplikáciu, pretože obsahuje:

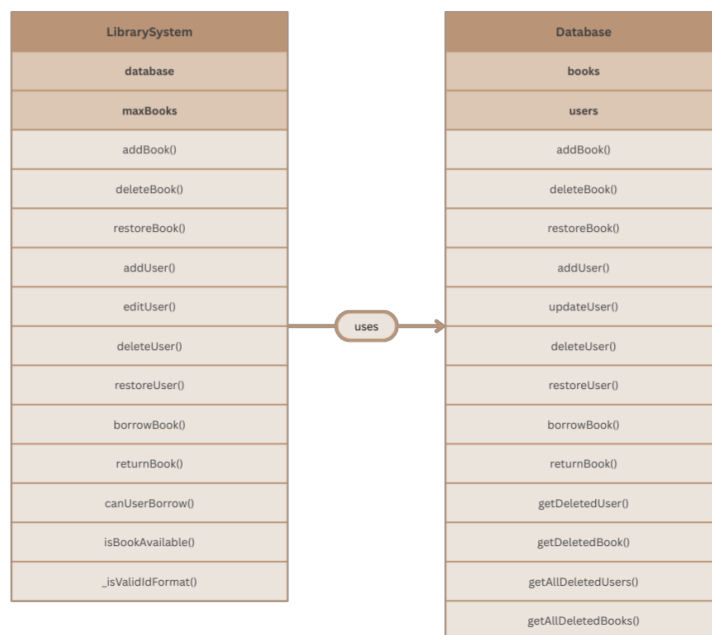
- Komplexné biznis pravidlá (napr. limity na požičiavanie kníh)
- Správu stavu (dostupnosť kníh, stav používateľov)
- Interakcie medzi rôznymi komponentmi
- Validáciu vstupov

9.1.1 Architektúra systému

Systém je implementovaný v jazyku Python a pozostáva z dvoch hlavných komponentov. Architektúra systému je znázornená na obrázku 9.1.

Hlavné komponenty systému sú:

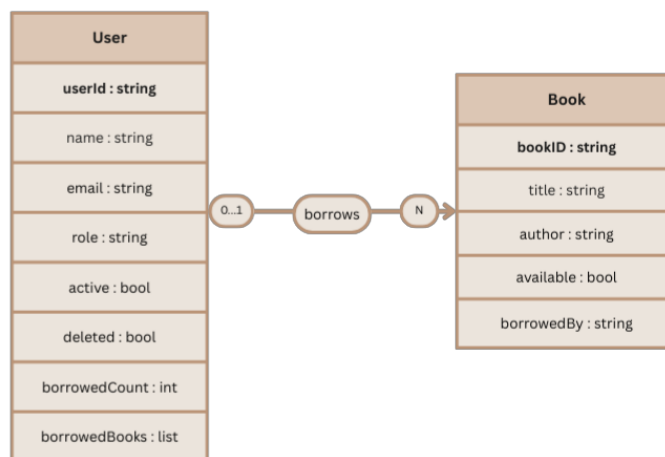
1. **LibrarySystem:** Hlavná trieda aplikácie, ktorá implementuje biznis logiku a orches-truje interakcie medzi komponentmi
2. **Database:** Trieda reprezentujúca databázu, ktorá spravuje stav kníh a používateľov



Obr. 9.1: Architektúra systému

9.1.2 Dátový model

Aplikácia pracuje s dvoma hlavnými entitami, ktorých vzťah je znázornený na obrázku 9.2.



Obr. 9.2: Dátový model systému

9.1.3 Implementácia testov

Jednotkové testy

Jednotkové testy sa zameriavajú na testovanie individuálnych komponentov systému v izolácii. Implementované sú pomocou frameworku `pytest`, časť testov je duplicitne implementovaná aj pomocou knižníc `unittest` a `hypothesis` pre demonštráciu rozdielov. Príklad implementácie jednotkového testu 9.1.1:

👁 Príklad jednotkového testu

```
1 def testAddBookAllRequiredFields(self, setupLibrary):
2     """Test adding book with all required fields"""
3     success, message = setupLibrary.addBook(
4         bookId="newBook",
5         title="Test Book",
6         author="Test Author"
7     )
8     assert success == True
9     assert "successfully" in message
```

Ukážka 9.1.1: Vytvorenie knihy

Integračné testy

Integračné testy overujú správnu interakciu medzi komponentmi systému. Príklad implementácie integračného testu 9.1.2:

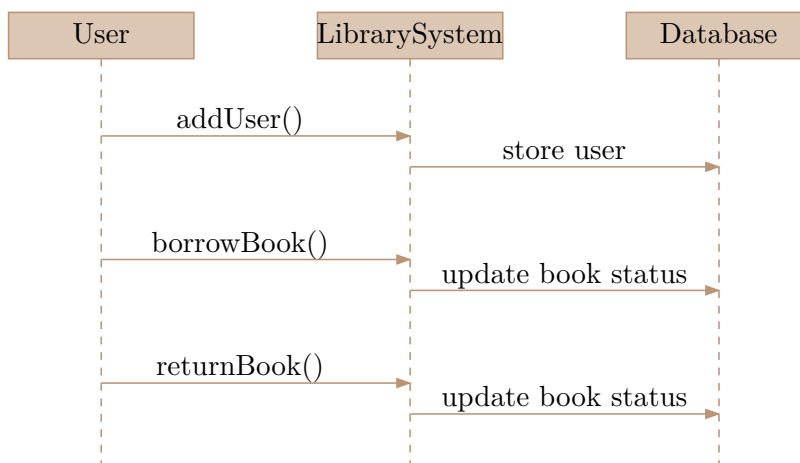
👁️ **Príklad integračného testu na proces pridania, výpožičky a vrátenia knihy.**

```
1 def testBorrowAndReturnBookFlow(self, setupLibrary):
2     """Test complete flow of borrowing and returning a book"""
3     library = setupLibrary
4     userSuccess, _ = library.addUser(
5         userId="testUser",
6         name="Test User",
7         email="test@example.com"
8     )
9     bookSuccess, _ = library.addBook(
10        bookId="testBook",
11        title="Test Book",
12        author="Test Author"
13    )
14    assert userSuccess and bookSuccess
15    borrowSuccess, _ = library.borrowBook("testUser", "testBook")
16    assert borrowSuccess == True
17    returnSuccess, _ = library.returnBook("testUser", "testBook")
18    assert returnSuccess == True
```

Ukážka 9.1.2: Integračný test

End-to-End testy

End-to-End (E2E) testy simulujú reálne používateľské scenáre 9.1.3. Príklad toku E2E testu je znázornený na obrázku 9.3.



Obr. 9.3: End-to-End (E2E) testy pre knižničný systém. Diagram ilustruje interakcie medzi troma hlavnými komponentami: Používateľ, Knižničný systém a Databáza

👁️ Príklad E2E testu cyklu života užívateľa

Implementácia komplexného E2E testu:

```
1 def testCompleteUserLifecycle(self, setupLibrary):
2     """E2E Test: Complete user lifecycle"""
3     library = setupLibrary
4
5     # 1. Create user
6     success, message = library.addUser(
7         'newUser',
8         'New User',
9         'new@test.com',
10        'student'
11    )
12    assert success == True
13
14    # 2. Edit user details
15    success, message = library.editUser(
16        'newUser',
17        name='Updated Name'
18    )
19    assert success == True
20
21    # 3. Borrow and return book
22    library.borrowBook('newUser', 'book1')
23    library.returnBook('newUser', 'book1')
24
25    # 4. Delete user
26    success, message = library.deleteUser('newUser')
27    assert success == True
```

Ukážka 9.1.3: E2E test

9.1.4 Testovacia stratégia

Testovacia stratégia je navrhnutá tak, aby pokryla všetky kritické aspekty systému:

Funkčné testovanie zahŕňa jednotkové testy zamerané na validáciu biznis logiky, integračné testy pre overenie správnosti interakcií medzi jednotlivými komponentmi, a E2E (end-to-end) testy, ktoré validovali kompletné používateľské scenáre.

V rámci nefunkčného testovania by sa kládol dôraz predovšetkým na výkonnostné testy, ktoré by hodnotili správanie systému z hľadiska odozvy, stability a škálovateľnosti pod rôznymi úrovňami záťaže. Taktiež by sa kládol dôraz na testovanie použiteľnosti užívateľského rozhrania. Tieto typy testov ale nebolo možné efektívne implementovať vzhľadom na veľkosť demonštračnej aplikácie.

9.1.5 Kontajnerizácia a monitorovanie aplikácie

Aplikácia je plne kontajnerizovaná pomocou technológie **Docker**. Implementované je aj monitorovanie stavu kontajnera. V prípade zlyhania kontroly stavu sa kontajner automaticky reštartuje.

Systém Správy Logov

Logy sú zbierané a preposielané pomocou softvéru **Fluent Bit**. Všetky logy sa ukladajú do súboru `app.log`. Je spustená taktiež **cron** úloha ktorá každú minútu analyzuje obsah logov. Aktuálne sa vyhľadávajú a spracúvajú iba logy obsahujúce tieto výrazy: **ERROR**, **warning**, **fail**.

Notifikačný Systém

Pri nájdení kritických logov systém automaticky posiela upozornenia e-mailom pomocou nástroja **msmt**. Každý odoslaný log je zahashovaný a uložený do evidencie. Mechanizmus zabráňuje duplicitnému odosielaniu upozornení pri opätovnom spustení kontajnera.

Testovanie a Automatizácia

Po spustení **Docker** kontajnera sa taktiež automaticky spúšťajú všetky testy. Celý vývojový proces je automatizovaný pomocou aj pomocou **CI/CD pipeline**. Testy sú automaticky spúšťané po každej aktualizácii kódu v git vetve.

9.2 K-Dispatch aplikácia

V rámci predkladanej bakalárskej práce sa zameriame na analýzu aplikácie k-Dispatch so zreteľom na prvky zabezpečujúce kľúčovú funkcionálnosť systému. Centrálné komponenty aplikácie sú navrhnuté primárne na dekódovanie vstupných parametrov, plánovanie a následnú realizáciu spracovateľských procesov, monitorovanie efektívnosti využívania distribuovaných výpočtových zdrojov a účtovanie nákladov spojených s využitím výpočtového času na vysokovýkonných výpočtových klastroch (HPC). V nasledujúcich častiach detailne charakterizujeme jednotlivé programové moduly zodpovedné za implementáciu uvedených funkčných oblastí. [24] [25]

Predmetom skúmania budú ďalej databázové štruktúry, ktoré podrobíme dôkladnej analýze prostredníctvom špecifických testovacích postupov. Pozornosť budeme venovať aj poskytovanému webovému rozhraniu.

9.2.1 Modul Dispatch & Transfer

Trieda `DispatchAndTransfer` implementuje komplexný systém na spracovanie a riadenie výpočtových úloh v prostredí systému k-Dispatch. Hlavné funkcionality zahŕňajú:

- **Dekódovanie vstupu:** Parsovanie vstupných súborov, vytváranie grafov závislostí úloh a generovanie skriptov pre výpočtové klastre.
- **Manažment úloh:** Monitorovanie stavu úloh, reštartovanie zlyhaných úloh a správa závislostí medzi úlohami v rámci pracovného toku.

- **Správa zdrojov:** Sledovanie spotreby výpočtových hodín pre účtovanie, pripájanie k vzdialeným HPC klastrom a manažment alokácie výpočtových zdrojov.
- **Manipulácia so súbormi:** Prenos súborov na/zo vzdialených strojov, sťahovanie výsledných súborov a čistenie dočasných súborov.
- **Spracovanie chýb:** Rozsiahle bloky `try/except` na zachytenie rôznych typov zlyhaní a detailné logovanie chýb a operácií.

Navrhované testovacie scenáre pre `DispatchAndTransfer`

1. Spracovanie poškodeného HDF5 súboru - (1) Poskytnúť poškodený vstupný súbor, (2) Overiť, že systém správne deteguje poškodenie, (3) Skontrolovať, či sa stav nastaví na ERROR, (4) Zabezpečiť, aby boli zaznamenané príslušné chybové hlásenia
2. Zlyhania SSH pripojenia
 - Zlyhanie pripojenia počas inicializácie - (1) Simulovať zlyhanie SSH pripojenia pri prvom pripojení k vzdialenému stroju, (2) Overiť, že chyba je zachytená a správne zaznamenaná
 - Strata pripojenia počas prenosu súborov - (1) Simulovať prerušenie SSH pripojenia počas nahrávania/sťahovania súboru, (2) Skontrolovať zotavenie alebo správne ošetrenie chyby
 - Časový limit pripojenia počas odosielania úlohy - (1) Simulovať vypršanie časového limitu pri odosielaní úloh do fronty, (2) Overiť, že systém správne zaznamená zlyhanie
3. Problémy so vzdialeným strojom - (1) Testovať správanie, keď je vzdialený diskový priestor plný, (2) Testovať, keď vzdialený stroj odmieta odoslanie úloh (prekročená kvóta), (3) Testovať, keď je vzdialený stroj dočasne nedostupný
4. Zlyhania databázy - (1) Testovať ošetrenie výpadkov pripojenia k databáze, (2) Testovať chyby transakcií počas aktualizácie stavu úloh, (3) Testovať konflikty pri uzamykaní záznamov
5. Zlyhania pri alokácii zdrojov - (1) Testovať situáciu, keď nie je k dispozícii žiadna vhodná alokácia, (2) Testovať, keď sa hodiny vyčerpajú uprostred výpočtu
6. Chyby pri správe úloh - (1) Testovať správanie, keď je prekročený maximálny počet pokusov o reštart, (2) Testovať ošetrenie úloh, ktoré zlyhajú ihneď po odoslaní, (3) Testovať ošetrenie "zombie" úloh (zobrazujú sa ako spustené, ale nepostupujú)
7. Problémy so súborovým systémom - (1) Testovať, keď nie je možné stiahnuť výsledný súbor, (2) Testovať, keď je lokálne úložisko nedostatočné, (3) Testovať správne vyčistenie po zlyhaní
8. Scenáre zrušenia - (1) Testovať zrušenie úloh, ktoré sú vo fronte, (2) Testovať zrušenie spustených úloh, (3) Testovať zrušenie úloh, ktoré už na vzdialenom systéme neexistujú
9. Podmienky súbehu - (1) Testovať viacero aktualizácií prebiehajúcich súčasne, (2) Testovať požiadavku na zrušenie počas aktualizácie stavu

10. Testy zotavenia - (1) Testovať zotavenie systému po reštarte s úlohami v rôznych stavoch, (2) Testovať správanie pri prevzatí liečebných plánov po zlyhaní systému

9.2.2 Modul Remote Machine

Trieda `RemoteMachine` predstavuje abstraktnú triedu, ktorá zabezpečuje komunikáciu so vzdialenými výpočtovými klastrami. Hlavnými funkciami tejto triedy sú:

- **Pripojenie k vzdialenému stroju** - spravuje SSH pripojenia prostredníctvom knižnice Fabric a kľúčov
- **Prenos súborov** - nahráva a sťahuje súbory potrebné pre výpočty
- **Správa úloh** - odosielanie, sledovanie stavu a rušenie výpočtových úloh v plánovačoch dávok (batch schedulers)
- **Správa vzdialených priestorov** - vytváranie adresárovej štruktúry a čistenie po výpočtoch

Základná funkcionálna je definovaná ako množina abstraktných metód, ktoré musia byť implementované v konkrétnych podtriedach pre jednotlivé typy výpočtových systémov. Trieda tiež poskytuje rozsiahly systém spracovania chýb prostredníctvom vlastných výnimiek a stavových kódov.

Dôležitou súčasťou funkcionality je aj správa šablón pre skripty dávkových úloh a parametrizácia príkazov pre rôzne plánovače úloh.

Navrhované testovacie scenáre pre modul

Napriek tomu, že `RemoteMachine` je abstraktná trieda, jej testovanie je esenciálne pre zabezpečenie správneho fungovania konkrétnych implementácií. Nasledujúce testovacie scenáre pokrývajú hlavnú funkcionálnu.

- Testovanie Factory metódy
 1. Test správneho vytvorenia konkrétnej implementácie - (1) Zavolať metódu `factory()` s existujúcim názvom stroja, (2) Overenie, že vrátená inštancia je správneho typu (konkrétne podtrieda triedy `RemoteMachine`).
 2. Test reakcie na neexistujúci názov stroja - (1) Zavolať metódu `factory()` s názvom stroja, ktorý neexistuje, (2) Overiť, či je korektne vyvolaná výnimka `RemoteMachineException` s príslušnou chybovou správou.
- Testovanie pripojenia k vzdialenému stroju
 1. Test úspešného pripojenia - (1) Inicializovať implementáciu s platnými údajmi, (2) Zavolať metódu `tryConnection()`, (3) Overiť, že nebola vyvolaná žiadna výnimka.
 2. Test zlyhania pripojenia z dôvodu neexistujúceho stroja - (1) Inicializovať s neplatným názvom hostiteľa, (2) Zavolať metódu `tryConnection()`, (3) Skontrolovať vyvolanie výnimky `RemoteMachineException` s chybovým kódom `Errors.MACHINEOFFLINE_ERR`.

3. Test zlyhania pripojenia kvôli chybnému SSH kľúču - (1) Inicializovať s neplatným SSH kľúčom, (2) Zavolať metódu `tryConnection()`, (3) Overiť vyvolanie výnimky s chybovým kódom `Errors.SSHKEY_ERR`.
- Testovanie prenosu súborov
 1. Test úspešného nahratia súboru - (1) Vytvoriť testovací súbor, (2) Zavolať metódu `uploadFile()`, (3) Overiť, že daný súbor skutočne existuje na vzdialenom stroji.
 2. Test úspešného stiahnutia súboru - (1) Vytvoriť súbor na vzdialenom stroji, (2) Zavolať metódu `downloadFile()` (3) Overiť, či bol súbor úspešne a kompletne stiahnutý lokálne.
 3. Test nahratia súboru pri nedostatku miesta - (1) Simulovať zaplnený disk na vzdialenom stroji, (2) Zavolať metódu `uploadFile()`, (3) Overiť, či systém správne signalizuje túto chybu.
 4. Test prerušenia prenosu súboru - (1) Simulovať prerušenie sieťového spojenia počas prebiehajúceho prenosu súboru, (2) Verifikovať, či systém adekvátne ošetrí túto situáciu a vyvolá očakávanú výnimku `fabric.exceptions.GroupException`.
 - Testovanie správy úloh
 1. Test úspešného odoslania úlohy - (1) Pripraviť testovací skript pre úlohu, (2) Zavolať metódu `submitJob()`, (3) Overiť, či systém vrátil platný identifikátor pre novovytvorenú úlohu.
 2. Test odoslania úlohy so závislosťou - (1) pripraviť dva testovacie skripty, (2) odoslať prvú úlohu pomocou `submitJob()`, (3) odoslať druhú úlohu s definovanou závislosťou na prvú pomocou `submitJob(jobScript, [jobId1])`, (4) overiť správne nastavenie závislosti.
 3. Test získania stavu úlohy - (1) Pripraviť a odoslať testovaciu úlohu, (2) zavolať metódu `getJobStatus()`, (3) verifikovať, či vrátený stav zodpovedá očakávanému stavu úlohy.
 4. Test zrušenia úlohy - (1) Pripraviť a odoslať testovaciu úlohu, (2) zavolať metódu `cancelJob()`, (3) pomocou `getJobStatus()` overiť, že úloha bola úspešne zrušená.
 5. Test reštartovania úlohy - (1) Pripraviť a odoslať testovaciu úlohu do plánovača, (2) zavolať metódu `restartJob()`, (3) skontrolovať, či bol stav úlohy správne aktualizovaný.
 - Testovanie správy adresárov
 1. Test vytvorenia adresárovej štruktúry - (1) Zavolať metódu `createFoldersOnRemoteMachine()`, (2) skontrolovať, vytvorenie všetkých adresárov.
 2. Test čistenia vzdialeného priestoru - (1) Vytvoriť testovacie súbory na vzdialenom stroji, (2) zavolať metódu `cleanRemoteSpace()`, (3) overiť, či boli všetky dočasné súbory odstránené.
 3. Test vytvorenia adresárov pri nedostatku oprávnení - (1) Simulovať situáciu s nedostatočnými oprávneniami, (2) zavolať metódu `createFoldersOnRemoteMachine()`, (3) overiť korektné vyvolanie a spracovanie príslušnej výnimky.

- Testovanie ošetrovania chýb
 1. Test ošetrovania výnimiek pri prenose súborov - (1) Simulovať prerušenie sieťového spojenia počas prenosu dát, (2) zavolať metódu `uploadFile()` alebo `downloadFile()`, (3) overiť, či sa správne odchytila vzniknutá chyba.
 2. Test ošetrovania výnimiek pri odosielaní úloh - (1) Simulovať chybu na strane plánovača úloh, napríklad jeho dočasnú nedostupnosť, (2) zavolať metódu `submitJob()`, (3) kontrolovať, či systém túto chybovú situáciu korektne identifikoval a správne spracoval.
 3. Test ošetrovania výnimiek pri zisťovaní stavu úlohy - (1) Simulovať chybu počas komunikácie s plánovačom, (2) zavolať metódu `getJobStatus()`, (3) overovať, či je v prípade takejto chyby korektne vrátená hodnota `None`.
- Testovanie konfigurácie
 1. Test spracovania ciest šablón - (1) Inicializovať implementáciu s rôznymi hodnotami vstupných parametrov, (2) kontrolovať správnosť ciest definovaných prostredníctvom vlastností ako napríklad `remoteScratchPath`, `jobScriptsPath`, atď.
 2. Test nastavenia príkazov plánovača - (1) Overiť, že príkazy ako `jobSubmitCmd`, `jobStatCmd`, atď. obsahujú očakávané hodnoty nastavení.
 3. Test správneho spracovania šablóny hlavičky úlohy - (1) Overiť, že vlastnosť `jobScriptHeader` obsahuje správne hodnoty pre daný plánovač.
- Testovanie limitov úloh
 1. Test získania limitov počtu úloh - (1) Zavolať `getSubmittedJobsLimits()`, (2) skontrolovať, či vrátené hodnoty zodpovedajú očakávaným limitom.
 2. Test prekročenia maximálneho počtu úloh - (1) Simulovať stav, kedy je už dosiahnutý maximálny povolený počet úloh, (2) skúsiť odoslať ďalšiu novú úlohu, (3) overovať, či systém túto situáciu korektne spracuje.

Keďže trieda `RemoteMachine` obsahuje mnoho abstraktných metód, je potrebné implementovať testovateľnú podtriedu, ktorá bude použitá v testoch. V tejto podtriede možno použiť mocky pre simuláciu interakcie so vzdialeným prostredím, aby testy nevyžadovali skutočné pripojenie k výpočtovému klastru.

9.2.3 Modul `MediumParameters`

Trieda `MediumParameters` predstavuje abstraktnú triedu, ktorá slúži na parsovanie a správu parametrov média z rôznych plánovacích súborov vo formáte HDF5. Hlavnými funkciami tejto triedy sú:

- Parsovanie vstupných súborov - Trieda dokáže načítať a analyzovať dáta z rôznych typov HDF5 súborov, ktoré obsahujú parametre pre simulácie
- Rozpoznávanie typu pracovného toku - Identifikuje typ simulácie alebo workflow na základe metadát v súbore
- Správa vlastností média - Udržiava informácie o vlastnostiach simulovaného média, ako sú absorpcia, linearita, homogenosť atď.

- Určenie parametrov simulácie - Poskytuje informácie o rozmeroch simulačnej mriežky, počte časových krokov a ďalších parametroch potrebných pre simuláciu

Základná funkcionálna je definovaná ako množina abstraktných metód, ktoré musia byť implementované v konkrétnych podtriedach pre jednotlivé typy plánovacích súborov. Trieda tiež poskytuje rozsiahly systém spracovania chýb prostredníctvom vlastných výnimiek a stavových kódov.

Testovacie scenáre pre triedu `MediumParameters`

Vzhľadom na to, že `MediumParameters` je abstraktná trieda, jej testovanie musí zahŕňať konkrétne implementácie. Nasledujúce testovacie scenáre pokrývajú kľúčovú funkcionálnu tejto triedy.

- Testovanie Factory metódy
 1. Test správneho vytvorenia konkrétnej implementácie pre HDF5 súbor - (1) Pripraviť testovací HDF5 súbor s atribútom `workflow_type` nastaveným na známy typ, (2) zavolať metódu `factory()` s cestou k tomuto súboru, (3) overiť, že vrátená inštancia je správneho typu, (4) overiť, že vlastnosť `workflowName` obsahuje očakávanú hodnotu.
 2. Test neexistujúceho súboru - (1) Zavolať metódu `factory()` s cestou k neexistujúcemu súboru, (2) overiť, že je vyvolaná výnimka `MediumParameterException` s príslušnou správou.
 3. Test nepodporovanej prípony súboru - (1) Zavolať metódu `factory()` s cestou k súboru s nepodporovanou príponou, (2) overiť, že je vyvolaná výnimka `MediumParameterException` s kódom `Errors.EXT_ERR`.
 4. Test poškodeného HDF5 súboru - (1) Pripraviť poškodený HDF5 súbor, (2) zavolať metódu `factory()` s cestou k tomuto súboru, (3) overiť, že je vyvolaná výnimka `MediumParameterException` s príslušnou správou.
 5. Test neznámeho atribútu súboru `workflow_type` - (1) Pripraviť HDF5 súbor s atribútom `workflow_type` nastaveným na neznámy typ, (2) zavolať metódu `factory()` s cestou k tomuto HDF5 súboru, (3) overiť, že bola vyvolaná výnimka `MediumParameterException` s kódom `Errors.WORKFLOW_ERR`.
- Testovanie abstraktnej metódy `__call__()`
 1. Test implementácie v podtriede - (1) Vytvoriť testovaciu podtriedu implementujúcu metódu `call()`, (2) vytvoriť inštanciu pomocou `factory()`, (3) zavolať metódu `call()` na tejto inštancii, (4) overiť, že metóda bola úspešne volaná a spracovala vstupný súbor.
 2. Test chýbajúcej implementácie - (1) Vytvoriť testovaciu podtriedu bez implementácie `call()`, (2) pokúsiť sa vytvoriť a zavolať inštanciu, (3) overiť, že je vyvolaná výnimka `NotImplementedError`.
- Testovanie spracovania chýb
 1. Test správy o chybe - (1) Zavolať setter `errorMessage` s testovacou správou, (2) overiť, že getter `errorMessage` vráti správne nastavenú správu.

2. Test vytvorenia vlastnej výnimky - (1) Vytvoriť `MediumParameterException` s testovacou správou, (2) overiť, že vlastnosť `message` obsahuje správne hodnoty, (3) overiť, že `str()` vráti očakávaný reťazec.
 3. Test zachytenia a spracovania výnimky pri parsovaní - (1) Vytvoriť testovaciu podtriedu, ktorá vyvoláva výnimky pri parsovaní, (2) pokúsiť sa spracovať súbor pomocou tejto podtriedy, (3) overiť správne zachytenie a spracovanie výnimky.
- Testovanie vlastností média
 1. Test nastavenia a získania vlastností - (1) Vytvoriť inštanciu konkrétnej podtriedy pomocou `factory()`, (2) nastaviť rôzne hodnoty pre vlastnosti pomocou príslušných setterov (`isAbsorptive`, `isLinear`, `isHomogeneous`, `isAxisymmetric`, `isElastic`), (3) overiť, že gettery vracajú správne hodnoty, (4) overiť, že vlastnosť `flags` vracia správnu kombináciu hodnôt.
 2. Test nastavenia a získania parametrov simulácie - (1) Vytvoriť inštanciu konkrétnej podtriedy pomocou `factory()`, (2) nastaviť rôzne hodnoty pre parametre simulácie (`sonications`, `dimensions`, `timeSteps`, `memoryEstimation`, `outputFileSize`, `userProgramParameters`), (3) overiť, že gettery vracajú správne hodnoty.
 - Testovanie typu pracovného toku (workflow)
 1. Test nastavenia a získania typu workflow - (1) Vytvoriť inštanciu konkrétnej podtriedy pomocou `factory()`, (2) nastaviť hodnotu `workflowEnumObject`, (3) overiť, že vlastnosť `workflowName` bola správne nastavená (lowercase názov enum), (4) nastaviť inú hodnotu vlastnosti `workflowName`, (5) overiť, že hodnota bola správne zmenená.
 2. Test mapovania typov workflow - (1) Pre každý typ v `_workflowTypes` overiť, že existuje zodpovedajúci záznam v `_workflowTypesCodeRepr`, (2) vytvoriť inštanciu `InputType` pre každý typ workflow, (3) overiť, že metódy `getCodeRepr()`, `getFileRepr()` a `getEnumName()` vracajú očakávané hodnoty.
 - Testovanie enumerácií
 1. Test enumov `FileExtension` - (1) Overiť, že `FileExtension.getValuesList()` vracia zoznam všetkých hodnôt, (2) overiť, že metóda rozpoznáva všetky podporované prípony súborov.
 2. Test enumov `Errors` - (1) Overiť, že každá hodnota v `Errors` obsahuje zmysluplnú chybovú správu, (2) otestovať použitie `Errors` pri vytváraní výnimiek.
 3. Test enumov `InputType` - (1) Overiť, že pre každý prvok v `_workflowTypes` existuje zodpovedajúci enum, (2) otestovať konverziu medzi “file representation” a “code representation”, (3) skontrolovať, že `InputType.getCodeRepr()` vracia očakávané hodnoty.
 - Testovanie s reálnymi dátami
 1. Test integrácie s reálnymi HDF5 súborami - (1) Použiť reálne vzorky HDF5 súborov pre jednotlivé typy workflow, (2) zavolať `factory()` pre každý súbor, (3) overiť, že súbory sú správne klasifikované a parsované, (4) skontrolovať, že hodnoty médií a simulačné parametre sú správne extrahované.

2. Test s veľkými reálnymi dátami - (1) Použiť veľké HDF5 súbory (ak sú dostupné), (2) otestovať výkon a efektivitu parseru, (3) overiť správnosť extrahovaných parametrov.

Pre testovanie `MediumParameters` bude potrebné vytvoriť testovaciu implementáciu (podtriedu), keďže táto trieda je abstraktná. Táto testovacia podtrieda by mala implementovať minimálne metódu `__call__()`, aby bolo možné inštanciu triedy používať v testoch. Prípadne je možné využiť existujúce konkrétne implementácie, ktoré už v systéme existujú.

9.2.4 Modul Dependency Graph

Trieda `DependencyGraph` predstavuje abstraktnú triedu, ktorá implementuje graf závislostí úloh pre plánovanie a riadenie výpočtových úloh. Hlavnými funkciami tejto triedy sú:

- **Tvorba grafu závislostí** - Trieda vytvára a spravuje graf, ktorý reprezentuje závislosti medzi výpočtovými úlohami, kde úlohy na rovnakej úrovni môžu byť vykonávané súbežne, zatiaľ čo úlohy medzi úrovňami majú vzájomné závislosti
- **Optimalizácia výpočtových zdrojov** - Trieda vyhodnocuje rôzne možné konfigurácie spustenia pre každú úlohu v grafe a vyberá najvhodnejšiu konfiguráciu na základe kritérií ako čas výpočtu a náklady
- **Generovanie úlohových skriptov** - Trieda generuje skripty špecifické pre výpočtový klaster, ktoré sú potrebné na spustenie jednotlivých výpočtových úloh
- **Správa databázových záznamov** - Trieda vytvára a aktualizuje záznamy v databáze pre výpočtové úlohy a ich závislosti

Základná funkcionálna je definovaná ako množina abstraktných metód, ktoré musia byť implementované v konkrétnych podtriedach. Trieda tiež poskytuje rozsiahle možnosti pre hodnotenie a porovnávanie rôznych konfigurácií grafu závislostí, čo umožňuje optimalizáciu využitia zdrojov v heterogénnom výpočtovom prostredí.

Testovacie scenáre pre triedu DependencyGraph

Vzhľadom na to, že `DependencyGraph` je abstraktná trieda, jej testovanie musí zahŕňať konkrétne implementácie. Nasledujúce testovacie scenáre pokrývajú kľúčovú funkcionálnu triedy.

- Testovanie Factory metódy
 1. Test správneho vytvorenia konkrétnej implementácie - (1) Zavolať `factory()` s názvom existujúcej podtriedy, (2) overiť, že vrátená inštancia je správneho typu (podtrieda `DependencyGraph`), (3) overiť, že vrátená inštancia má inicializované základné atribúty (`_graph`, `_score`, atď.).
 2. Test neexistujúcej podtriedy - (1) Zavolať metódu `factory()` s neexistujúcim názvom podtriedy, (2) overiť, že je vyvolaná výnimka `ValueError` s príslušnou správou.
- Testovanie metódy `__call__()`

1. Test úspešného vytvorenia grafu - (1) Vytvoriť inštanciu konkrétnej podtriedy pomocou `factory()`, (2) pripraviť testovacie alokácie, záznam liečebného plánu a parametre média, (3) zavolať metódu `call()` s týmito parametrami, (4) overiť, že bol vytvorený graf a nastavené `bestGraph`, `bestScore` a `_bestHpcAllocation`.
 2. Test bez dostupných binárnych súborov - (1) Vytvoriť inštanciu s nastavením, ktoré spôsobí, že nebudú vybrané žiadne binárne súbory pre úlohy, (2) zavolať metódu `call()`, (3) overiť, že je vyvolaná výnimka `ValueError` s príslušnou správou o chýbajúcich binárnych súboroch, (4) overiť, že stav liečebného plánu je nastavený na `JobStatus.ERROR`.
 3. Test preťaženého výpočtového klastra - (1) Simulovať situáciu, keď všetky dostupné klastre dosiahli maximálny počet úloh, (2) zavolať metódu `call()`, (3) overiť, že je vyvolaná výnimka `ValueError` s príslušnou správou o preťažení, (4) overiť, že stav liečebného plánu je nastavený na `JobStatus.NEW`.
- Testovanie manipulácie s grafom
 1. Test metódy `getLevel()` - (1) Vytvoriť inštanciu konkrétnej podtriedy s inicializovaným grafom, (2) zavolať metódu `getLevel()` pre rôzne úrovne grafu, (3) overiť, že metóda vracia správne zoznamy úloh pre jednotlivé úrovne, (4) otestovať aj správanie pri neexistujúcej úrovni.
 2. Test metódy `itemsInLevel()` - (1) Vytvoriť inštanciu s inicializovaným grafom, (2) zavolať metódu `itemsInLevel()` pre rôzne úrovne, (3) overiť, že metóda vracia správny počet úloh v každej úrovni, (4) otestovať aj správanie pri neexistujúcej úrovni.
 3. Test metódy `__len__()` - (1) Vytvoriť inštanciu s inicializovaným grafom, (2) zavolať funkciu `len()` s touto inštanciou, (3) overiť, že je vrátený správny počet úrovní grafu.
 4. Test metódy `_addToGraph()` - (1) Vytvoriť inštanciu s inicializovaným prázdny grafom, (2) zavolať metódu `_addToGraph()` s rôznymi prvkami a úrovňami, (3) overiť, že prvky sú správne pridané do grafu, (4) otestovať pridávanie prvkov do existujúcej aj novej úrovne, (5) otestovať pridávanie jednotlivých prvkov aj zoznamov.
 - Testovanie hodnotenia a výberu konfigurácií
 1. Test metódy `_chooseExecConfigurationAndEvaluate()` - (1) Vytvoriť inštanciu s inicializovaným grafom obsahujúcim niekoľko úloh, (2) zavolať metódu `_chooseExecConfigurationAndEvaluate()` s rôznymi alokáciami a HPC, (3) overiť, že metóda `selectExecConfiguration()` je volaná pre každú úlohu v grafe, (4) overiť, že skóre grafu je správne aktualizované.
 2. Test metódy `_compareAndUpdate()` - (1) Vytvoriť inštanciu s inicializovaným grafom a skóre, (2) inicializovať aj hodnoty `bestGraph` a `bestScore`, (3) zavolať metódu `_compareAndUpdate()` s rôznymi parametrami `alpha`, (4) overiť, že `bestGraph` a `bestScore` sú aktualizované, iba ak nový graf má lepšie skóre.
 3. Test výpočtu skóre grafu - (1) Vytvoriť inštanciu s grafom obsahujúcim úlohy s rôznymi atribútmi (čas, zdroje), (2) zavolať metódy `_countReservedCoreHours()` a `_countWallClockTime()`, (3) overiť, že vypočítané hodnoty zodpovedajú očakávaným súčtom z jednotlivých úloh.

- Testovanie generovania skriptov úloh
 1. Test metódy `generateJobScripts()` - (1) Vytvoriť inštanciu konkrétnej podtriedy s inicializovaným grafom, (2) zavolať metódu `generateJobScripts()` s príslušnými parametrami, (3) overiť, že pre každú úlohu v grafe sú vygenerované vhodné skripty, (4) skontrolovať obsah skriptov, či obsahujú očakávané príkazy a nastavenia.
 2. Test metódy `_renderTemplate()` - (1) Vytvoriť inštanciu s definovanými šablónami, (2) zavolať metódu `_renderTemplate()` s rôznymi šablónami a dátami, (3) overiť, že výsledný text obsahuje správne nahradené hodnoty zo šablóny, (4) otestovať aj správanie pri neplatných dátach alebo šablónach.
- Testovanie ukladania do databázy
 1. Test metódy `saveToDatabase()` - (1) Vytvoriť inštanciu konkrétnej podtriedy s inicializovaným najlepším grafom, (2) zavolať metódu `saveToDatabase()` s príslušnými parametrami, (3) overiť, že pre každú úlohu v grafe sú vytvorené záznamy v databáze, (4) skontrolovať, že závislosti medzi úlohami sú správne zaznamenané.
 2. Test metódy `_createDependency()` - (1) Vytvoriť inštanciu s inicializovanými úlohami, (2) zavolať metódu `_createDependency()` s rôznymi kombináciami úloh, (3) overiť, že záznamy závislostí sú správne vytvorené v databáze, (4) otestovať vytvorenie závislosti pre jedného predchodcu aj pre zoznam predchodcov.
 3. Test metódy `_createFileRecord()` - (1) Vytvoriť inštanciu a identifikátor úlohy v databáze, (2) zavolať metódu `_createFileRecord()` s rôznymi typmi súborov, (3) overiť, že záznamy súborov sú správne vytvorené v databáze, (4) otestovať vytvorenie záznamov pre jeden súbor aj pre zoznam súborov.
- Testovanie správy zdrojov
 1. Test metódy `_cleanTemporaryGraph()` - (1) Vytvoriť inštanciu s inicializovaným dočasným grafom, (2) zavolať metódu `_cleanTemporaryGraph()`, (3) overiť, že `_graph` a `_score` sú nastavené na `None`, (4) overiť, že objekty z grafu sú správne vymazané z pamäte.
- Testovanie pomocných metód
 1. Test metódy `_getActTimestampString()` - (1) Zavolať metódu, (2) overiť, že vrátený reťazec má správny formát časovej stopy.
 2. Test metódy `_flattenNestedList()` - (1) Zavolať metódu s rôznymi vnoreniami zoznamov, (2) overiť, že vrátený zoznam je správne "sploštený" a obsahuje všetky prvky.

Vzhľadom na komplexnosť triedy `DependencyGraph` a jej interakcie s inými komponentmi, je vhodné použiť kombináciu jednotkových testov pre jednoduché metódy a integračných testov pre overenie správnej interakcie s ostatnými triedami v systéme.

9.3 Návrh testovacích scenárov

Pri návrhu testov pre databázové tabuľky je potrebné postupovať systematicky, aby sme pokryli všetky aspekty správnej funkcionality. Nasledujúce sekcie popisujú metodológiu a konkrétne testovacie scenáre, ktoré je vhodné implementovať pre komplexné otestovanie databázových tabuliek.

Na demonštráciu testovacích metód so zvolila tabuľku `users` na ktorej budem ukazovať príklady testovania.

9.3.1 Testovanie povinných polí

Pri testovaní povinných polí je kľúčové overiť, že databáza správne odmieta záznamy, v ktorých chýbajú povinné atribúty [9.3.1](#). V prípade tabuľky `User` sú povinné nasledujúce polia: `group`, `state`, `name`, `surname`, `login` a `email`. Navrhované testovacie scenáre:

- Pre každé povinné pole vytvoríme test, v ktorom dané pole vynecháme, zatiaľ čo ostatné polia vyplníme platnými hodnotami
- Overíme, že databáza vráti chybu typu `IntegrityError` alebo podobnú
- Pre textové polia otestujeme aj zadanie prázdnych reťazcov, čo by tiež malo byť zamietnuté, ak pole nepovoľuje prázdne hodnoty

🔗 Príklad testu na overenie povinného poľa

```
1 def test_povinne_pole_name(self):
2     # Create a user without filling in the required "name" field
3     user = User()
4     user.group = self.test_group
5     user.state = self.test_state
6     # name is intentionally omitted
7     user.surname = "Testing"
8     user.login = "testlogin"
9     user.email = "test@example.com"
10    # We expect an error when attempting to save
11    with self.assertRaises(IntegrityError):
12        user.save()
```

Ukážka 9.3.1: Test povinných atribútov

9.3.2 Testovanie unikátnych obmedzení

Unikátne obmedzenia zabezpečujú, že určité hodnoty alebo ich kombinácie sa v databáze nevyskytujú duplicitne. V prípade tabuľky `User` je pole `login` definované ako unikátne [9.3.2](#).

🕒 Príklad testu na overenie unikátnosti loginu

Navrhované testovacie scenáre:

- Vytvoríme platný záznam používateľa s konkrétnym loginom
- Pokúsime sa vytvoriť druhého používateľa s rovnakým loginom, ale odlišnými hodnotami v ostatných poliach
- Overíme, že databáza vráti chybu unikátnosti
- Ak je unikátnosť definovaná case-insensitive, otestujeme aj vytvorenie používateľa s rovnakým loginom, ale v inej kapitalizácii

```
1 def testUniqueLogin(self):
2     # Create the first user
3     user1 = User()
4     user1.group = self.test_group
5     user1.state = self.test_state
6     user1.name = "First"
7     user1.surname = "User"
8     user1.login = "samelogin"
9     user1.email = "first@example.com"
10    user1.save()
11    # Attempt to create a second user with the same login
12    user2 = User()
13    user2.group = self.test_group
14    user2.state = self.test_state
15    user2.name = "Second"
16    user2.surname = "User"
17    user2.login = "samelogin" # Same login as the first user
18    user2.email = "second@example.com"
19
20    # We expect an error when attempting to save
21    with self.assertRaises(IntegrityError):
22        user2.save()
```

Ukážka 9.3.2: Test unikátnych atribútov

9.3.3 Testovanie referenčnej integrity

Referenčná integrita zabezpečuje, že hodnoty v cudzích kľúčoch odkazujú na existujúce záznamy v referenčných tabuľkách. V tabuľke `User` sú `group` a `state` cudzie kľúče [9.3.3](#).

🕒 Príklad testu na overenie referenčnej integrity

Navrhované testovacie scenáre:

- Pokúsime sa vytvoriť používateľa s neexistujúcim ID skupiny
- Pokúsime sa vytvoriť používateľa s neexistujúcim ID stavu
- Overíme, že databáza vráti chybu referenčnej integrity

```
1 def testNonExistentFK(self):
2     # Create a user with a non-existent group ID
3     user = createUser(
4         user.name = "Test"
5         ...
6     )
7     # Assign a non-existent group ID
8     with self.assertRaises((IntegrityError, ForeignKeyConstraintError)):
9         user.group_id = 9999 # Assume that ID 9999 does not exist
10        user.save()
```

Ukážka 9.3.3: Test referenčnej integrity

9.3.4 Testovanie dátových typov

Správna validácia dátových typov je kľúčová pre integritu databázy. Každý atribút v tabuľke `User` má definovaný dátový typ, ktorý by mal byť dodržaný 9.3.4. Pre naše špecifické využitie, nie sú testy dátových typov nutné, vzhľadom na to že o automatickú zmenu typu sa stará knižnica `peewee` v ktorej je naša databáza implementovaná.

🔗 Příklad komplexního testu datových typů

Navrhované testovací scénáře:

- Systematicky testujeme každé pole s různými neplatnými datovými typy
- Pre textové polia (`name`, `surname`, `login`, `email`) skúšame vložiť čísla, zoznamy, slovníky atď.
- Pre boolovské polia (`is_dsm_admin`, `is_group_admin`, atď.) skúšame vložiť reťazce, zoznamy, slovníky atď.
- Pre cudzie kľúče skúšame vložiť reťazce a iné neplatné typy namiesto ID.

```
1 def testDataTypes(self):
2     # Define various test values
3     test_values = {
4         "string": "invalid_value",
5         "integer": 12345,
6         "float": 123.45,
7         "list": [1, 2, 3],
8         "dict": {"key": "value"},
9         "boolean": True,
10        "none": None,
11        "bytes": b"binary_data",
12    }
13    # Test boolean field with invalid types
14    for type_name, test_value in test_values.items():
15        if isinstance(test_value, (bool, type(None), int)):
16            continue # These can be converted to boolean
17
18        user = createUser(
19            user.group_id = self.test_group
20            ...
21        )
22
23        with self.assertRaises((TypeError, ValueError, AttributeError)):
24            user.is_dsm_admin = test_value
25            user.save()
26
27    # Similar tests for each additional field...
```

Ukážka 9.3.4: Test typov atribútov

9.3.5 Testovanie hraničných prípadov

Hraničné prípady predstavujú situácie na okraji platných hodnôt alebo špecifické scenáre, ktoré môžu odhaliť chyby v implementácii 9.3.5.

🔗 Príklad testu hraničných prípadov

Navrhované testovacie scenáre:

- Testovanie prázdnych, ale platných hodnôt (napr. prázdny reťazec v nepovinných poliach)
- Testovanie maximálnych dĺžok textových polí
- Testovanie špeciálnych znakov v textových poliach
- Testovanie rôznych formátov e-mailových adries

```
1 def testEmptyOptionalFields(self):
2     # Create a user with empty but valid values in optional fields
3     user = createUser(
4         user.group_id = self.test_group
5         ...
6         user.passwd = "" # Empty string, not null
7         user.phone = "" # Empty string, not null
8         user.photo = "" # Empty string, not null
9     )
10    # Attempt to save and verify that it passes without error
11    try:
12        user.save()
13        saved = True
14    except Exception as e:
15        saved = False
16
17    self.assertTrue(saved)
18
19    # Verify that values were saved as empty strings
20    saved_user = User.get(User.id == user.id)
21    self.assertEqual(saved_user.passwd, "")
22    self.assertEqual(saved_user.phone, "")
23    self.assertEqual(saved_user.photo, "")
```

Ukážka 9.3.5: Test hraničných prípadov

Systematickým testovaním všetkých uvedených aspektov databázovej tabuľky **User** zabezpečíme, že tabuľka správne validuje vstupné dáta, dodržiava definované obmedzenia a zachováva integritu dát. Tieto testovacie scenáre poslúžia ako základ pre implementáciu komplexnej testovacej sady v nasledujúcej časti práce.

Pri testovaní databázových tabuliek je dôležité venovať pozornosť aj testovaniu **getter** a **setter** funkcií, ktoré zabezpečujú prístup k atribútom objektov a ich modifikáciu. Tieto metódy sú základným stavebným kameňom objektovo-orientovaného programovania a správne fungovanie týchto metód je kľúčové pre integritu dát.

9.3.6 Prístup k testovaniu getterov

Gettery sú metódy, ktoré umožňujú čítanie hodnôt atribútov objektu. Pri ich testovaní postupujeme nasledovne 9.3.6:

1. **Testovanie základnej funkčnosti** – V prvom kroku overíme, či getter vracia správnu hodnotu, ktorá bola nastavená pri vytvorení objektu alebo pomocou settera. Copy
2. **Testovanie formátu návratovej hodnoty** – Overíme, či getter vracia hodnotu v očakávanom dátovom type a formáte.
3. **Testovanie null hodnôt** – Ak atribút môže obsahovať null hodnotu, testujeme, či getter správne vracia null a nespôsobuje žiadne výnimky.
4. **Testovanie prístupu k privátnym atribútom** – Overíme, či getter poskytuje prístup k privátnym atribútom objektu, ktoré by inak neboli priamo dostupné.

9.3.7 Prístup k testovaniu setterov

Settery sú metódy, ktoré umožňujú nastavenie hodnôt atribútov objektu. Ich testovanie by malo zahŕňať:

1. **Testovanie základnej funkčnosti** – Overíme, či setter správne nastaví hodnotu atribútu. Po nastavení hodnoty pomocou settera by mal getter vrátiť rovnakú hodnotu. Copy
2. **Testovanie validácie** – Testujeme, či setter vykonáva validáciu vstupných hodnôt a zamietá neplatné hodnoty. Napríklad, ak má byť vek pozitívne číslo, setter by mal zamietnuť záporné hodnoty.
3. **Testovanie konverzie typov** – Overíme, či setter správne konvertuje vstupné hodnoty na požadovaný dátový typ, ak je to súčasťou jeho funkcionality.
4. **Testovanie hraničných prípadov** – Testujeme hraničné hodnoty, ako sú minimálne a maximálne povolené hodnoty, prázdne reťazce alebo špeciálne znaky.
5. **Testovanie prístupu k privátnym atribútom** – Overíme, či setter správne modifikuje privátne atribúty objektu.

🔗 Príklad testu getterov a setterov

Pri implementácii testov pre gettery a settery v tabuľke `User` by sme mohli za predpokladu existujúcich `getter` a `setter` funkcií postupovať takto:

```
1 def testGetterSetter(self):
2     # Create user object
3     user = createUser(
4         user.name = "Jan"
5         ...
6     )
7
8     # Testing getters - verify it returns valid values
9     self.assertEqual(user.getName(), "Jan")
10    self.assertEqual(user.getSurname(), "Novak")
11    self.assertEqual(user.getLogin(), "jnovak")
12    self.assertEqual(user.getEmail(), "jan.novak@example.com")
13    self.assertTrue(user.getDsmAdmin())
14
15    # Testing value change
16    user.setName("Peter")
17    self.assertEqual(user.name, "Peter")
18
19    # Testing null values for optional fields
20    self.assertIsNone(user.phone)
21    user.phone = "+421 900 123 456"
22    self.assertEqual(user.phone, "+421 900 123 456")
```

Ukážka 9.3.6: Test getteru a setteru

9.3.8 Testovanie špeciálnych prípadov

Pri testovaní getterov a setterov je dôležité zvážiť aj špeciálne prípady, ktoré môžu nastať **9.3.7:**

1. **Vypočítané atribúty** – Niektoré gettery môžu vracať hodnoty, ktoré sú vypočítané na základe iných atribútov. V takom prípade testujeme, či výpočet prebieha správne.

👁️ Príklad testu vypočítaného atribútu

```
1 def testComputedAttrib(self):
2     user = createUser(
3         name = "Jan"
4         surname = "Novak"
5     )
6     # Assume that full_name is a computed property
7     self.assertEqual(user.full_name, "Jan Novak")
8
9     # Change the base attributes and verify that the computed attribute
    also changes
10    user.name = "Peter"
11    self.assertEqual(user.full_name, "Peter Novak")
```

Ukážka 9.3.7: Test hraničných prípadov getteru

2. **Validácia v setteroch** – Ak settery obsahujú validačnú logiku, je potrebné testovať rôzne scenáre, ktoré by mali vyvolať chybu validácie 9.3.8.

👁️ Príklad testu validácie v settere

```
1 def testValidEmail(self):
2     user = createUser(
3         ...
4     )
5     # Testing valid email address
6     user.setEmail("jan.novak@example.com")
7     self.assertEqual(user.email, "jan.novak@example.com")
8
9     # Testing invalid email address
10    with self.assertRaises(ValueError):
11        user.email = "neplatny_email"
```

Ukážka 9.3.8: Validácia v setteri

3. **Lazy loading** – Ak sa používa lazy loading pre načítavanie hodnôt z databázy, testujeme, či getter správne inicializuje hodnotu pri prvom prístupe.
4. **Kaskádové zmeny** – Ak zmena jedného atribútu má vplyv na hodnoty iných atribútov, testujeme, či tieto kaskádové zmeny prebiehajú správne 9.3.9.

🔗 Príklad testu kaskádovej zmeny

```
1 def test_kaskadova_zmena(self):
2     user = createUser()
3     user.role = "admin"
4     # Assume that setting the "admin" role automatically sets permission
      flags
5     self.assertTrue(user.is_dsm_admin)
6     self.assertTrue(user.is_group_admin)
7
8     # Changing the role should change the flags
9     user.role = "user"
10    self.assertFalse(user.is_dsm_admin)
11    self.assertFalse(user.is_group_admin)
```

Ukážka 9.3.9: Testovanie kaskádových zmien

Dôkladné testovanie getterov a setterov je základom spoľahlivého testovania objektovo-orientovaných systémov. Aj keď sa tieto metódy môžu zdať triviálne, často obsahujú dôležitú validačnú a konverzačnú logiku, ktorá je kritická pre správne fungovanie systému. Systematickým testovaním getterov a setterov zabezpečíme, že objekty správne spravujú svoje interné stavy, čo prispieva k celkovej robustnosti a spoľahlivosti aplikácie. Testovanie getterov a setterov je obzvlášť dôležité v prípade ORM (Object-Relational Mapping) modelov, ako je náš príklad s tabuľkou `User`, pretože tieto metódy slúžia ako most medzi objektovým modelom aplikácie a relačným modelom databázy. Správna implementácia a testovanie týchto metód zabezpečí, že dáta budú správne reprezentované v aplikačnej logike a budú konzistentne uložené v databáze.

9.3.9 Výkonnostné testovanie

Výkonnostné testovanie predstavuje kľúčovú súčasť komplexnej testovacej stratégie, ktorej hlavným cieľom je zhodnotenie efektivity, rýchlosti a škálovateľnosti systému pri rôznych úrovniach záťaže. Na rozdiel od funkčných testov, ktoré overujú správnosť implementácie, výkonnostné testovanie sa zameriava na kvalitatívne aspekty systému ako odozva, priepustnosť a využitie zdrojov.

V kontexte databázových aplikácií, výkonnostné testovanie nadobúda osobitný význam. Neefektívne navrhnuté SQL dotazy môžu viesť k výraznému spomaleniu aplikácie, najmä pri narastaní objemu dát alebo počtu súbežných používateľov. Dobre navrhnutý systém výkonnostného testovania dokáže včas identifikovať takéto problémové miesta a poskytnúť metriky na kvantifikáciu zlepšenia po optimalizácii.

Implementácia v aplikácii k-Dispatch

V rámci aplikácie k-Dispatch som implementovala komplexný systém výkonnostného testovania zameraný primárne na overenie efektivity databázových operácií. Systém je navrhnutý s dôrazom na štyri kľúčové aspekty:

1. **Porovnávanie alternatívnych implementácií SQL dotazov** – Každý test porovnáva neefektívny a optimalizovaný spôsob implementácie rovnakého dotazu
2. **Simulácia väčších objemov dát** – Využitie generátora testovacích dát na vytvorenie realistickej bázy dát rôznych veľkostí
3. **Testovanie pri súbežnom prístupe** – Simulácia situácií, kedy k systému pristupuje viacero používateľov naraz
4. **Komplexné vyhodnotenie výsledkov** – Detailná analýza zlepšenia výkonnosti s presným meraním časových rozdielov a konzistentné logovanie výsledkov

Pre generovanie dostatočného množstva testovacích dát som vytvorila súbor `seeder.py`, ktorý dokáže napopulovať databázu realistickými záznamami pre používané entity systému. Tento nástroj umožňuje špecifikovať presné množstvo jednotlivých entít, čím je možné otestovať škálovateľnosť systému od malých až po veľké databázy.

Metodológia testovania výkonu SQL dotazov

Jadrom implementovaného výkonnostného testovania je systematické porovnávanie efektívnych a neefektívnych verzií SQL dotazov. Každý test sleduje rovnaký vzor:

1. Najprv sa vykoná dotaz implementovaný neefektívnym spôsobom, typicky obsahujúci vnorené dotazy, neoptimálne spojenia tabuliek alebo nevyužívajúci existujúce indexy.
2. Následne sa vykoná optimalizovaná verzia rovnakého dotazu využívajúca techniky ako spojenia tabuliek (JOIN), Common Table Expressions (CTE), vhodné indexy a správne filtrovanie dát.
3. Pre obe verzie sa meria čas vykonávania, počet vrátených riadkov a ďalšie metriky.
4. Výsledky sa porovnávajú a vypočíta sa percentuálne zlepšenie výkonnosti.

Tento prístup nielen identifikuje problematické miesta v aplikácii, ale poskytuje aj konkrétne ukážky optimalizácií, ktoré sa môžu aplikovať na existujúci kód.

Testovanie pri vysokej záťaži

Pre realistické zhodnotenie výkonnosti databázy v produkčných podmienkach je nevyhnutné testovať systém pri simulovanej vysokej záťaži. Za týmto účelom som implementovala riešenie v súbore `load_test.py`, ktoré umožňuje spúšťať testy s viacerými simulovanými používateľmi súčasne a taktiež umožňuje možnosť špecifikovať počet iterácií jednotlivých testov.

Pre simuláciu reálnych používateľov som využila technológiu multiprocessingu, ktorá umožňuje spúšťať viacero procesov paralelne na dostupných jadrách procesora. Táto metóda ponúka niekoľko výhod oproti tradičnému prístupu s viacvláknovým (multithreading) testovaním:

- **Skutočná paralelizácia** – Na rozdiel od vlákien, ktoré v Pythone kvôli GIL (Global Interpreter Lock) nemôžu dosiahnuť skutočnú paralelizáciu CPU operácií, procesy bežia úplne nezávisle a paralelne
- **Izolácia zdrojov** – Každý simulovaný používateľ má vlastný pamäťový priestor, čo lepšie reflektuje reálnu situáciu v produkčnom prostredí
- **Robustnosť** – Zlyhanie jedného procesu neovplyvňuje ostatné simulované používateľské relácie
- **Efektívne využitie hardvéru** – Riešenie automaticky škáluje počet pracovných procesov podľa dostupných CPU jadier, s optimálnou distribúciou záťaže

Táto implementácia:

- Vytvára definovaný počet simulovaných používateľov, ktorí pristupujú k systému simultánne
- Synchronizuje spustenie dotazov, aby sa dosiahla realistická špička v záťaži
- Identifikuje potenciálne úzke miesta, ktoré sa prejavujú len pri väčšom počte súbežných prístupov

Výhodou tohto prístupu je, že dokáže odhaliť problémy, ktoré by pri sekvenčnom testovaní zostali skryté, napríklad zámky na úrovni databázy, problémy s prepojeniami alebo neefektívne využitie pripojení.

Vyhodnotenie a archivácia výsledkov

Dôležitou súčasťou výkonnostného testovania je metodika zberu, vyhodnotenia a archivácie výsledkov. Implementovaný systém:

- Generuje detailné správy o každom vykonanom teste s podrobnými štatistikami
- Zhromažďuje údaje o systémovej prostredí, v ktorom testy prebiehali (hardvér, verzia databázy, konfigurácia)
- Umožňuje porovnanie výsledkov medzi rôznymi verziami aplikácie alebo konfiguráciami
- Poskytuje prehľadné súhrnné štatistiky pre rýchlu identifikáciu trendov a problémových miest

Výsledné správy obsahujú nielen technické metriky ako časy vykonávania, ale aj prehľadné zhrnutie s percentuálnym vyjadrením zlepšenia.

9.3.10 Testovanie webového rozhrania k-Dispatch

V našom prípade sme sa rozhodli implementovať testy webového rozhrania prostredníctvom nástroja Selenium v kombinácii s programovacím jazykom Python. Toto rozhodnutie bolo motivované najmä dostupnosťou a flexibilitou Pythonu, ktorý poskytuje jednoduché a efektívne rozhranie pre Selenium WebDriver.

Vytvorený test slúži primárne ako názorná ukážka toho, ako môžu automatizované testy webových aplikácií vyzerat v praxi. Dôraz bol kladený na demonštráciu základných testovacích techník na získanie praktického pohľadu na proces testovania a poskytnutie základov pre prípadné rozšírenie testovacej sady v reálnych projektoch. V prípade potreby komplexného testovania produkčnej aplikácie by bolo nutné testovací rámec rozšíriť o ďalšie scenáre a integrovať ho do kontinuálneho integračného procesu.

Kapitola 10

Záver

Predkladaná bakalárska práca sa venovala problematike testovania softvéru s praktickou aplikáciou na demonštračnej aplikácii a systéme k-Dispatch. Boli splnené všetky stanovené ciele vrátane implementácie demonštračnej aplikácie, analýzy, návrhu a implementácie testovacích scenárov pre k-Dispatch. Demonštračná aplikácia knižničného systému poslúžila pre ukážku rôznych typov testovania vrátane jednotkových, integračných a end-to-end testov. Aplikácia bola kontajnerizovaná pomocou technológie Docker a vybavená CI/CD pipeline pre automatizáciu spúšťania testov. Pre k-Dispatch boli analyzované kľúčové moduly Dispatch & Transfer, Remote Machine, Medium Parameters a Dependency Graph, a navrhnuté komplexné testovacie scenáre. Pre databázu aplikácie boli implementované jednotkové testy overujúce všetky databázové metódy, na tabuľke **Users** boli otestované CRUD operácie pre demonštráciu na ďalšie tabuľky, validácia vstupov a dodržiavanie referenčnej integrity. Významnou súčasťou bolo výkonnostné testovanie SQL dotazov s cieľom identifikovať a optimalizovať neefektívne databázové operácie, pričom implementovaný nástroj dokázal generovať testovacie záznamy a simulovať vysokú záťaž. Nakoniec boli vytvorené automatizované UI testy pomocou Selenium pre overenie webového rozhrania aplikácie, začínajúc testom prihlásenia ako šablóny pre ďalší vývoj testovacích scenárov.

10.1 Použité implementačné prístupy a technológie

Pre implementáciu testov bol použitý Python s frameworkom pytest. Výkonnostné testovanie porovnávalo efektívne a neefektívne verzie SQL dotazov, high-load testovanie využívalo multiprocessing pre simuláciu súbežných používateľov. Webové rozhranie bolo testované nástrojom Selenium. Súčasťou práce bola kontajnerizácia pomocou Docker, ktorá zabezpečila konzistentné testovanie. Pre demonštračnú aplikáciu boli vytvorené GitLab skripty implementujúce CI/CD pipeline pre automatické spúšťanie testov pri každom pushnutí do vetvy a taktiež využíva nástroj **Fluent Bit** na monitorovanie logov z kontajnera a nástroj **msmt** na odoslanie emailu s telom chybového logu. Podobný prístup je aplikovateľný aj pre systém k-Dispatch.

10.2 Možnosti ďalšieho rozšírenia

Táto práca položila základy pre systematické testovanie k-Dispatch, no existuje viacero smerov pre ďalší vývoj:

1. **Rozšírenie pokrytia testami** – Implementácia ďalších testovacích scenárov nerealizovaných v rámci tejto práce.
2. **Automatizácia testov pre k-Dispatch** – Implementácia CI/CD pipeline, ktorá by zabezpečila automatické spúšťanie všetkých testov pri zmenách a poskytla by rýchlu spätnú väzbu vývojovému tímu.
3. **Integrácia s monitorovacími nástrojmi** – Prepojenie s nástrojmi ako Prometheus či Grafana pre vizualizáciu výkonnostných metrík.
4. **Rozšírenie testovania webového rozhrania** – Vytvorenie komplexnejšej sady automatizovaných testov pre používateľské rozhranie.
5. **Bezpečnostné testovanie** – Implementácia testov pre autentifikáciu, autorizáciu a ochranu pred bežnými útokmi.

Literatúra

- [1] *Gray Box Testing* Software Testing Fundamentals [online]. 2010. Dostupné z: <http://softwaretestingfundamentals.com/gray-box-testing/>. Aktualizováno 3. 3. 2018, [cit. 2020-01-22].
- [2] *The Selenium Browser Automation Project :: Documentation for Selenium* [online]. 2020. Dostupné z: <https://www.selenium.dev/documentation/en/>. [cit. 2020-04-03].
- [3] AMMANN, P. a OFFUTT, J. *Introduction to Software Testing*. 1. vyd. New York: Cambridge University Press, 2008. 346 s. ISBN 978-0-521-88038-1.
- [4] BROWSERSTACK. *What is White Box Testing? (Example, Types, Techniques)* online. 2025. Dostupné z: <https://www.browserstack.com/guide/white-box-testing>.
- [5] COLLINS, T. *What is Grey Box Testing? (Techniques Example)* online. 2024. Dostupné z: <https://www.browserstack.com/guide/grey-box-testing>.
- [6] CRISPIN, L. a. J. G. *Agile Testing: A Practical Guide for Testers and Agile Teams*. Addison-Wesley Professional, 2009. ISBN 978-0321534460.
- [7] DAS, S. *What is Black Box Testing: Types, Tools Examples* online. 2024. Dostupné z: <https://www.browserstack.com/guide/black-box-testing>.
- [8] FLUENT BIT TEAM. *Fluent Bit Documentation* online. Máj 2025. Dostupné z: <https://docs.fluentbit.io/manual>. Online; accessed 13-May-2025.
- [9] FOUNDATION, P. S. *Pytest: helps you write better programs*. 2015. Dostupné z: <https://docs.pytest.org/en/stable/#documentation>.
- [10] FOUNDATION, P. S. *Unittest — Unit testing framework*. 2025. Dostupné z: <https://docs.python.org/3/library/unittest.html>. Updated on May 11, 2025.
- [11] FOUNDATION, P. S. *Welcome to Hypothesis!* 2025. Dostupné z: <https://hypothesis.readthedocs.io/en/latest/>.
- [12] GEEKSFORGEEKS. *Differences between Functional and Non-Functional Testing* online. 2023. Dostupné z: <https://www.geeksforgeeks.org/differences-between-functional-and-non-functional-testing/>.
- [13] GEEKSFORGEEKS. *Types of Software Testing* online. 2023. Dostupné z: <https://www.geeksforgeeks.org/types-software-testing/>.
- [14] GEEKSFORGEEKS. *Load Testing – Software Testing* online. 2024. Dostupné z: <https://www.geeksforgeeks.org/software-testing-load-testing/>.

- [15] GEEKSFORGEEEKS. *Software Development Life Cycle (SDLC)* online. 28. októbra 2024. Dostupné z: <https://www.geeksforgeeks.org/software-development-life-cycle-sdlc/>.
- [16] GEEKSFORGEEEKS. *Software Testing Life Cycle (STLC)* online. 28. októbra 2024. Dostupné z: <https://www.geeksforgeeks.org/software-testing-life-cycle-stlc/>.
- [17] GEEKSFORGEEEKS. *What is Stress Testing in Software Testing?* online. 2024. Dostupné z: <https://www.geeksforgeeks.org/stress-testing-software-testing/>.
- [18] GEEKSFORGEEEKS. *Performance Testing – Software Testing* online. 2025. Dostupné z: <https://www.geeksforgeeks.org/performance-testing-software-testing/>.
- [19] HACKERONE. *What Is Security Testing?* online. 2024. Dostupné z: <https://www.hackerone.com/knowledge-center/what-security-testing>.
- [20] HAMILTON, T. *What is Software Testing?* online. 2024. Dostupné z: <https://www.guru99.com/software-testing-introduction-importance.html>. [cit. 2024-10-21].
- [21] HELP, S. T. *Types Of Software Testing: Different Testing Types With Details* online. 2024. 2024-10-29. Dostupné z: <https://www.softwaretestinghelp.com/types-of-software-testing/>.
- [22] HELP, S. T. *What is Unit Testing? Types, Tools, and Benefits* online. 2024. 2024-10-29. Dostupné z: <https://www.softwaretestinghelp.com/unit-testing/>.
- [23] INC., D. *Docker Guides*. 2025. Dostupné z: <https://docs.docker.com/guides/>. Accessed on May 11, 2025.
- [24] JAROŠ, M. a JAROŠ, J. K-Dispatch: Enabling Cost-Optimized Biomedical Workflow Offloading. In: *HPDC '24: Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing*. New York: Association for Computing Machinery, 2024, s. 358–360. ISBN 979-8-4007-0413-0. Dostupné z: <https://www.fit.vut.cz/research/publication/13179/>.
- [25] JAROŠ, M.; TREEBY, B.; GEORGIU, P. a JAROŠ, J. K-Dispatch: A Workflow Management System for the Automated Execution of Biomedical Ultrasound Simulations on Remote Computing Resources. In: *Proceedings of the Platform for Advanced Scientific Computing Conference, PASC 2020*. New York: Association for Computing Machinery, 2020, s. 1–10. ISBN 978-1-4503-7993-9. Dostupné z: <https://dl.acm.org/doi/pdf/10.1145/3394277.3401854>.
- [26] KANADE, V. *What Is Regression Testing? Definition, Techniques, and Tools* online. 2022. 2022-09-06. Dostupné z: <https://www.spiceworks.com/tech/devops/articles/what-is-regression-testing/>.
- [27] KOJNOKOVÁ, M. *Non functional testing – typy, výhody a riziká pri jeho ignorácii* online. Dostupné z: <https://msgtester.sk/non-functional-testing/>. [cit. 2024-06-07].
- [28] KOJNOKOVÁ, M. *Web app testing – tools, methods, strategy and more* online. 2024. Dostupné z: <https://msgtester.sk/en/testing-web-applications/>.

- [29] M, K. *A Detailed Guide To Grey Box Testing* online. 2024. Dostupné z: <https://www.qatouch.com/blog/grey-box-testing/>.
- [30] M, K. *What is Black Box Testing? Techniques, Types and Examples* online. 2024. Dostupné z: <https://www.qatouch.com/blog/black-box-testing/>.
- [31] M, K. *What is White Box Testing? Techniques, Types, and Examples* online. 2024. Dostupné z: <https://www.qatouch.com/blog/white-box-testing/>.
- [32] MAYERS, G.; SANDLER, C. a BADGETT, T. *The Art of Software Testing*. Wiley, 2011. ISBN 978-1118031964.
- [33] MRÁZIK, M. *Automatické testování software*. Brno, 2020. Bachelor's thesis. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Pavol Korček, Ph.D.
- [34] PATTON, R. *Testování softwaru*. 1. vyd. Praha: Computer Press, 2002. 314 s. ISBN 80-7226-636-5. Přeloženo z: Software Testing.
- [35] REQTEST. *What is Regression Testing, Types, Need, Techniques and Tools* online. 2017. 2017-09-30. Dostupné z: <https://reqtest.com/en/knowledgebase/regression-testing-types-techniques-tools/>.
- [36] SEELA, S. *What Is Integration Testing? (Tutorial With Integration Testing Example)* online. 2023. 2024-02-13. Dostupné z: <https://www.softwaretestinghelp.com/what-is-integration-testing/>.
- [37] YASAR, K. *Software testing* online. 2022. Dostupné z: <https://www.techtarget.com/whatis/definition/software-testing>.
- [38] YEGULALP, S. *What is Docker? The spark for the container revolution* InfoWorld [online]. 2019. Dostupné z: <https://www.infoworld.com/article/3204171/what-is-docker-the-spark-for-the-container-revolution.html>. [cit. 2020-05-19].

Príloha A

Obsah odovzdaných súborov

- README.md - Dokumentácia k poskytnutej zložke, jej spustení a detailoch spustenia
- thesis - Adresár obsahujúci bakalársku prácu so všetkými zdrojmi
- k-Dispatch-CI - Adresár obsahujúci demonštračnú aplikáciu.
 - **app** - Adresár obsahujúci hlavné zdrojové súbory aplikácie. Zahŕňa implementačné Python triedy a komplexnú testovaciu infraštruktúru.
 - * **tests** - Zložka obsahujúca End-to-End (E2E), integračné a jednotkové testy aplikácie.
 - * **database.py** - Databázová vrstva aplikácie.
 - * **library.py** - Knížničné vrstva aplikácie.
 - **doc** - Adresár obsahujúci komplexnú dokumentáciu testovacích prípadov vrátane vizuálnych diagramov.
 - **monitoring** - Adresár obsahujúci konfiguračné súbory a nástroje pre komplexné monitorovanie aplikácie, zhromažďovanie logov, nastavenia metrík a spracovanie chybových hlásení.
 - **docker-compose.yml** - Konfiguračný súbor definujúci služby, siete a zväzky pre Docker aplikáciu. Určuje parametre pre Fluent Bit a samotný kontajner aplikácie.
 - **Dockerfile** - Súbor s postupnosťou inštrukcií pre vytvorenie Docker obrazu aplikácie.
 - **entrypoint.sh** - Shell skript, ktorý sa vykoná pri spustení Docker kontajnera. Zabezpečuje inicializáciu testov, konfiguráciu emailových notifikácií a vytvorenie naplánovaných úloh.
 - **README.md** - Úvodný dokument s popisom projektu, inštrukciami pre inštaláciu a základným návodom na použitie.
- database - Adresár obsahujúci poskytnuté časti aplikácie k-Dispatch a moju prácu
 - **database**.
 - * **app** - Adresár obsahujúci prevzaté zdrojové súbory aplikácie implementujúce základnú funkcionality.
 - * **database** - Adresár obsahujúci prevzatú štruktúru databázy vrátane schém, tabuliek a definícií vzťahov.

- * **db_unit_tests** - Adresár s komplexnou sadou jednotkových testov zameraných na verifikáciu funkcionality jednotlivých tabuliek databázy.
- * **performance_tests** - Adresár obsahujúci výkonnostné testy databázy a detailné záznamy z ich priebehu pre analýzu efektivity implementácie.
- * **web_tests** - Adresár s implementáciou testov webovej aplikácie využívajúcich framework Selenium pre automatizáciu testovania užívateľského rozhrania.
- * **CONTRIBUTION.md** - Dokumentácia popisujúca autorské a prevzaté súbory s jasným oddelením vlastnej a cudzej práce.
- * **docker-compose.yml** - Konfiguračný súbor pre orchestráciu viacerých Docker kontajnerov aplikácie a jej závislostí.
- * **Dockerfile** - Definícia Docker obrazu pre vytvorenie izolovaného prostredia aplikácie.
- * **entrypoint.sh** - Shell skript vykonávaný pri spustení Docker kontajnera. Zabezpečuje inštaláciu databázy, inicializáciu dát cez seeder a spustenie webového servera.
- * **install_db.py** - Prevzatý a modifikovaný Python skript pre automatizovanú inštaláciu a konfiguráciu databázy.
- * **seeder.py** - Nástroj pre naplnenie databázy predvolenými hodnotami s možnosťou konfigurácie ľubovoľného počtu testovacích záznamov.
- * **README.md** - Dokumentácia poskytujúca prehľad projektu, inštalačné inštrukcie a základný návod na použitie aplikácie.