Automatické hodnocení úkolů v předmětu Programování síťových aplikací

Bc. Martin Krčma

Diplomová práce
2024

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

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

Akademický rok: 2023/2024

ZADÁNÍ DIPLOMOVÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Bc. Martin Krčma**
Osobní číslo:	**A22301**
Studijní program:	**N0613A140022 Informační technologie**
Specializace:	**Softwarové inženýrství**
Forma studia:	**Prezenční**
Téma práce:	**Automatické hodnocení úkolů v předmětu Programování síťových aplikací**
Téma práce anglicky:	**Automatic Assessment of Tasks in the Course Programming Network Applications**

Zásady pro vypracování

1. Prostudujte současná zadání a vzorová řešení pro cvičení v předmětu Programování síťových aplikací.
2. Pro vybrané úlohy navrhněte způsoby testování správnosti studentských řešení.
3. Implementujte testování úkolů spustitelné na počítači studenta i na GitLab serveru pro odevzdání úloh.
4. Vytvořte repozitáře úloh na serveru GitLab a pomocí kontinuální integrace implementujte automatické testování odevzdaných úkolů.
5. Doplňte zadání úloh o podrobné instrukce, aby studenti zvládli řešení vypracovat samostatně.

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

Seznam doporučené literatury:

1. HAROLD, Elliotte. Java Network Programming. 4th edition. O'Reilly Media, 2013. ISBN 978-1449357672.
2. BECK, Kent. Test Driven Development: By Example. Kent Beck: Addison-Wesley Professional, 2003. ISBN 978--0321146533.
3. CHACON, Scott. Pro Git. 2nd edition. Apress, 2014. ISBN 978-1484200773.
4. KOSKELA, Lasse. Effective Unit Testing. Manning Publications, 2013. ISBN 978-1935182573.
5. LOELIGER, Jon a Matthew MCCULLOUGH. Version Control with Git. 2nd edition. O'Reilly Media, 2012. ISBN 978--1449316389.

Vedoucí diplomové práce: **Ing. Tomáš Dulík, Ph.D.**
Ústav informatiky a umělé inteligence

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

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

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

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

Ve Zlíně dne 5. ledna 2024

Prohlašuji, že

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

Prohlašuji,

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

Ve Zlíně, dne

Martin Krčma v.r.
podpis studenta

ABSTRAKT

Tato diplomová práce se zabývá návrhem a implementací univerzálního black box nástroje pro hodnocení studentských řešení úkolů v předmětu Programování síťových aplikací. Na začátku práce je proveden rozbor a analýza aktuálních úkolů a jsou stanoveny cíle práce. U vybraných úkolů je určen postup při jejich testování. Zvolené postupy jsou následně implementovány s využitím univerzálním softwaru, který bude zajišťovat automatizované hodnocení studentských aplikací. Testování vybraných úkolů bude prováděno jak lokálně, tak v repozitáři na GitLab serveru pomocí kontinuální integrace.

Klíčová slova: Síťové aplikace, Automatizované hodnocení, Testování softwaru, Automatizace testů, Java, GitLab, Gradle.

ABSTRACT

This master's thesis deals with the design and implementation of a universal black box tool for evaluating student solutions to assignments in the Network Application Programming course. The work begins with an analysis of current assignments and the establishment of work objectives. A testing procedure is defined for selected assignments. These selected procedures are subsequently implemented using universal software, which will ensure automated evaluation of student applications. Testing of selected assignments will be performed both locally and in the repository on the GitLab server using continuous integration.

Keywords: Network Applications, Automated Assessment, Software Testing, Test Automation, Java, GitLab, Gradle.

Chtěl bych poděkovat panu Ing. Tomáši Dulíkovi, Ph.D. za odborné vedení této diplomové práce a cenné rady.

Prohlašuji, že odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou totožné.

OBSAH

ÚVOD

Tato diplomová práce se zaměřuje na implementaci automatizovaného hodnocení úloh v předmětu Programování síťových aplikací. Pro efektivní hodnocení studentských úloh v tomto předmětu bude využita black box technika softwarového testování. Dalším cílem práce je příprava sady úloh, která bude doplněna o podrobná zadání, aby studenti byli schopni samostatně vypracovat všechny úkoly.

První část této práce jsou popsány úkoly, které jsou aktuálně zařazeny ve výuce předmětu Programování síťových aplikací. V části kapitoly 1.2 jsou rozebrána všechna tato zadání, což poskytlo důležitý základ pro praktickou část práce. Způsob testování správnosti studentských řešení pro vybrané úlohy, jak je specifikováno v 2. bodu zadání, bude podrobně probírán v kapitolách 8 a 9.

Nejrozsáhlejší část práce se věnuje implementaci a popisu testovacích mechanismů. Zde bude popsána struktura samotného black-box testovacího nástroje a popis nejdůležitějších částí jeho implementace. Nebude také chybět vysvětlení způsobu jeho konfigurace a definování testovacích sad. Jedná se o 3. bod zadání této práce a konkrétně se tím zabývají kapitoly 10, 11 a 12.

Dalším úkolem této práce bylo vytvoření repozitářů úloh pro server GitLab a zajištění automatického testování odevzdaných úkolů s využitím kontinuální integrace. Postup při vytváření projektu je popsán v části kapitoly 8.4 a konfigurace CI/CD pipeliny se nachází v části kapitoly 11.14. Repozitáře pro všechny úlohy jsou připravené v digitální příloze této práce.

Posledním úkolem této práce je doplnění podrobného zadání pro všechny vybrané úlohy. Tím je zajištěno, že studenti budou mít veškeré potřebné informace pro úspěšné vypracování úkolů. Zadání pro každou úlohu, jsou zapsány v repozitářích pomocí markdown. Všechny tyto zadání jsou zahrnuty také jako součást příloh na konci této práce.

I. TEORETICKÁ ČÁST

1 CÍLE PRÁCE

V této kapitole bude představen hlavní účel a smysl diplomové práce. Bude zde také proveden rozbor všech aktuálních úloh z předmětu Programování síťových aplikací, který je vyučován na Fakultě aplikované informatiky Univerzity Tomáše Bati ve Zlíně. Na závěr kapitoly budou stanoveny cíle práce.

1.1 Důvody vzniku práce

Hlavním důvodem vzniku této práce byl zlepšení kvality výuky předmětu Programování síťových aplikací. Úlohy budou rozšířeny o automatizované hodnocení a tak i ulehčení jejich opravování. Studenti budou mít zároveň možnost okamžitě vidět, zda jejich řešení funguje správně. Dalším důvodem byla potřeba částečně přepracovat stávající sadu úloh, která je v předmětu již delší dobu stejná.

1.2 Rozbor úloh v předmětu programování síťových aplikací

Nyní bude proveden rozbor všech aktuálních úloh, které jsou součástí výuky předmětu Programování síťových aplikací. Seřazení úkolů v této časti, odpovídá pořadí, jak je studenti v předmětu vypracovávají.

1.2.1 Email sender

Cílem úlohy email sender je vytvořit jednoduchého klienta pro odesílání e-mailů v jazyce Java. V tomto úkolu je třeba implementovat předloženou šablonu kódu. Veškerá implementace bude ve třidě pro odesílání emailů a její funkcionalita bude volána z hlavní třídy. Testování funkčnosti odesílání bude provedeno na SMPT serveru UTB.

Jde o první úlohu, kde student pochopí základní principy síťové komunikace a základní princip odesílání emailů.

1.2.2 Telnet klient bez použitím vláken

V úloze studenti musí implementovat program, který funguje jako telnet klient. Vytvoří soket s definovanou IP adresou a portem. Program bude umožnovat odesílání zpráv na daný soket a zobrazovat odpovědi z protistrany na konzoli.

Úkol klade důraz na správnou práci se sokety, správnou obsluhu vstupního a výstupního streamu soketu a řešení potenciálního blokování vstupu a výstupu.

1.2.3 Telnet klient s použitím vláken

Tento úkol se zaměřuje na vylepšení předchozí varianty telnet klienta. Vylepšením je zde myšleno použití více vláken. Zatímco v předchozí verzi tohoto programu bylo využíváno pouze jedno vlákno, tak zde budou vlákna dvě. Jedno vlákno bude zajišťovat odesílání zpráv a druhé vlákno bude zajištovat přijímání a zobrazování těchto zpráv v konzoli.

Úkol je tedy primárně zaměřen na práci s více vlákny, a jaké výhody to přináší v oblasti síťových aplikací.

1.2.4 Telnet virus

Úloha telnet virus se opět zaměřuje na telnet klienta. Tentokrát ale není cílem vytvořit program, který bude odesílat a přijímat zprávy, ale cílem bude vytvořit takový program, který testovaný server zahltí velkým množstvím zpráv. Za tímto účelem je nutné použít vyšší množství vláken. Zadáním úlohy je použít přibližně 10 000 vláken s využitím třídy ThreadPoolExecutor, která je standartně dostupná v JDK.

1.2.5 Server s více vlákny

Úkolem je vytvořit server, který dokáže přijímat spojení od libovolného počtu klientů. Ve hlavní smyčce programu se nekonečně opakují instrukce pro přijímání nových spojení pomocí metody accept(). Každé nové spojení je následně předáno nové instanci vlákna, které se stará o komunikaci s klientem prostřednictvím vstupního a výstupního streamu spojení tohoto klienta. Samotný server pak provádí funkci "echo". To znamená, že klientovi, který mu pošle zprávu, odpoví zpět stejnou zprávou. Tato funkcionalita je zajištěna ve vlákně, které zpracovává každé spojení s klientem.

Doposud ve všech předchozích úkolech se vždy pracovalo pouze s jednou cílovou adresou. Tento úkol se tedy primárně zaměřuje na získání zkušeností s aplikacemi, které vyžadují komunikaci s vyšším množstvím klientů.

1.2.6 Server s více vlákny pomocí ExecutorService

Zadání tohoto úkolu z hlediska funkcionality je stejné jako v předchozím případě. Nyní však je hlavním úkolem optimalizace serveru. To bude v tomto případě zajištěno využitím třídy ThreadPoolExecutor, namísto vytváření nových instancí vláken pro každého klienta, jak tomu bylo v předchozím případě.

Tento úkol se zaměřuje na to, jaké výhody přináší využití poolu vláken oproti neustálému vytváření nových vláken a jejich následnému odstraňování za běhu aplikace.

1.2.7 Instant Message server

Úkol se zaměřuje na vytvoření serveru pro okamžité odesílání zpráv, který umožňuje komunikaci mezi více klienty. K tomuto úkolu je k dispozici funkční varianta serveru, který však vyžaduje implementovat další funkce. Úkolem je přidat nové funkce, které zahrnují identifikaci uživatele pomocí jeho zvolené přezdívky a zasílání soukromých zpráv. Za bonusové body je možné přidat funkci pro vytváření soukromých diskuzních místností.

Pro ověření funkčnosti se student k serveru přípoji s Telnet klientem, který již vytvořil dříve v rámci úkolu. K serveru se připojí s několika instancemi Telnet klienta a ověří odesílání soukromých zpráv a veřejných zpráv všem ostatním.

Jde o pokročilejší úkol, ve kterém studenti uplatní všechny získané znalosti z předchozích úkolů tohoto předmětu.

1.2.8 Web crawler

Cílem je vytvořit program, který prohledá web a vypíše 20 nejčastěji vyskytujících se slov. Prohledávání webu bude probíhat zadaným URL a zohlední také maximální hloubku zanoření. Program je ovládán pomocí parametrů příkazového řádku, jako je URL stránky a maximální hloubka zanoření.

Jedná se o pokročilejší úkol. Je třeba zajistit správné parsování HTML stránek, procházení všech odkazů na webových stránkách a analyzovat četnosti slov, které se na stránkách nacházejí. Za bonusové body je program možné rozšířit o paralelní zpracovaní ve více vláknech, ošetření kódování stránek a využívání Java knihovny JSOUP.ORG.

V tomto úkolu jde primárně o práci s obsahem na webových stránkách a analýzou jejich obsahu. Sekundárně, pokud se student rozhodne vypracovat variantu za bonusové body, jde i o efektivní zpracovaní více úloh v jednom okamžiku.

1.2.9 Servlet pro zpracování obrázků

Zadáním úkolu je vytvořit HTTP Servlet, který umožní uživateli nahrát obrázek a poté provést nad tímto obrázkem transformaci pomocí minimálně dvou dostupných filtrů. Transformovaný obrázek spolu s původní verzí obrázku budou zobrazeny na HTML stránce.

Studenti mají k dispozici plně funkční servlet s jedním filtrem, který mohou využít jako výchozí bod pro implementaci požadované funkcionality. Při implementaci je důležité správně nakonfigurovat a použít dostupné filtry a zpracovat vstup uživatele, aby bylo možné provést požadovanou transformaci obrázku.

Za prémiové body student může doplnit ošetření sessions, aby cizí uživatelé vzájemně neviděli své obrázky. Dále také implementovat efektivnější zobrazení obrázků, zajistit efektivnější načítání obrázků a nahradit pomalé konvoluční filtry obrázků za funkce z knihovny JTransform, která dokáže FFT transformaci počítat i na více vláknech.

1.2.10 SSL soket

Úkol spočívá v implementaci jednoduchého SSL "telnet" klienta a SSL "telnet echo" serveru v Javě. Klient a server budou využívat šifrovanou komunikaci pomocí SSL. Základní implementace serveru a klienta je poskytnuta studentům v příloze úkolu. Pro spuštění serveru bude potřeba certifikát, který studenti musí vygenerovat pomocí nástroje "keytool". Tento certifikát je nutný pro zabezpečenou komunikaci.

Tento úkol má studenty seznámit se základními principy šifrované komunikace v oblasti síťových aplikací.

1.2.11 Instant Message server a klient s Websockety

Úkol IM server je rozšířením předešlého úkol „Instant Message serveru“ o grafické uživatelské rozhraní. Toto GUI bude implementováno pomoci Java Websocketu. Výsledná aplikace bude tedy webová aplikace.

Úloha se primárně zaměřuje na práci s web sokety v jazyce Java a ve druhé řadě také na propojení back-end částí aplikace, která zajišťuje komunikaci mezi uživateli a front-end části, která v tomto případě umožňuje uživatelům posílání zpráv mezi sebou.

1.2.12 Web služba

Tento úkol je bonusový a není tak pro studenty povinný. Úkol spočívá v implementaci klienta pro webovou službu s protokolem SOAP, poskytovanou českou státní správou. Webová služba by měla být vybrána z nabídky, která je uvedená v přednáškách předmětu síťových aplikací.

Pro implementaci student můžete využít knihovny pro webové služby, které jsou součástí Java SE od verze 6, nebo také starší/tradiční knihovny jako Apache Axis, CXF a další.

1.2.13 Ping flood s využitím Raw sockets

Obdobně jak u předchozího případu, jde o bonusový úkol. Úkolem je implementace programu, který umožní zahltit cílový počítač pakety "ping". K tomu bude využívat Java knihovnu RockSaw, která umožňuje práci s raw sokety.

Jde zde primárně o základní pochopení síťové komunikace v systému, na nejnižší úrovni.

1.3 Stanovení cílů práce

Hlavním cílem této práce je navrhnout a implementovat automatizované hodnocení pro všechny vybrané úlohy. Za tímto účelem bude vyvinut univerzální softwarový nástroj pro testování správnosti studentských řešení úkolů v předmětu Programování síťových aplikací. Tento nástroj by měl být schopen spustit testy jak na počítači studenta, tak i na GitLab serveru. Tento software bude univerzální, a bude jednoduše možné jeho správnou konfigurací nastavit na testování úkolu podobného charakteru i v pozdějších letech. Druhotným cílem je přepracovat vybraná zadání a doplnit je o podrobné instrukce, aby studenti byli schopní úlohy úspěšně vypracovat.

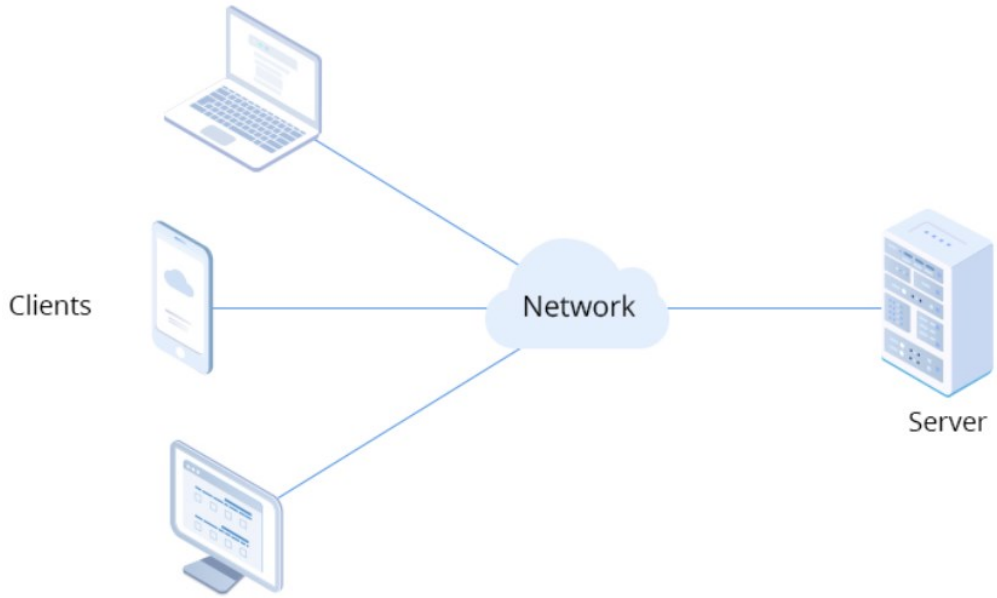
2 ZÁKLADNÍ POJMY SÍŤOVÝCH APLIKACÍ

Obsahem kapitoly bude popis základních pojmů síťových aplikací. Budou vysvětleny základní komunikační modely, které tyto aplikace využívají a také jejich architektury. Ke konci této kapitoly budou popsány typy síťových aplikací, které se v dnešní době nejčastěji používají.

2.1 Klient-Server komunikační model

V klient-server komunikačním modelu existuje minimálně jeden server. Server je centrální prvek, který zprostředkovává veškerou komunikaci mezi klienty. Klienti se k serveru připojují, aby prováděli určité úkoly. Klienti mohou být různá zařízení, jako jsou mobilní telefony, počítače nebo notebooky. Servery v síti mohou zajišťovat různé funkce, jako je správa uživatelského přístupu, ukládání dat, řízení internetového připojení a monitorování síťového provozu. [2]

Mezi výhody klient-server komunikačního modelu patří centralizovaný management, snadná škálovatelnost, zvýšená bezpečnost dat, jednoduchá správa a snadná integrace nových funkcí. Mezi nevýhody patří riziko přetížení sítě, vysoké náklady na pořízení a údržbu, riziko výpadku sítě v případě selhání serveru. [2]

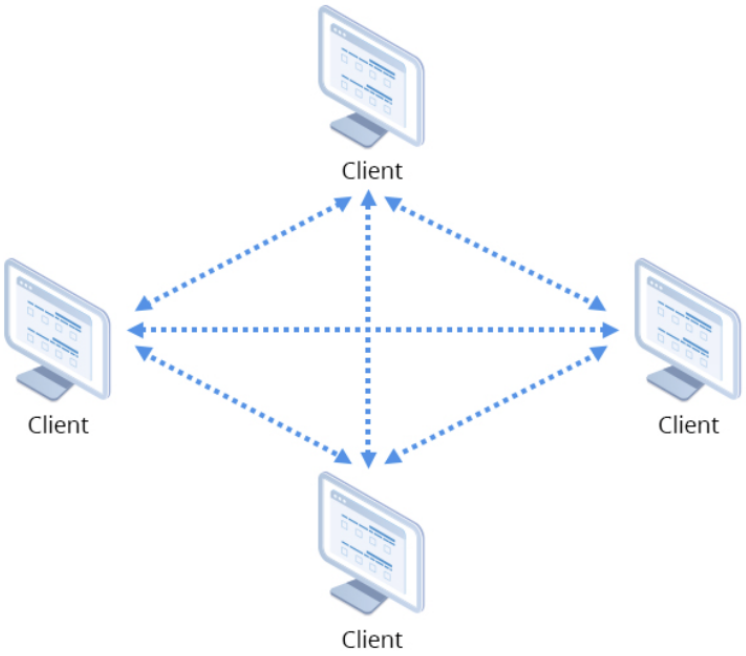


Obrázek 1. Klient-Server komunikační model [2]

2.2 P2P komunikační model

V P2P komunikačním modelu neexistuje žádný centrální server, jak tomu bylo u modelu klient-server. Namísto toho jsou všichni klienti v síti propojeny mezi sebou a sdílejí informace a zdroje. V síti P2P může každý počítač fungovat jako klient nebo jako server, což mu umožňuje žádat o služby nebo je poskytovat. Každý počítač je nazýván peer a má stejné schopnosti a práva v síti jako ostatní. Žádný peer nemá kontrolu nad ostatními. [2]

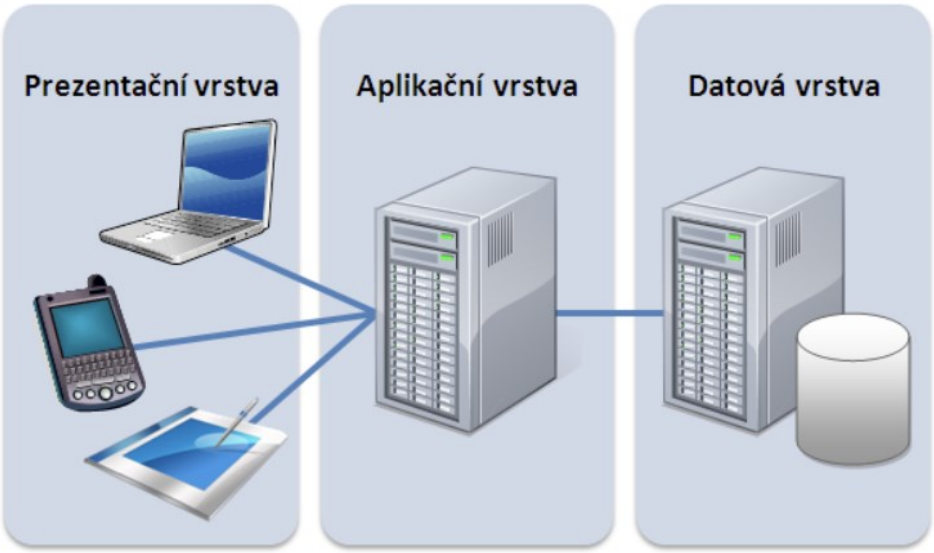
Mezi výhody patří jednoduché sdílení dat, flexibilita, škálovatelnost, nízké náklady, odolnost proti poruchám a výpadkům. Nevýhodou je nemožnost centralizovaného řízení, složitější správa sítě a závislost na dostupnosti peerů v síti. [2]



Obrázek 2. P2P komunikační model [2]

2.3 Třívrstvá architektura

Třívrstvá architektura představuje strukturu informačních systémů, která je rozdělena do tří základních vrstev: prezentační, aplikační a datová vrstva. Prezentační vrstva slouží k interakci s uživatelem a zajišťuje zobrazení dat. Aplikační vrstva zprostředkovává výpočty a operace mezi vstupem a daty. Datová vrstva je nejnižší vrstvou, která řídí práci s daty v databázi a provádí datově-funkční operace. [3]



Obrázek 3. Třívrstvá architektura IS [3]

2.4 Architektura orientovaná na služby

Architektura orientovaná na služby, představuje koncept tvorby informačních systémů, který se zaměřuje na poskytování a využívání služeb. Nejedná se o konkrétní technologii, ale spíše přístup k tvorbě softwaru. [5]

Jádrem je myšlenka, že IS je tvořen souborem poskytovaných služeb, které společně tvoří požadovanou funkcionalitu. Tento přístup umožňuje flexibilně měnit jednotlivé komponenty systému bez závislosti na konkrétních technologiích a aplikacích. [5]

2.5 Vícevrstvá architektura klienta

Při vývoji aplikací se často setkáváme s potřebou navrhovat aplikace s komplexní architekturou, která je rozdělena do více vrstev. Nyní si popíšeme základní dělení architektury softwarových klientů a jejich výhody a nevýhody.

2.5.1 Tlustý klient

Tlustý klient je počítačový systém, který obsahuje jak prezenční, tak aplikační vrstvu a připojuje se přímo k serveru, nebo databázi. Tlustý klient stahuje velké objemy dat přes síť, zpracovává je a výsledky pak přenáší zpět na server. Tlustý klient vyžaduje výkonný hardware a vyžaduje také vysokou šířku pásma pro komunikaci se serverem. Rovněž je u tlustého klienta obtížnější nasazení a jeho následná aktualizace. Výhodou je, že server nevyžaduje tak výkonný hardware a je zde rychlá odezva s možností práce offline. [4]

2.5.2 Tenký klient

Jde nejčastěji webový prohlížeč, který komunikuje pouze s prezentační vrstvou. Tenký klient neprovádí žádnou rozhodovací logiku a veškeré zpracování dat probíhá na serveru. Pro tenké klienty je minimalizována náročnost instalace a konfigurace. Tenký klient vyžaduje méně výkonný hardware a nevyžaduje také vysokou šířku pásma pro komunikaci se serverem, jak tomu je u tlustého klienta. Není zde žádná možnost práce offline, ale naopak je mnohem snadnější jeho nasazení, následné aktualizace a správa. [4]

2.6 Typy síťových aplikací

V této části si popíšeme typy síťových aplikací. Vybral jsem ty typy aplikací, které se v dnešní době nejvíce využívají a lidé se s nimi každý den setkávají.

1. **Webové aplikace** – Jsou to aplikace, které jsou přístupné prostřednictvím webového prohlížeče. Tyto aplikace běží na webovém serveru a poskytují uživatelům možnost interakce s obsahem prostřednictvím svého prohlížeče.
2. **Komunikační aplikace** – Komunikační aplikace umožňují uživatelům komunikovat a posílat si informace přes síť. Patří sem například e-mailové aplikace, instant messaging aplikace, aplikace umožňující videohovor a další.
3. **Multimediální aplikace** – Tyto aplikace pracují s různými formami multimediálního obsahu, jako jsou obrázky, videa nebo zvuk. Patří sem například aplikace pro streamování videa a hudby.
4. **Herní aplikace** – Jde o herní aplikace, které umožnují hráčům hrát s ostatními hráči přes síť. Může se jednat například o počítačové hry, mobilní hry nebo hry ve virtuální realitě.
5. **Databázové aplikace** – Databázové aplikace umožňují uživatelům ukládat, organizovat a manipulovat s daty uloženými v databázích. Tyto aplikace, využívají převážné jen vývojáři aplikací.
6. **IoT aplikace** – Tyto aplikace jsou využívány pro ovládání nebo monitorování různých zařízení, které jsou připojených k síti. Patří sem například aplikace pro chytré domácnosti.

3 SÍŤOVÁ KOMUNIKACE A PROTOKOLY

Způsob, jakým aplikace v síti mezi sebou komunikují je základní a nezbytnou znalostí pro vývoj síťových aplikací. Tato kapitola se zabývá základními principy a technologiemi, které jsou důležité pro pochopení komunikace aplikací v síti.

Jako první se zaměříme na popis a porovnání modelů OSI a TCP/IP. Následně bude popsán způsob adresace zařízení v síti. Další část se bude zabývat nejdůležitějšími protokoly, které se využívají při komunikaci v sítích a nakonec budou popsány způsoby směrování paketů v síti.

3.1 Model OSI a TCP/IP

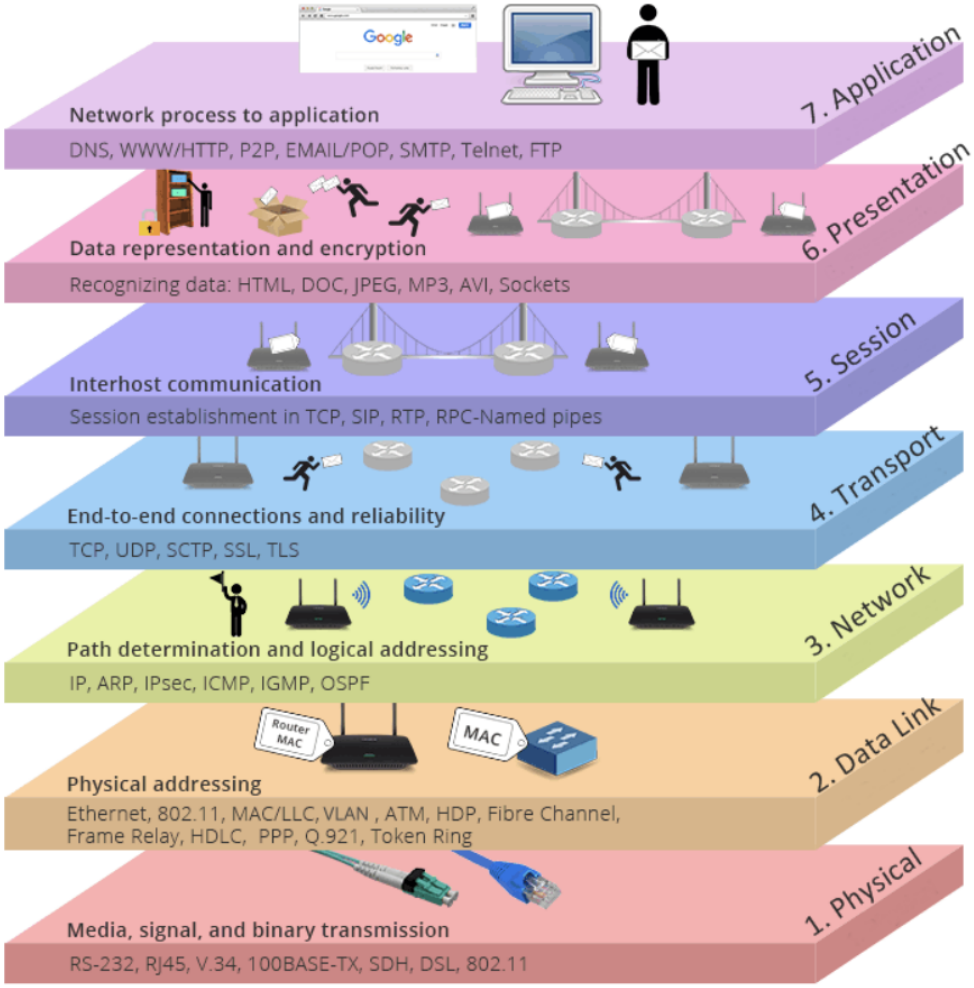
Modely OSI a TCP/IP představují základní rámce pro porozumění komunikačních procesů, které probíhají v počítačových síťových. Zaměříme se na Model OSI, který rozděluje síťovou komunikaci do sedmi vrstev. Poté bude popsán Model TCP/IP, který je standardem v dnešních sítích.

3.1.1 Model OSI

Tento model definuje 7 vrstev, které rozdělují práci mezi různé softwarové a hardwarové komponenty sítě. Díky tomuto rozdělení je komunikace efektivnější a srozumitelnější. Jde o konceptuální teoretický model, který vysvětluje principy komunikace. [6]

1. **Fyzická vrstva** – Definuje fyzické specifikace datového připojení, jako jsou uspořádání pinů konektoru nebo provozní napětí. Také obstarává přenos dat ve fyzickém médiu.
2. **Linková vrstva** – Zajišťuje přímý přenos dat mezi dvěma systémy. Převádí data z fyzické vrstvy do strukturovaných rámců. Zajišťuje adresaci pomocí MAC adres.
3. **Síťová vrstva** – Stará se o směrování paketů a logickou adresaci, umožňuje komunikaci mezi různými uzly v síti.
4. **Transportní vrstva** – Zajišťuje přenos dat mezi systémy a udržuje kvalitu přenosu a integrity dat prostřednictvím funkce korekce chyb.
5. **Relační vrstva** – Řídí spojení mezi počítači, zajišťuje autentizaci, autorizaci a řídí výměnu dat mezi nimi.
6. **Prezentační vrstva** – Tato vrstva zajišťuje kompatibilitu dat s komunikačními prostředky a provádí převod dat do vhodné formy, jako je komprese nebo šifrování dat.

7. **Aplikační vrstva** – Vrstva interaguje přímo s aplikacemi systému a zajišťuje komunikační funkce. Tato vrstva je nejblíže koncové aplikaci a zároveň definuje protokoly pro tyto aplikace.



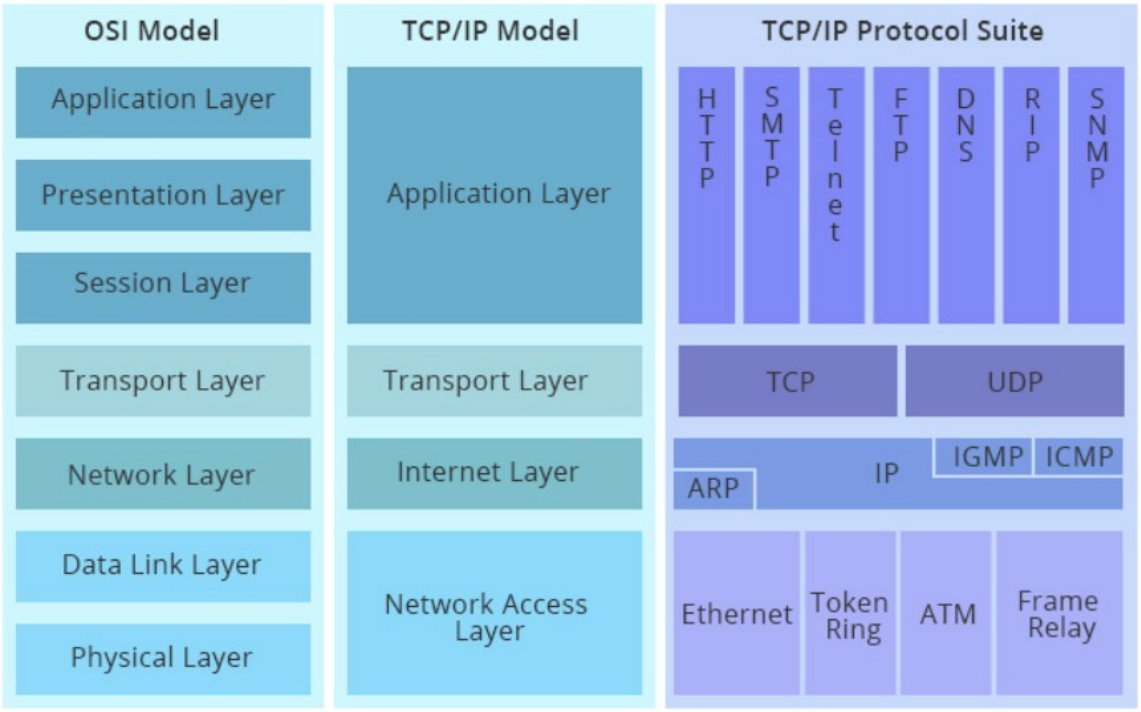
Obrázek 4. Model OSI [6]

3.1.2 Model TCP/IP

Model TCP/IP je složený ze čtyř vrstev. Jeho název vychází z jeho dvou nejdůležitějších protokolů TCP a IP. Model TCP/IP je založen na standardizovaných protokolech vyvinutých pro Internet narozdíl od modelu OSI, který je nezávislý na protokolech a je spíše obecný. Nyní stejně jako v předchozím případě popíšeme jednotlivé vrstvy. [6]

1. **Síťová vrstva** – Je to nejnižší vrstva tohoto modelu. Stejně jak v předchozím případě i zde zajišťuje přenos dat ve fyzickém médiu.
2. **Internetová vrstva** – Internetová vrstva zajišťuje adresování hostitelů a směrování. Hlavním protokolem této vrstvy je protokol IP.

3. **Transportní vrstva** – Transportní vrstva řídí spojení mezi hosty a zajišťuje spolehlivý, nebo nespolehlivý přenos dat. Hlavními protokoly této vrstvy jsou TCP a UDP.
4. **Aplikační vrstva** – Stejně jak u modelu OSI, poskytuje aplikacím přístup ke službám ostatních vrstev a definuje protokoly, které aplikace používají k výměně dat.



Obrázek 5. Model TCP/IP v porovnání s modelem OSI [6]

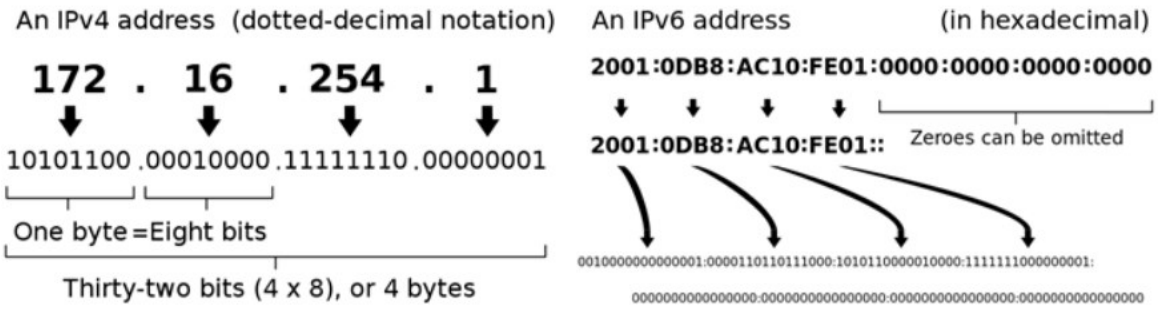
3.2 Adresace v síťové komunikaci

Nyní se zaměříme na způsob, jakým jsou identifikovány jednotlivá zařízeni v síťové komunikaci. Jako první bude popsán IP protokol. Následně bude vysvětleno co je to port a soket a k čemu slouží v oblasti síťových komunikací.

3.2.1 IP protokol

IP protokol je základní protokol TCP/IP modelu, který řídí komunikaci mezi sítěmi na Internetu. Každému zařízení připojenému k síti je přiřazena jedinečná IP adresa pro identifikaci. IP adresy jsou obvykle zapsány v podobě čtyř oddělených dekadických čísel v rozsahu od 0 po 255. Při práci s IP protokolem jsou zapotřebí i směrovací protokoly, jako je OSPF, IS-IS nebo BGP, které pomáhají směrovačům určovat cestu, kterou budou data v síti přenášena. IP protokol nedokáže zaručit spolehlivost přenosu, ani dodržení pořadí příchozích paketů. [7]

Existují dvě verze IP protokolu: IPv4 a IPv6. Protokol IPv6 je vylepšením IPv4. Jejich funkce se nijak neliší, jediný rozdíl je ve velikosti samotné adresy zařízeni v sítí. U protokolu IPv4 je to 32 bitů a u protokolu IPv6 je to 128 bitů. Protokol IPv6 tak nabízí mnohonásobné vyšší množství zařízení v síti, které umožňuje adresovat. Dalším rozdílem je způsob zápisu. Adresa IPv6 se zapisuje v hexadecimální podobě po 16 bitových částech, které jsou odděleny dvojtečkou. [7]



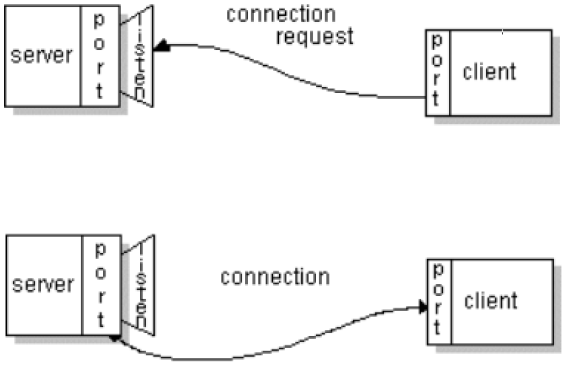
Obrázek 6. IPv4 adresa a IPv6 adresa [6]

3.2.2 Port

Port je číslo identifikující konkrétní službu nebo proces na síťovém zařízení, který umožňuje přijímat a zpracovávat data přes síť. Porty jsou využívány k určení, ke které aplikaci nebo službě mají být doručena příchozí data síťového provozu. Čísla portu jsou v rozsahu od 0 po 65 535. Jsou rozděleny do dvou hlavních kategorií: porty používané serverem a porty používané klientem. Pro porty využívané serverem je rozsah od 0 po 49 151. Pro klientské porty je to od 49 152 po 65 535. [7]

3.2.3 Soket

Klient-serverová komunikace je základním principem komunikace v sítích. Server běží na specifickém počítači a naslouchá na specifickém portu na požadavky klientů. Klient se s ním spojí, čímž se mezi nimi vytvoří spojení. Oba pak můžou komunikovat mezi sebou. Server má jeden soket pro naslouchání a další pro každého klienta. Klient má jeden soket pro komunikaci se serverem. [11]



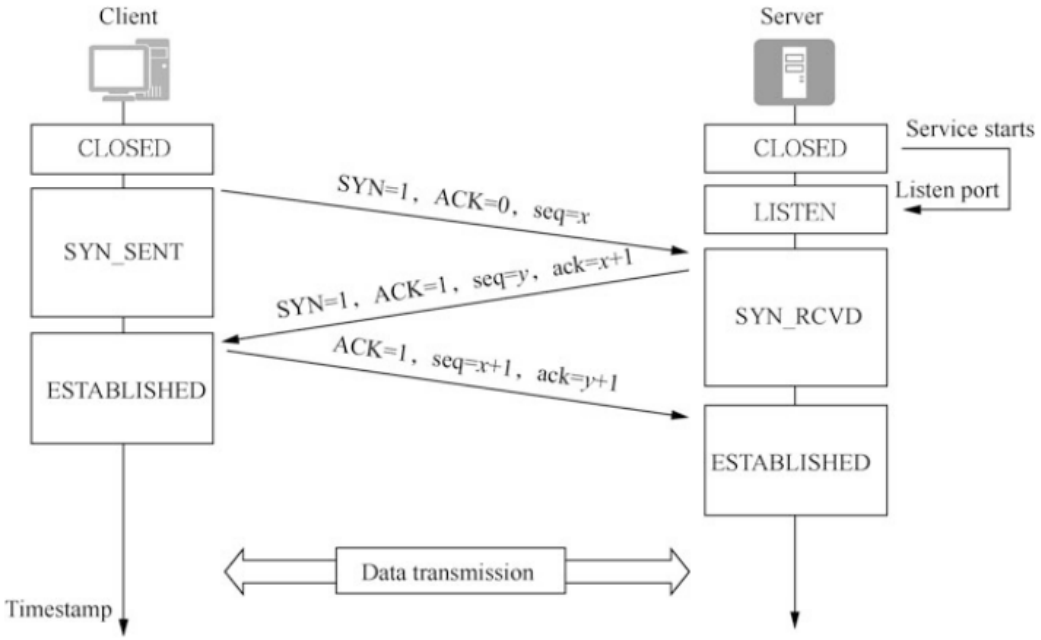
Obrázek 7. Soket – navázání spojení mezi klientem a serverem [11]

3.3 Transportní protokoly

Transportní vrstva umožňuje komunikaci mezi aplikacemi na různých počítačích. V TCP/IP model obsahuje dva hlavní protokoly a to TCP a UDP. Obsahem této časti bude jejich stručný popis.

3.3.1 TCP

TCP je protokol, který funguje na transportní vrstvě v modelu TCP/IP. Jeho hlavním účelem je zajištění spolehlivého a seřazeného přenosu dat mezi aplikacemi na různých síťových zařízeních. Tento protokol před samotným přenosem vždy navazuje spojení. Tento proces se nazývá třícestný handshake. Schéma tohoto procesu je na obrázku č. 8. [7]



Obrázek 8. Třícestný handshake u TCP protokolu [7]

U třícestného handshaku klient začíná tím, že pošle serveru zprávu s žádostí o spojení obsahující příznak SYN. Server odpoví potvrzovací zprávou s příznaky SYN a ACK. Klient poté reaguje potvrzovací zprávou s příznakem ACK a tím je vytvořeno spojení. Tímto způsobem je zajištěno spolehlivé a bezpečné navázání spojení. [7]

3.3.2 UDP

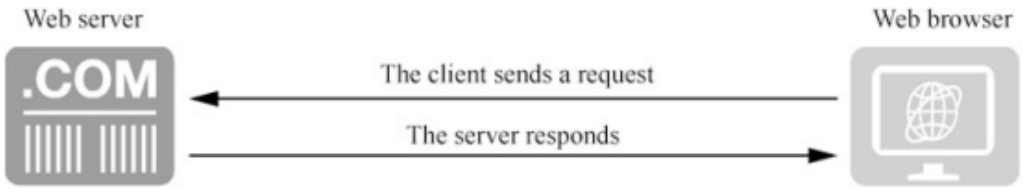
UDP je transportní protokol, který funguje na stejné úrovni jako TCP, ale na rozdíl od něj nenavazuje spojení s komunikujícími protistranou. To znamená, že neprovádí opětovné odesílání ztracených paketů nebo seřazování paketů. [7]

3.4 Aplikační protokoly

Aplikační vrstva je ta, která poskytuje uživatelským aplikacím přímý přístup k síťovým službám a funkcím, které umožňují komunikaci mezi různými zařízeními v síti. Aplikační protokoly, které fungují na této vrstvě, určují, jakým způsobem data budou vyměňována mezi aplikacemi. V této části se zaměříme na popis několika klíčových aplikačních protokolů, které hrají důležitou roli v síťové komunikaci.

3.4.1 HTTP

HTTP je klíčovým protokolem Internetu, který umožňuje komunikaci mezi webovým prohlížečem a webovým serverem. Definuje pravidla pro přenos dat, kdy klient posílá požadavky a server na ně odpovídá. Tyto požadavky a odpovědi obsahují informace jako typ požadavku, adresu požadovaného zdroje a další detaily. Je využíván v celé řadě aplikací a služeb, od webových stránek a API až po streamovací služby. [8]



Obrázek 9. HTTP komunikace s web serverem a webovým prohlížečem [7]

3.4.2 FTP

Je to protokol, který umožňuje přenos souborů mezi dvěma počítači v síti využívající TCP. Typicky se používá pro přenos velkých souborů, skupin souborů nebo stahování softwaru, hudby a dalšího obsahu. [8]

Funguje na principu klient-server. Klient se připojí k FTP serveru a následně může na něj nahrávat nebo z něj stahovat soubory. Klient a server si posílají zprávy pro zahájení, ukončení nebo správu přenosu dat. Je třeba mít na paměti, že FTP se považuje za nebezpečný protokol, protože přenáší přihlašovací údaje a obsah souborů v nešifrované podobě. Proto se doporučuje používat SFTP, který pro zabezpečení přenosu dat využívá šifrování SSL/TLS. [8]

3.4.3 Telnet

Telnet je protokol, který byl dříve hojně využívaný a sloužil ke vzdálenému přístupu k počítačům přes síť. Umožňuje vzdálené připojení k jinému systémům a jeho ovládání pomocí terminálu. V dnešní době je nahrazen bezpečnější alternativou SSH, která poskytuje šifrování. [8]

3.4.4 SSH

Je to protokol umožňující bezpečné vzdálené připojení k počítači a jeho vzdálené ovládání. Primárně slouží správcům serverů pro jejich vzdálenou správu a údržbu, ale umožňuje i zabezpečený přenos souborů a tunelování síťových připojení. Jak už bylo řečeno, jde o bezpečnějšího nástupce protokolu Telnet. Všechna data přenášena po sítí jsou tak šifrována. [8]

3.4.5 SMTP

SMTP je protokol sloužící k odesílání emailů mezi servery. Emailoví klienti jako Gmail nebo Outlook ho využívají k odesílání a mail servery k přijímání a ukládání emailů. Tento protokol se tak stará jen o přesun samotného emailu. [8]

3.4.6 DNS

DNS je nezbytnou součástí Internetové infrastruktury, která překládá doménové jména na IP adresy. DNS funguje jako rozsáhlá databáze, která propojuje tato jména s odpovídajícími IP adresami. Když zadáme doménové jméno do webového prohlížeče, náš počítač odešle dotaz na DNS server. Ten prohledá databázi a vrátí zpět IP adresu serveru, na kterém se nachází požadovaná webová stránka. Prohlížeč pak se serverem na této adrese komunikuje a zobrazí nám požadovaný obsah. [8]

3.5 Bezpečnostní protokoly

Mezi bezpečnostní protokoly patři SSL a TLS. TLS je nástupcem SSL a poskytuje pokročilejší metody šifrování a ověřování. Oba jsou to protokoly zajišťující bezpečné spojení mezi dvěma zařízeními na Internetu nebo lokální síti. Šifrují data přenášená mezi zařízením a serverem, čímž chrání citlivé informace před zneužitím. Protokoly nabízejí šifrování, ověření a kontrolu integrity dat. Pro zabezpečení jsou využívány v prohlížečích, ale i v dalších protokolech jako HTTPS, FTPS a SMTPS. [9]

Webová stránka pro identifikaci odešle prohlížeči digitální certifikát. Po jeho ověření prohlížečem může dojít k přenosu dat. Šifrování probíhá symetrickou kryptografií pro zajištění důvěrnosti a kontrola integrity dat se provádí pomocí kódu Message Authentication Code. [9]

3.6 Směrování v počítačové síti

Tato část kapitoly se zabývá nejzákladnějšími principy směrování v počítačových sítích. Budou popsány základní možnosti směrování a vybrané směrovací protokoly, které se v sítích využívají.

3.6.1 Typy směrování

V počítačových sítí máme různé typy směrování. Nyní si popíšeme dvě nejzákladnější a nejpodstatnější metody.

1. **Statické –** Statické směrování je jednoduchou metodou pro směrování paketů v počítačových sítích. Je vhodné pro propojení malého počtu sítí, kde jsou cesty k cílovým sítím jednoznačné a nemění se v čase. Směrovací tabulky jsou nastavovány administrátorem ručně na každém směrovači. [10]
2. **Dynamické –** Dynamické směrování je pokročilou metodou pro směrování v rozsáhlých sítích, ve kterých jsou předpokládány častější změny její topologie. Oproti statickému směrování, kde je směrovací tabulka nastavována manuálně, si směrovače v dynamickém směrování tabulky sestavují a aktualizují automaticky. [10]

3.6.2 Protokoly pro směrování

V počítačových sítích máme celou řadu nejrůznější protokolů pro dynamické směrování. Protokoly se dělí na dvě skupiny. Tou první jsou interní směrovací protokoly, které

se používají pro směrování uvnitř autonomních systémů. Druhou skupinou jsou externí, které se využívají pro směrování mezi autonomními systémy. [10]

Nyní v jednoduchosti popíšeme dva vybrané směrovací protokoly. Jeden protokol bude ze skupiny interních směrovacích protokolů a druhý bude ze skupiny externích.

1. **RIP –** RIP je jedním z nejdéle používaných interních směrovacích protokolů. Jako metriku používá počet skoků, přičemž každý router představuje jeden skok. Maximální počet skoků je 15, což limituje použití RIPu v rozsáhlých sítích. Služební zprávy se vyměňují pouze mezi sousedními routery a obsahují seznam až 25 sítí s jejich vzdálenostmi od vysílajícího routeru. [10]
2. **BGP –** Je to jediný protokol používaný pro externí směrování v síti Internetu. Tento protokol zajišťuje určování cesty mezi autonomními systémy. Jedná se o extrémně komplexní protokol a velmi důležitou součást Internetu. BGP zajišťuje propojení autonomních systémů a sdílení informací o dostupných sítích. Tím umožňuje najít cestu k libovolné síti na Internetu. [10]

4 PARALELNÍ PROGRAMOVÁNÍ

Kapitola paralelní programování obsahuje základní popis nejdůležitějších termínů v oblasti paralelního programování. Paralelní programovaní je v mnoha aplikacích velmi důležité a umožňuje v jeden okamžik vykonávání více úloh. Nejprve bude zmíněno jaký význam má paralelní programování pro síťové aplikace a poté už bude následovat popis nejdůležitějších termínů z oblasti paralelního programování.

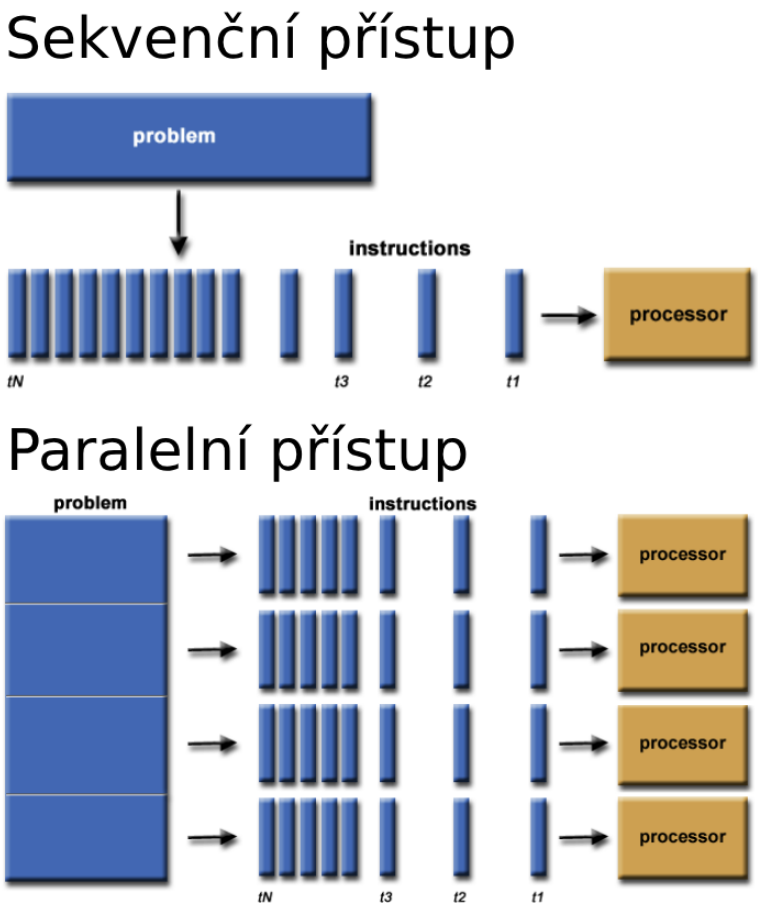
4.1 Význam pro síťové aplikace

Jak už bylo řečeno v úvodu této kapitoly, paralelizace je ve většině aplikací velmi důležitá a u síťových aplikací není tomu jinak. Obecně nám paralelní programování umožňuje rozdělit složitější úlohu na jednodušší části, které pak lze zpracovávat souběžně v jeden okamžik a tím tak dosáhnou vyšší efektivity. [23]

V síťových aplikacích ve většině případech potřebujeme také vykonávat více úloh v jeden okamžik a pokud možno co nejrychleji je to možné. Ať už se jedná o klientskou aplikaci, která vyžaduje současně zpracovávat příchozí a odchozí síťové komunikace nebo jde o složitější serverové aplikace. Serverové aplikace musí umožňovat souběžné zpracovaní velkého množství požadavků od mnoha klientů. Je tedy efektivní a nezbytné, aby tyto úlohy prováděné na serveru byly paralelizované.

4.2 Paralelní a sekvenční přístup

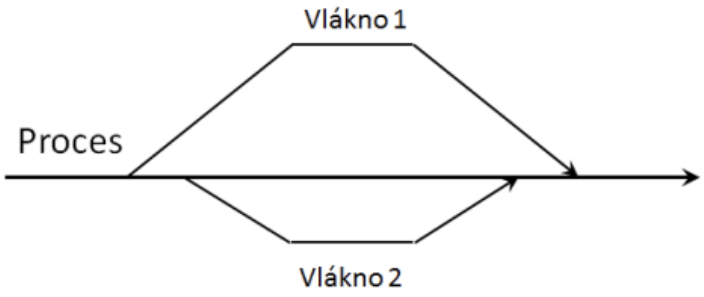
Existuji dva přístupy programování aplikací. Tím prvním a jednodušším na implementaci je sekvenční přístup. Jde o seznam instrukcí, které jsou jednotlivě vykonávány v určitém pořadí. Velikou nevýhodou toho přístupu je však jeho nedostatečná rychlost výpočtu. Z tohoto důvodu je někdy využíváno paralelního přístupu, který efektivně využívá více výpočetních jednotek a tím dosahuje rychlejšího výpočtu. Tento přistup je však náročnější na implementaci a vyžaduje synchronizaci a komunikaci mezi jeho vlákny. [23]



Obrázek 10. Sekvenční a paralelní průstup [25]

4.3 Proces a vlákno

Když uživatel spustí aplikaci je vytvořen proces, který má přístup k potřebným datům, se kterými pracuje. V rámci tohoto procesu existují vlákna. Každý proces má minimálně jedno vlákno a má možnost vytvářet další. Každé vlákno má přístup k datům procesu a může být vykonáváno na výpočetní jednotce samostatně a zajistit paralelizaci úlohy, která je procesem vykonávána. [23]



Obrázek 11. Proces se dvěma vlákny [23]

4.4 Synchronizace a sdílení zdrojů

Při paralelním vykonávání programu často určité sekce kódu přistupují ke kritickým sekcím. Těmito sekcemi je myšlena určitá část kódu v programu, ve které je určitým způsobem přistupováno k libovolným datům, které jsou následně modifikovány. Přístup do těchto kritických sekcí kódu musí být řízen určitými synchronizačními pravidly, aby nedošlo k porušení jejich konzistence. [24]

Nyní si popíšeme některé z možných způsobů synchronizace sdílených zdrojů.

4.4.1 Mutex

Mutex, nebo jinými slovy vzájemné vyloučení, je jedním z nejjednodušších řešení, jak řídit přístup do kritické sekce. Při použití mutexu má do kritické sekce přístup pouze jeden proces. Před samotným vstupem do sekce musí nejdříve získat zámek mutexu. Následně započne s prováděním kódu v kritické sekci. Po dokončení uvolňuje zámek a tak umožňuje přístup jiným procesům do této kritické sekce. [24]

Implementace mutexu obsahuje dvě hlavní metody acquire() a release(). Metoda acquire umožňuje získat zámek a metoda release tento zámek uvolňuje. Zámek je v mutexu reprezentován logickou proměnnou, typicky boolean. Pokud při volaní metody acquire() zámek mutexu není dostupný, je proces zablokován do té doby, dokud není zámek uvolněn. [24]

4.4.2 Semafor

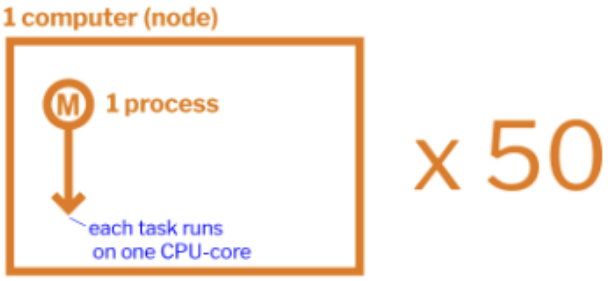
Je to další synchronizační nástroj, který je reprezentován celočíselnou proměnnou. Operace s touto proměnnou jsou vždy prováděny atomicky pomocí funkcí wait() a signal(). Semafor podobně jako mutex, řídí přístup do kritické sekce, ale narozdíl od mutexu umožňuje vstup definovanému počtu procesů. Je tedy možné specifikovat maximální množství procesů, které se v dané kritické sekci v jeden okamžik může nacházet. [24]

4.5 Základní přístupy k paralelnímu programování

Existuje mnoho přístupů a možností paralelizace. V této části budou popsány čtyři základní možné přístupy k paralelnímu programování.

4.5.1 Triviálně paralelní přístup

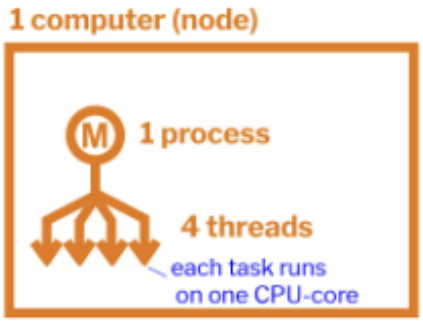
Jedná se o nejtriviálnější typ paralelizmu. Jde o takový způsob paralelizace, kdy každá úloha může být spuštěna zcela nezávisle na ostatních úlohách. Tento způsob je velice snadný na implementaci, jelikož zde není žádná komunikace mezi procesy. [25]



Obrázek 12. Trapně paralelní přístup k programování [25]

4.5.2 Paralelismus se sdílenou pamětí

Jde o způsob paralelizace, kdy každá úloha má přístup ke sdílené paměti a všechny jsou vykonávány na jednom počítači. U tohoto způsobu paralelizace je vyžadovaná komunikace mezi procesy. Program vždy na začátku začíná s jedním vláknem a následně se rozvětvuje na více vláken. Ty jsou po dokončení práce opět spojeny do jednoho vlákna. Tento přístup se nazývá fork/join. Nejběžněji je pro tyto potřeby využívaná knihovna OpenMP. [25]



Obrázek 13. Paralelismus se sdílenou pamětí [25]

4.5.3 Paralelismus s distribuovanou pamětí

U tohoto přístupu je současně vykonáváno několik procesů, které nesdílí stejnou paměť. Tento přístup je často nazýván jako multiprocessing. Úlohy jsou u tohoto přístupu spouštěny na různých počítačích a mají tak vlastní adresní prostor. Jde o jeden ze složitějších způsobů paralelizace, jelikož je vyžadována složitější úroveň komunikace mezi procesy. [25]

Obrázek 14. Paralelismus s distribuovanou pamětí [25]

4.5.4 Paralelismus s využitím akcelerátorů

Při tomto způsobu paralelizace je využíváno specializovaných typů hardwaru, jakými jsou například grafické karty nebo programovatelné hradlové pole. Těchto specializovaných hardwarů je využíváno z důvodu urychlení výpočtu, jelikož CPU, není optimalizováno na vykonávání paralelní úloh, které se skládají ze stovky až tisíců vláken. [25]

5 TESTOVÁNÍ SOFTWARU

Kapitola se zaměří na základy testování softwaru. Úvodem kapitoly bude popsán proces testování, jeho životní cyklus a základní terminologie. Následně budou popsány úrovně testování a testovací metody. Jelikož se tato práce zaměřuje na automatizované hodnocení tak budou popsán i principy přípravy automatizovaného testováni na platformě GitLab.

5.1 Proces testování

Cílem testování je v programu odhalit chyby, které dosud nebyly odhaleny. Programy jsou vždy testovány jak s platnými tak i s neplatnými vstupy. Proces testování započíná už ve fázi samotné analýzy požadavku a pokračuje až do poslední fáze údržby samotného softwaru. [26]

Při návrhu požadavků během analýzy provádíme statické testování, abychom ověřili, zda opravdu odpovídá daným požadavkům. V další fázi, kdy už je připraven spustitelný kód aplikace, je využíváno dynamického testování. Dynamické testování využívá různé techniky, které budou blíže popsány v další části této kapitoly. [26]

5.2 Základní terminologie

Nyní si definujeme základní terminologii, která je užívána při testování [26]:

1. **Omyl** – Omyl je při vývoji softwaru způsobený lidský faktorem, jehož důsledkem pak vzniká chyba.
2. **Chyba** – Může se jednat například o chybějící nebo nesprávný příkaz, který je chybou v programu.
3. **Selhání** – K selhání softwarů dochází v důsledku chyby. To se projevuje nefunkčností systému nebo jeho součástí.
4. **Incident** – Ve chvíli kdy dojde k poruše, může, ale nemusí to být zřejmé. Incident je příznak spojený se selháním, který upozorní uživatele na výskyt poruchy.
5. **Testování** – Zabývá se hledáním chyb, selháními a incidenty.
6. **Test case** – Je konkrétní testovací případ, který má své jednoznačné označení a je spojen s konkrétním programem nebo funkcionalitou. Obsahuje sadu vstupů, které jsou použity k provedení testu, a seznam očekávaných výstupů, které by měly být získány po provedení. Jeho hlavním cílem je definovat postup potřebný k ověření správného chování dané funkce programu.

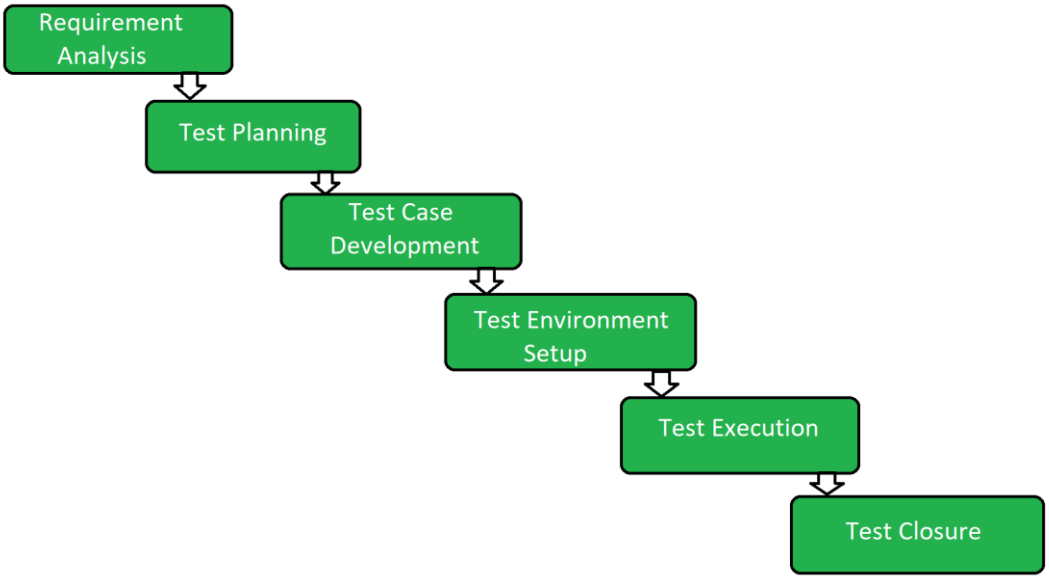
7. **Test suit** – Jde o sadu testovacích případů a skriptů, která se využívá k nalezení nových chyb.
8. **Test script** – Jde o podrobný soubor instrukcí, které mají být postupně provedeny v rámci testovacího případu.
9. **Test ware** – Zahrnuje veškerou dokumentaci k testování. Jde o specifikace testů, testovací případy, testovací data nebo specifické prostředí.
10. **Testovací orákulum** – Jde o libovolný prostředek, který je využíván k předvídání výsledků testů.
11. **Test log** – Jedná se o chronologický záznam všech údajů o provedeném testu.
12. **Test report** – Výstupní dokument, který obsahuje výsledky z testování.

5.3 Životní cyklus testování

Životní cyklus testování softwaru je systematický přístup k testování softwarové aplikace s cílem zajistit, že splňuje určité požadavky a neobsahuje chyby. Jeho hlavním cílem je identifikovat a zdokumentovat jakékoliv chyby nebo problémy. [27]

Životní cyklus testování softwaru zahrnuje tyto fáze [27]:

1. **Analýza požadavků** – Je to první krok v testovacím cyklu. Zaměřuje se na porozumění požadavkům a zajištění jejich správného porozumění.
2. **Plánování testů** – Jsou definovány všechny plány testování a manažer testovacího týmu vypočítá odhadované úsilí a náklady na testování.
3. **Vývoj testovacího případů** – Zahrnuje vývoj testovacích případů, připravují se potřebné testovací data, identifikují se očekávané výsledky pro každý testovací případ a ověřují se jednotlivě testovací případy.
4. **Příprava testovacího prostředí** – V této fázi probíhá nastavení prostředí, ve kterém bude software testován. Nastavení tohoto prostředí ve většině případů provádí vývojář nebo zákazník.
5. **Provedení testů** – V této fázi jsou provedeny všechny připravené testovací případy.
6. **Uzavření testů** – Jde o poslední fázi životního cyklu. V této fázi jsou zdokumentovány všechny činnosti související s testováním. Hlavním cílem je zajistit, aby byly dokončeny všechny činnosti související s testováním.

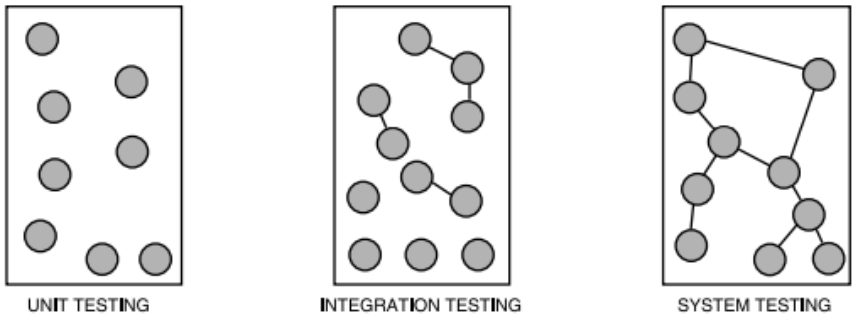


Obrázek 15. Životní cyklus testování softwaru [27]

5.4 Úrovně testování

Při testování softwaru máme tři různé úrovně testování:

1. **Unit testy** – Testování jednotek spočívá v procesu, kdy je modul spuštěn izolovaně od zbytku softwaru a je testován pomocí připravených testovacích případů. Jde o techniku white-box testování. [26]
2. **Integrační testy** – Integrační testování se zaměřuje na interakce mezi komponentami v systému. Zahrnuje testování vzájemné interakce mezi moduly a mezi jinými systémy. [26]
3. **Systémové testy** – Zaměřuje se na úplné testování plně integrovaného systému. Testy se provádějí na vlastnostech, které jsou přítomné pouze tehdy, když je spuštěn celý systém. [26]



Obrázek 16. Úrovně testování [26]

5.5 Metody testování softwaru

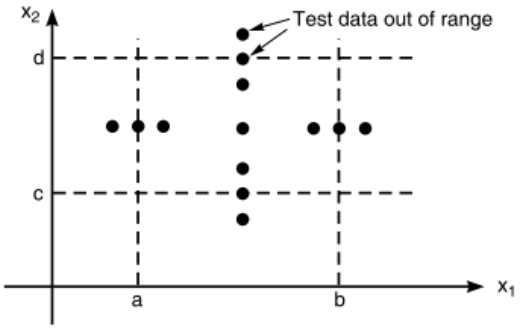
V této části se zaměříme na možnosti dynamického testování softwaru. Dynamické testování ověřuje chování softwaru při jeho běhu a vyhodnocuje jeho výstupy. Existují tři možnosti takového testování, které si nyní popíšeme.

5.5.1 Black Box testování

Black-box testování je metoda testování softwaru, při které je systém testován bez znalosti interního kódu a aplikační logiky. Testuje se pouze na základě specifikace vstupů a očekávaných výstupů. Existuje několik technik, které se při black-box testování využívají. Ty si nyní popíšeme. [26]

5.5.1.1 Analýza hraničních hodnot

Testuje chování softwaru na hranicích mezi platnými a neplatnými vstupy. Je založena na předpokladu, že programátoři nejčastěji dělají chybu při volbě mezi operátory <= a < používaných při rozhodovacích podmínkách. [26]



Obrázek 17. Analýza hraničních hodnot [26]

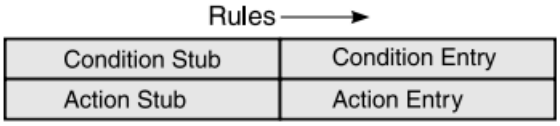
5.5.1.2 Testovaní ekvivalence tříd

U tohoto testování je vstupní a výstupní doména rozdělena na konečný počet tříd ekvivalence. Poté je z každé třídy vybrán jeden zástupce, který je použit při testování softwaru. Hlavním cílem této metody je snížit redundanci mezi testovacími případy. [26]

5.5.1.3 Testovaní založené na rozhodovací tabulce

Tato testovací technika využívá rozhodovacích tabulek. Rozhodovací tabulky jsou přesným a kompaktním způsobem, jak modelovat komplikovanou logiku. Tato technika oproti

ostatním vyžaduje nejvyšší přesnost, jelikož vyžaduje logickou přesnost. Využívají kombinací vstupních podmínek k rozhodování o tom, jaké akce budou provedeny. [26]



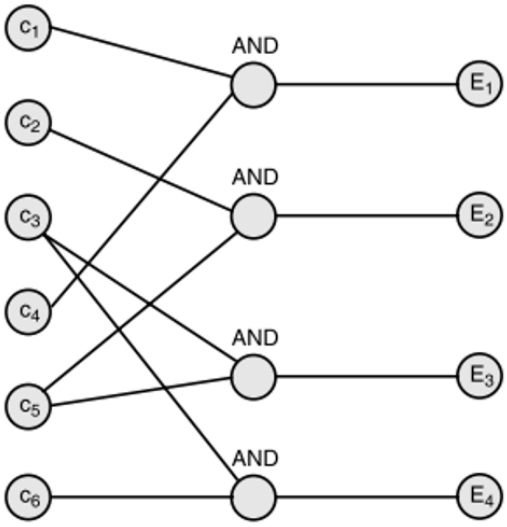
Obrázek 18. Struktura rozhodovací tabulky [26]

5.5.1.4 Technika grafů příčiny a následku

Analyzuje vztahy mezi různými vstupy a výstupy softwaru za účelem vytvoření testovacích případů. Tato metoda logicky spojuje příčinu „vstup“ s následkem „výstup“. [26]

Postup při této technice je následující [26]:

1. U modulu je identifikována vstupní podmínka a akce.
2. Vytvoření grafu příčiny a následku.
3. Transformace grafu na rozhodovací tabulku.
4. Převod pravidel rozhodovací tabulky na testovací případy.

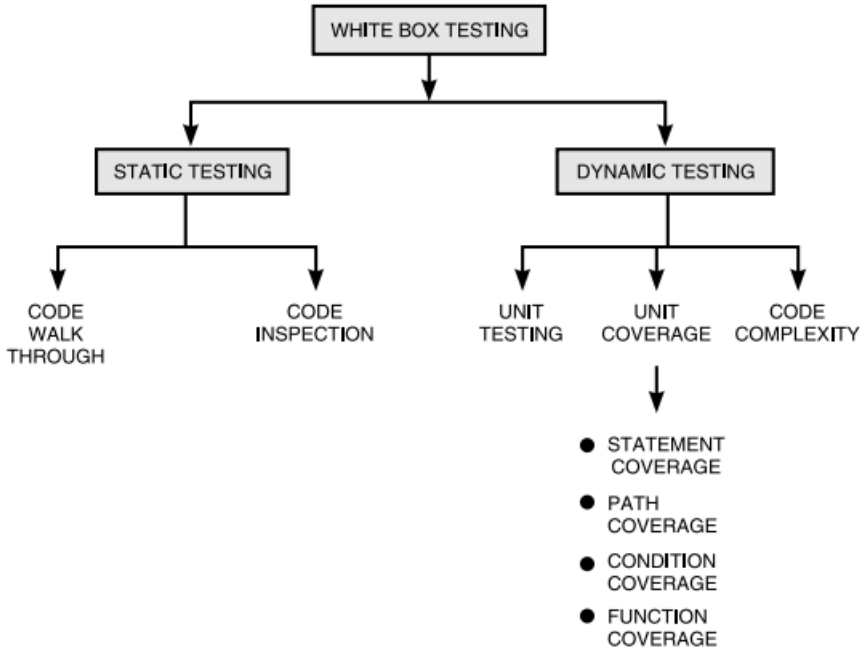


Obrázek 19. Technika grafů příčiny a následků [26]

5.5.2 White Box testování

Jde o další ze způsobu testování softwaru. Oproti metodě black-box tady tato metoda vyžaduje znalost zdrojového kódu testované aplikace. Při této metodě je testována externí funkčnost kód aplikace. Při použití této metody jsou testovací případy navrhovány s využitím vnitřní struktury aplikace. [26]

White-box testovací metoda může zahrnovat jak statické, tak dynamické testování softwaru. Statické testovaní může být prováděno člověkem s využitím specializovaných nástrojů bez nutnosti spuštění aplikace. Naopak u dynamických testů je vyžadováno spustitelných souborů. [26]



Obrázek 20. Klasifikace u White-Box testování [26]

5.5.3 Gray Box testování

Gray-box testování je metoda testování softwaru, která kombinuje prvky dvou již zmiňovaných metod a to white-box a black-box testování. [26]

5.6 Automatizované testy

Automatizované testování automatizuje manuální proces testování. Je používáno k nahrazení nebo doplnění manuálního testování. Manuální testování je nákladné a časově náročné a z toho důvodu se používá automatizované testování. [26]

5.7 Automatizované testování v GitLab

Obsahem této části bude popis a vysvětlení principu základní konfigurace CI/CD pipeliny na platformě GitLab za účelem automatizovaného testování. Bližší popis platformy GitLab se nachází v kapitole 6, kde jsou popisovány nástroje a technologie.

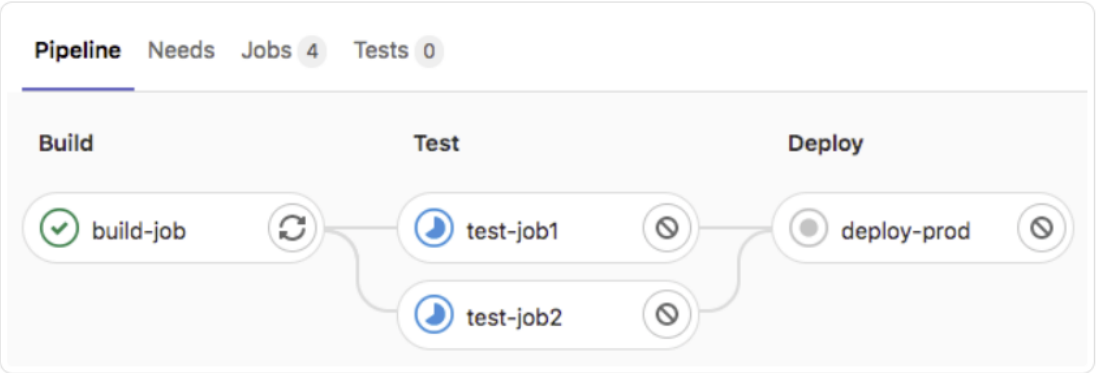
5.7.1 GitLab CI/CD

CI/CD je zkratka pro kontinuální integraci a kontinuální nasazení. Zaměřuje na pravidelné a opakované vytváření, testování, nasazování a sledování změn v kódu. Cílem je minimalizovat riziko vytváření nového kódu na základě jeho chybných nebo neúspěšných předchozích verzí. [29]

5.7.2 Postup konfigurace CI/CD pipeline

Pro úspěšné nakonfigurování pipeline je potřeba provést několik kroků, které budou nyní popsány.

1. **Zajištění dostupnosti runnerů** – V případě, že využíváme GitLab.com, můžeme tento krok přeskočit, jelikož GitLab.com zajišťuje instanci runnerů za nás. Pokud používáme vlastní GitLab server, je nutné nastavit vlastní runnery.
2. **Vytvoření souboru .gitlab-ci.yml** – Tento soubor se nachází v kořenovém adresáři repozitáře a definuje CI/CD úlohy. Je napsán v jazyce YAML a obsahuje různá klíčová slova, která definují jednotlivé kroky pipeline.
3. **Spuštění pipeline** – Je možné ji spustit manuálně nebo automaticky. Manuální spuštění pipeline provedete kliknutím na tlačítko "Run Pipeline" v detailu commitu. Automatické spuštění pipeline se provede po každé změně repozitáře.
4. **Sledování stavu pipeline** – Po spuštění pipeline je možné sledovat její průběh v sekci "Pipelines" na GitLabu. Je možné sledovat stav jednotlivých úloh a případné chyby. Více na obrázku č. 22.



Obrázek 21. Vizualizace pipeline v prostředí GitLab [28]

Na tomto obrázku se nachází ukázkový příklad jednoduché konfigurace pipeline na platformě GitLab.

```
build-job:
  stage: build
  script:
    - echo "Hello, $GITLAB_USER_LOGIN!"

test-job1:
  stage: test
  script:
    - echo "This job tests something"

test-job2:
  stage: test
  script:
    - echo "This job tests something, but takes more time than test-job1."
    - echo "After the echo commands complete, it runs the sleep command for 20 seconds"
    - echo "which simulates a test that runs 20 seconds longer than test-job1"
    - sleep 20

deploy-prod:
  stage: deploy
  script:
    - echo "This job deploys something from the $CI_COMMIT_BRANCH branch."
  environment: production
```

Obrázek 22. Konfigurace pipeline v prostředí GitLab [28]

6 NÁSTROJE A TECHNOLOGIE

Tato kapitola se zaměřuje na nástroje a technologie používané při vypracování praktické části této diplomové práce a také na jejich alternativách, které jsou běžně využívány. Obsahem bude popis vývojových prostředí, systémů pro správu verzí, systému pro kontinuální integrací a nasazení. Následně budou popsány klíčové technologie jako JUnit, Mockito, Selenium, Java, Gradle a další.

6.1 Vývojové prostředí

Výběr správného vývojového prostředí závisí vždy na konkrétních potřebách a preferencích vývojáře. Nyní si popišme tři používané vývojové prostředí, které jsou vhodné pro vyvíjení aplikací v jazyce Java.

6.1.1 IntelliJ IDEA

IntelliJ je placené vývojové prostředí vyvinuté společností JetBrains. Komunitní edice tohoto IDE je zdarma. Toto IDE je známé svými pokročilými nástroji pro analýzu kódu a to ho činí kvalitním nástrojem pro vývoj aplikací v Javě. IntelliJ IDEA má moderní rozhraní a pěkné GUI, ve kterém je jednoduché a příjemné pracovat. Nevýhodou může být nižší výkon na starších počítačích. [1]

Obrázek 23. Logo IntelliJ IDEA [1]

6.1.2 Eclipse

Eclipse je bezplatné a open-source vývojové prostředí, které je vývojářům k dispozici od roku 2001. Je známý svou širokou škálou doplňků a rozšíření, které ho činí vhodnou volbou pro vývoj aplikací v Javě. Eclipse je také vysoce přizpůsobitelný, s uživatelsky přívětivým rozhraním, které umožňuje vývojářům upravit různá nastavení a preference.

Dle názorů některých vývojářů má GUI tohoto IDE rozsáhlé množství funkcí a není příliš přehledné. Není tak úplně vhodnou volbou pro začátečníky. [1]

Obrázek 24. Logo Eclipse [1]

6.1.3 NetBeans

Jde o bezplatné a open-source IDE vyvinuté společností Apache. Je známý svou silnou podporou pro vývoj JavaFX aplikací. NetBeans nemá tak velikou škálu doplňků jako například konkurenční Eclipse nebo IntelliJ. Podle názorů některých vývojářů je NetBeans méně přizpůsobitelné IDE a méně výkonný než jiná vývojová prostředí. [1]

Obrázek 25. Logo NetBeans [1]

6.2 Systémy pro správu verzí

V dnešním světě softwarového vývoje jsou systémy pro správu verzí nezbytností. Ty umožňují vývojářům sledovat změny, porovnávat různé verze kódu a vrátit se k předchozím verzí. Nyní se zaměříme na tři hlavní systémy pro správu verzím.

6.2.1 Git

Git je systém pro správu verzí, který umožňuje vývojářům sledovat změny provedené v kódu jejich projektu. Vznikl v roce 2005 a dnes je nejrozšířenějším systémem pro správu verzí. Původně byl vyvinut pro správu verzí jádra Linuxu. [15]

Mezi jeho klíčové vlastnosti patří:

- Každý klon projektu obsahuje kompletní historii změn.

- Umožňuje vytváření větví, efektivní a paralelní vývoj různých funkcí bez narušení hlavního kódu projektu.
- Je rychlý a efektivní při vývoji velkých projektů.
- Jde o decentralizovaný systém, ke své funkci tak nevyžaduje centrální server.

6.2.2 Mercurial

Mercurial je distribuovaný systém pro správu verzí podobný jako systém Git. Byl vyvinut také v roce 2005. [15]

Mezi jeho klíčové vlastnosti patří:

- Stejně jako Git, umožňuje každému vývojáři pracovat s plnou kopií repozitáře.
- Jedná se o decentralizovaný systém.
- Příkazy a pracovní postupy v systému Mercurial jsou snadnější a intuitivnější ve srovnání se systémem Git.

6.2.3 SVN

Subversion, často nazývaný zkratkou SVN, je systém pro správu verzí, který využívá centrální server k ukládání všech verzí souborů projektu. Jedná se o nástroj, který byl vydán v roce 2000 společností CollabNet Inc. [15]

Meze jeho klíčové vlastnosti patří:

- Všechny verze projektu jsou uloženy na jednom místě na centrálním serveru.
- Atomické commitování zajišťuje, že změny, které jsou do repozitáře commitovány, jsou zaznamenány jako samostatná entita. To usnadňuje sledování a správu chyb.

6.3 Kontinuální integrace a kontinuální nasazení

V této části budou popsání tři systémy pro kontinuální integraci a kontinuální nasazení. Každý z těchto systémů nabízí rozsáhlou sadu funkcí a možností, které umožňují automatizaci a zefektivnění vývojových procesů.

6.3.1 CircleCI

CircleCI usiluje o to stát se špičkou v oblasti platforem pro kontinuální integraci a kontinuální nasazení a usnadnit tak práci vývojářským týmům. Svou spolehlivost už prokázal v praxi u mnoha klientů, od startupů až po nadnárodní korporace jako je PayPal. [16]

Mezi klíčové vlastnosti patří:

- Snadné nastavení
- Konfigurace obsažená v jediném souboru config.yml v repozitáři
- Možnost spouštět úlohy paralelně
- Lze zcela hostovat samostatně
- Nižší možnost přizpůsobení než už GitLab CI/CD a GitHub Actions

6.3.2 GitLab CI/CD

Jde o rozšíření platformy GitLab, které umožňuje vývojářům spouštět CI/CD pipeliny. GitLab využívá model open core, takže jeho základní funkcionality jsou open source. [16]

Mezi klíčové vlastnosti patří:

- Konfigurace obsažená v jediném souboru gitlab-ci.yml v repozitáři
- AutoDevOps, která umožňuje vývojářům spouštět CI/CD pipeliny bez složité konfigurace.
- Podporuje pouze repozitáře GitLab

6.3.3 GitHub Actions

Jedná se CI/CD funkcionalitu v platformě GitHub. Tato platforma je aktuálně nejpoužívanější platformou Git. V roce 2018 byla zakoupena společností Microsoft. [16]

Mezi klíčové vlastnosti patří:

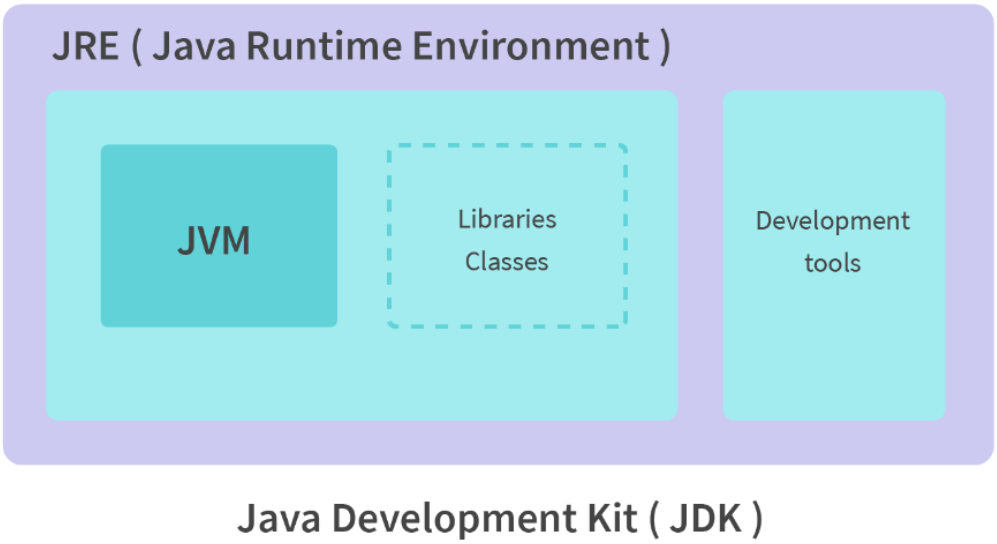
- Nativně integrován s repozitáři
- Nabízí předpřipravené akce pro více jazyků
- Podporuje pouze repozitáře GitHub
- Nejméně vyspělí ze tří zde zmiňovaných

6.4 Java

Java je objektivně orientovaný vysokoúrovňový programovací jazyk. Aplikace napsané v tomto jazyce nelze přímo spouštět na počítači, protože programy psané v programovacích jazycích vyšší úrovně musí být nejprve přeloženy do strojového kódu. K tomu Java využívá kompilátor, který překládá jazyk Java do strojového kódu. Všechny takto zkompilované aplikace jsou spouštěny vždy na JVM. [17]

JVM je interpret, který čte všechny příkazy krok za krokem z bytecode a převádí je do strojového jazyka, aby je počítač mohl vykonat. JVM je zásadním prvkem a bez něj by nebylo možné aplikace spouštět. [17]

JDK je kompletní sada obsahující kompilátor, Java Runtime Environment, ladící nástroje a dokumentaci. Pro vývoj v jazyce Java je nutné mít JDK nainstalovaný na počítači. JRE poskytuje prostředí pro spuštění Java programů a zahrnuje JVM, základní třídy a další podpůrné soubory. [17]



Obrázek 26. Struktura Java Development Kitu [17]

6.5 JUnit

JUnit je framework pro provádění unit testů v jazyce Java. Umožňuje psát unit testy, které ověřují očekávané výsledky a automaticky je spouštět. Poskytuje jednoduché způsoby definování testovacích metod a očekávaných výstupů pro ověření výsledků testů. Díky tomuto frameworku je možné psát kód rychleji a s vyšší kvalitou. Vývojáři mohou jednoduše organizovat své testy a sledovat jejich průběh. [18]

6.6 Mockito

Mockito je knihovna pro mockování v jazyce Java. Lze ho také použít s dalšími frameworky, jako jsou JUnit a TestNG. [19]

Mockování je proces vytváření objektů, které fungují jako falešné nebo klonované verze skutečných objektů. Používá se jako technika testování, kde se namísto skutečných objektů používají falešné objekty. [19]

Existují tři hlavní koncepty mockování: stub, fake a mock. Stub objekty poskytují předdefinovaná data. Fake objekty mají funkční implementace, ale jsou odlišné od produkčních. Mock objekty fungují jako falešné nebo klonované objekty v testování a jsou typicky vytvořeny pomocí knihoven jako je Mockito. Mockování je potřeba, když chceme testovat komponentu, která závisí na jiné komponentě, která ještě není hotová, nebo pokud jsou skutečné komponenty pomalé. [19]

6.7 Selenium

Selenium je sada knihoven a nástrojů, která slouží k automatizovanému testování webových stránek v prohlížeči. Jeho hlavní funkce jsou testování webových stránek a provádění testů přes různé prohlížeče. [20]

Selenium podporuje mnoho prohlížečů, mezi které například patří Google Chrome, Mozzila Firefox, Microsoft Edge, Safari a další. Podporuje také mnoho programovacích jazyků jako je Java, Python, C# a jiné. [20]

Podporuje integraci do CI/CD pipeline pro zajištění automatizovaného testování. Výsledkem automatizovaných testů je podrobný report o provedeném testu, což usnadňuje identifikaci oblastí, které se nechovají podle očekávání a požadavků. [20]

6.8 Gradle

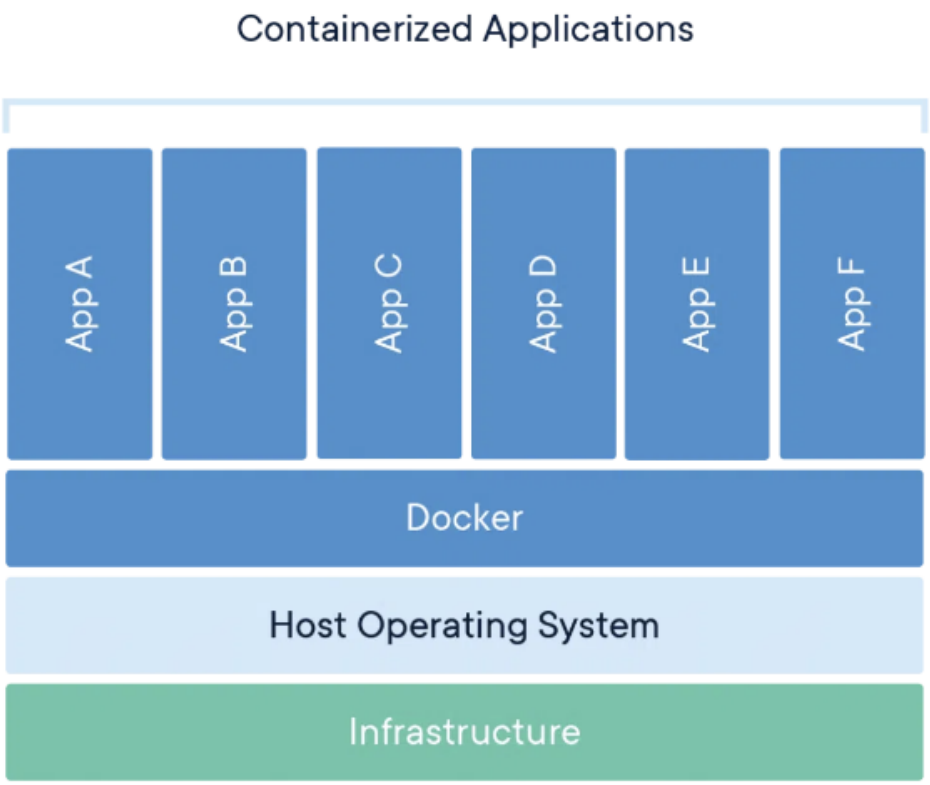
Gradle je efektivní nástroj pro automatizaci sestavení. Nabízí širokou podporu pro různé jazyky jako je Java, Kotlin, C/C++ a mnoho dalších. Gradle pro automatizované sestavení může využívat svých vestavěných funkcí, zásuvných modulů třetích stran, nebo vlastní logiky sestavování. Je rychlý, škálovatelný a dokáže sestavovat projekty libovolné složitosti. [21]

6.9 Docker

Nástroj Docker umožňuje spouštění aplikací v kontejnerech. Kontejner obsahuje kód aplikace, běhové prostředí a další potřebné závislostí. [22]

Docker nabízí odlehčené a standartní kontejnery. Odlehčené kontejnery sdílejí jádro operačního systému a tak nevyžadují vlastní operační systém pro danou aplikaci. Standardní kontejnery naopak obsahují i systém, na kterém aplikace běží, ale naopak nabízejí přenositelnost. Jak u standardního, tak u odlehčeného kontejneru je poskytovaná izolace

od okolního systému a tak i zajištěna bezpečnost. Docker je multiplatformní a je možné jej využívat na různých platformách, jako je Linux, Windows a další. [22]



Obrázek 27. Docker a aplikace běžící v kontejnerech [22]

6.10 YAML

YAML je jazyk pro serializaci dat. Často se používá pro konfigurační soubory a může být použit jako náhrada JSON. YAML je snadno čitelný a také se snadno mapuje na nativní datové struktury. [30]

6.11 REST

REST je architektonický styl, který definuje sadu omezení pro vytváření webových služeb. REST API je způsob, jak přistupovat k vývoji webových služeb jednoduchým a flexibilním způsobem. Je založen na odesílání požadavku z klienta na serveru. [31]

REST poskytuje následující typy požadavků [31]:

1. **GET** – Požadavek určený pro čtení zdroje.
2. **POST** – Požadavek určený pro vytvoření zdroje.
3. **PUT** – Požadavek určený pro aktualizaci zdroje.

4. **DELETE** – Požadavek určený pro odstranění zdroje.

6.12 SOAP

Simple Object Access Protocol je protokol pro komunikaci mezi distribuovanými částmi aplikace. Je flexibilní a umožňuje vytvářet API v různých jazycích. Pro komunikaci využívá XML a podporuje různé komunikační protokoly jako HTTP, SMTP nebo TCP. [32]

SOAP zahrnuje tři základní bloky: Envelope, Header a Body. SOAP požadavky jsou generovány klientem pomocí XML dokumentu a odesílány na server, který odpovídá s požadovanými daty. [32]

Tři základní bloky SOAP [32]:

1. **Envelope –** Obaluje veškerá data ve zprávě a identifikuje XML dokument jako SOAP zprávu.
2. **Header –** obsahuje další informace o SOAP zprávě, jako například autentizační údaje.
3. **Body –** obsahuje podrobnosti skutečné zprávy, která má být odeslána z webové služby.

6.13 MQTT

Message Queuing Telemetry Transport je jednoduchý protokol pro komunikaci IoT zařízení, využívající publish-subscribe model. Jeho výhody zahrnují jednoduchost, spolehlivost, bezpečnost a podporu různých jazyků. [33]

Vysvětlení nejdůležitějších pojmů protokolu MQTT [33]:

- **MQTT klient –** Aplikace nebo zařízení používající MQTT knihovnu. Může se například jednat o aplikaci pro rychlé zasílání zpráv.
- **MQTT broker –** Server zpracovávající připojení, odběry a směrování zpráv.
- **Publish-subscribe pattern –** Tento návrhový vzor se od klient-server liší tím, že odděluje klient odesílajícího zprávy od klienta, který je přijímá. Nevytváří tak přímí spojení. MQTT broker je zodpovědný za směrování a distribuci všech zprav.
- **Topic –** Jedná se o identifikátor pro směrování zpráv.
- **QoS –** Úroveň kvality služby určující spolehlivost doručení zpráv. MQT poskytuje tři úrovně.

7 PROGRAMOVÁNÍ SÍŤOVÝCH APLIKACÍ V JAZYCE JAVA

Obsahem této kapitoly bude stručný popis možností programování aplikací v jazyce Java. Budou zde popsány základní balíčky a základní komponenty, které se využívají při programování síťových aplikací. Obsah těchto částí vychází ze zdroje [12] kde se nachází dokumentace API specifikací Java SE. V další části budou popsány také možnosti vývoje pokročilých aplikací pomocí frameworků nebo knihoven.

7.1 Java balíčky využívané při vývoji síťových aplikací

JDK poskytuje několik různých balíčků, které je možné využít pro vývoj síťových aplikací.

1. **java.net –** Tento balíček poskytuje třídy pro implementaci síťových aplikací.
2. **java.nio –** Balíček definuje vyrovnávací paměti pro uchování dat a poskytuje přehled dalších balíčků, používaných v balíčku NIO.
3. **java.util.concurrent –** Poskytuje nástroje pro paralelní a asynchronní programování. Obsahuje různé třídy a rozhraní pro správu vláken, synchronizaci, plánování úloh a další pokročilé funkce pro práci s více vláknovými aplikacemi.

7.1.1 Základní komponenty pro vývoj síťových aplikací

JDK poskytuje celou řadu tříd a rozhraní, určených pro síťovou komunikaci nebo paralelizaci aplikací. Mezi základní a často využívané komponenty, při vývoji síťových aplikací v jazyce Java patří:

1. **java.net.InetAddress –** Slouží k reprezentaci IP adresy IPv4 nebo IPv6 a názvu hostitele. Poskytuje metody pro práci s IP adresami a hostiteli, jako je získání IP adresy z názvu hostitele a naopak.
2. **java.net.URL –** Třída pro reprezentaci URL adresy. Slouží k reprezentaci adresy zdroje, který je dostupný přes Internet
3. **java.net.Socket –** Objekt reprezentující koncový bod pro komunikaci mezi dvěma zařízeními v počítačové síti. Jedná se o spojení mezi dvěma koncovými body, které umožňuje přenos dat mezi nimi.
4. **java.io.InputStream –** Abstraktní třída pro čtení bytů ze vstupního streamu. U síťových aplikací umožňuje čtení příchozí komunikace soketu.
5. **java.io.OutputStream –** Abstraktní třída pro zápis bytů do výstupního streamu. U síťových aplikací umožňuje zápis odchozí komunikace soketu.

6. **java.net.ServerSocket –** Představuje serverový soket, který naslouchá a čeká na příchozí připojení od klientů. Používá se k vytváření serverových aplikací, které komunikují pomocí TCP/IP protokolu. ServerSocket lze použít k vytvoření serveru, který naslouchá určitému portu na určitém počítači.
7. **java.net.URLConnection –** Abstraktní třída pro otevření spojení pro komunikaci s URL. Pomocí této třídy je možné otevřít spojení s webovou adresou.
8. **java.net.DatagramSocket –** Třída umožňující zasílání a příjem datagramů pomocí UDP protokolu.
9. **java.net.DatagramPacket –** Třída reprezentující datagramový paket pro odesílání nebo přijímání datagramů.
10. **java.lang.Thread –** Třída pro vytváření a správu vláken.
11. **java.util.concurrent.ExecutorService –** ExecutorService je rozhraní, které zjednodušuje správu asynchronních úloh. Poskytuje pool vláken a metody pro odesílání úloh ke spuštění v tomto poolu.
12. **java.util.concurrent.ThreadPoolExecutor –** Jde o implementaci rozhraní ExecutorService, které využívá pool vláken k asynchronnímu spouštění úloh.

7.2 Pokročilé aplikace v jazyku Java

Java poskytuje široké možnosti pro tvorbu pokročilých aplikací a to zejména díky využití frameworků nebo knihoven. Nyní si popíšeme framework Spring a následně Jakarta EE, která je rozšířením Java SE.

7.2.1 Spring

Spring Framework poskytuje komplexní programovací a konfigurační model pro moderní podnikové aplikace založené na Javě. Spring umožňuje jednoduché testování aplikací, přístup k datům, implementaci webových rámců jako je MVC a další. [13]

7.2.2 Jakarta EE

Jakarta EE, neboli Jave EE, je soubor standardů a specifikací určených pro vývoj podnikových a webových aplikací. To je implementováno v knihovnách a aplikačním serveru, který programátoři využívají při tvorbě aplikací. Jakarta EE představuje rozšíření Java SE, poskytující další knihovny a také aplikační server. [14]

Dříve byl Spring považován za konkurenta Java EE, ale s postupným vývojem se tyto frameworky staly spíše doplňkovými. Spring funguje jako knihovny třetích stran, které mohou běžet na JVM nebo jiných aplikačních serverech, zatímco Jakarta EE zahrnuje kompletní aplikační server a další specifické nástroje pro podnikový vývoj. [14]

II. PRAKTICKÁ ČÁST

8 NOVÁ SADA PŘEPRACOVANÝCH ÚLOH

V této kapitole budou popsány nové úlohy pro předmět programování síťových aplikací. Tato sada úloh byla vytvořena z části již stávajících úloh, který byly následně doplněny o další zcela nové úlohy. Pro všechny úlohy bylo napsáno zadání a pokyny pro vypracování. Pro každou úlohu byl vytvořen samostatný repozitář, který bude pro každého studenta dostupný na školním GitLab serveru.

Jako první jsou popsány úlohy, které vychází ze stávající sady úloh a následně budou podrobněji popsány i všechny nové úlohy. Ke konci této kapitoly bude popsán způsob automatizovaného testování a hodnocení těchto úloh.

8.1 Vybrané úlohy

Z aktuální sady úloh, které jsou v předmětu síťových aplikací, jsem vybral ty nejvíce podstatné a důležité. Mezi ně patří Email sender, Telnet klient, Server, Instant Message Server a Web crawler. Tyto úlohy jsou použity pouze s drobnými změnami v zadání. Úlohy byly následně doplněny o další tři nové úlohy, které jsou RESTful API server, SOAP webová služba a nakonec MQTT klient.

8.2 Zadání všech úloh

Každé zadání obsahuje specifikaci požadavků a očekávané výstupy, aby studenti měli jasně dáno, co je od jejich výsledných řešení očekáváno. V této části se bude nacházet jen základní popis jednotlivých úloh. Kompletní zadání všech úloh se nachází v příloze této práce.

8.2.1 Email sender

Úkolem je implementace jednoduché aplikace, která umožní odesílat emailové zprávy pomocí SMTP protokolu. Studenti svá řešení implementují pouze s využitím třídy soketu, která je standardně dostupná v JDK. Tedy jinými slovy řečeno, bez použití jakýchkoli knihoven určených pro jednoduché odesílání emailů. Všechny parametry budou aplikaci předávány pomocí parametru aplikace při jejím spuštění.

8.2.2 Telnet klient

V tomto úkolu studenti implementují telnet klienta, který vytvoří soket pro komunikaci se serverem na specifikované IP adrese a portu. Po navázání spojení bude tento telnet klient odesílat vše, co uživatel napíše na standardní vstup, a na standardní výstup bude vypisovat

všechny odpovědi přicházející od komunikující protistrany v nepozměněné formě. Přijímání a odesílání zpráv musí být zpracováváno v oddělených vláknech. IP adresa a port nutný pro připojení k serveru bude nastaven z parametrů předaných aplikaci při jejím spuštění. Následující úloha je důležitá, jelikož výsledná aplikace telnet klienta bude využívána při následujících úlohách Server a Instant Message Server.

8.2.3 Server

Úkolem je naprogramovat jednoduchý server, který bude schopný přijímat spojení od neomezeného počtu klientů a vykonávat funkci echo. Funkce echo v tomto případě funguje tak, že zpráva odeslaná klientem na serveru bude serverem v identické podobě odeslána zpět tomu stejnému klientovi. Pro každého klienta bude třeba vytvořit jedno vlákno. Ve vlákně pro klienta bude zpráva přijata a následně také i odesílána zpět. Server bude také umožnovat nastavit maximální počet klientů, kteří se budou moct připojit. Jako klient, který se bude připojovat, studenti použijí svoji aplikaci telnet klienta.

8.2.4 Instant Message Server

Tato úloha je výrazně pokročilejší verzí předchozí úlohy. Cílem je implementovat Instant Message server, který umožní uživatelům komunikovat mezi sebou. Server má podobný princip jako předchozí úloha, avšak s přidanými funkcionalitami, které budou dále popsány. Stejně jako i předtím budou studenti používat telnet klienta pro komunikaci s ostatními klienty, připojenými k tomto serveru.

Jednou ze změn oproti předchozímu serveru bude použití dvou vláken pro jednoho klienta. Jedno vlákno pro zpracování příchozích zpráv a jedno vlákno pro zpracování odchozích zpráv. Jelikož klient bude odesílat zprávy všem ostatním klientům, kterých zde bude větší množství, budou určité části aplikace muset být naprogramovány jako thread safe, aby nedošlo ke kolizi přístupu více vláken. Server bude umožnovat nastavit přezdívky klientů, bude možné posílat soukromé zprávy a případně i vstupovat do vybraných diskuzních místností.

8.2.5 Web crawler

Cílem této úlohy je vytvoření web crawleru pro prohledávání webu a identifikaci 20 nejčastěji se vyskytujících slov. Aplikace bude ovladatelná přes terminál a umožní specifikovat parametry pro URL adresu webové stránky, hloubku zanoření a úroveň ladění programu. Parsování HTML stránek bude možné pomocí open-source knihovny JSoup nebo parseru,

který je součástí přímo JDK. Pro efektivní a rychlé zpracování bude aplikace implementována s využitím více vláken. Každá webová stránka bude tak zpracovávána zcela nezávisle v jednom vlákně.

8.2.6 RESTful API Server

V této úloze bude cílem vytvořit RESTful API Server s využitím frameworku Spring Boot, který poskytne službu pro správu uživatelů v databázi. Aplikace bude naslouchat na portu 8080 a bude využívat HTTP připojení. Databázový systém si studenti mohou zvolit dle svých preferencí. Cílem této úlohy je, aby se studenti naučili základy používání frameworku Spring spolu s prací s daty v databází, což je klíčová dovednost ve všech dnešních moderním aplikacích.

Tabulka 1. Endpointy RESTful API serveru

Typ	Popis	URL adresa	Předávané parametry
GET	Získání seznamu uživatelů	/users	N/A
GET	Získání informací o jednom uživateli	/getUser	ID uživatele
POST	Vytvoření nového uživatele	/createUser	Data nového uživatele
PUT	Aktualizace existujícího uživatele	/editUser	ID uživatele, Data aktualizovaného uživatele
DELETE	Smazání uživatele	/deleteUser	ID uživatele
DELETE	Smazání všech uživatelů	/deleteAll	N/A

8.2.7 SOAP webová služba

U této úloze bude cílem implementovat SOAP webovou službu pro správu knih a autorů v systému. Webová služba umožní základní práci s knihami a autory, včetně vytváření, aktualizace a mazání záznamů, stejně jako získávání informací. Aplikace bude ukládat data do relační databáze MySQL a bude naslouchat na portu 8080 s použitím HTTP protokolu.

Implementace SOAP webové služby v jazyce Java bude provedena pomocí frameworku Spring Boot. Pro ukládání knih a autorů v systému bude využita technologie JDBC a relační databáze MySQL. Požadavky a odpovědi této webové služby jsou podrobněji popsány v tabulkách pod tímto textem.

Tabulka 2. Seznam požadavků pro SOAP webovou službu

Název požadavku	**Popis**	**Atributy**	**Typy atributů**
getBookRequest	Získání informací o knize	bookId	long
createBookRequest	Vytvoření nové knihy	book	Book
updateBookRequest	Aktualizace existující knihy	bookId, book	long, Book
deleteBookRequest	Smazání knihy	bookId	long
getAuthorRequest	Získání informací o autorovi	authorId	long
createAuthorRequest	Vytvoření nového autora	author	Author
deleteAuthorRequest	Smazání autora	authorId	long

Tabulka 3. Seznam odpovědí pro SOAP webovou službu

Název odpovědi	**Popis**	**Atributy**	**Typy atributů**
getBookResponse	Odpověď s informacemi o knize	book	Book
createBookResponse	Odpověď s informacemi o vytvořené knize	book	Book
updateBookResponse	Odpověď s informacemi o aktualizované knize	book	Book
deleteBookResponse	Potvrzení smazání knihy	message	string

getAuthorResponse	Odpověď s informacemi o autorovi	author	Author
createAuthorResponse	Odpověď s informacemi o vytvořeném autorovi	author	Author
deleteAuthorResponse	Potvrzení smazání autora	message	string

Pro vyzkoušení SOAP webové služby můžou studenti využít například nástroj cURL. Pomocí tohoto nástroje je možné odeslat požadavek na SOAP webovou službu. Na obrázku č. 28 se nachází ukázka takového požadavku. V tomto případě se jedná o požadavek na vytvoření nového autora v MySQL databázi.

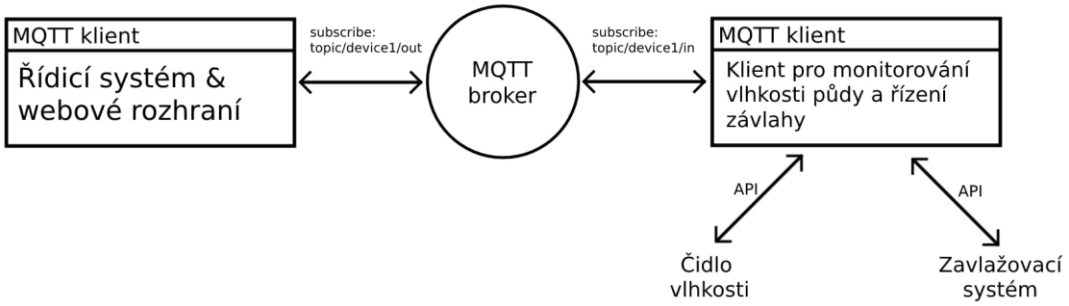
```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:lib="http://example.com/book-web-service">
   <soapenv:Header />
   <soapenv:Body>
      <lib:createAuthorRequest>
         <lib:author>
            <lib:id>1</lib:id>
            <lib:name>Your name</lib:name>
            <lib:surname>Your surname</lib:surname>
         </lib:author>
      </lib:createAuthorRequest>
   </soapenv:Body>
</soapenv:Envelope>
```

Obrázek 28. Požadavek na vytvoření autora u SOAP [vlastní zdroj]

8.2.8 MQTT Klient

Protokol MQTT je často využíván v aplikacích IoT a i proto zadání tohoto úkolu je zaměřeno na aplikaci pro IoT zařízení. V této úloze mají studenti za úkol implementovat MQTT klienta pro monitorování vlhkosti půdy a řízení závlahy v zemědělských prostředích. Klient bude pravidelně odesílat zprávy, obsahující aktuální naměřené hodnoty vlhkosti půdy a přijímat příkazy pro řízení zavlažovacích systémů. Jelikož v rámci tohoto úkolu studenti nemají přístup k takovému reálnému zařízení, bude chování čidla vlhkosti a zavlažovacího systému v programu simulováno. V repozitáři úkolu je definované rozhraní pro přístup k těmto zařízením. Pro tento případ jsou implementace rozhraní nahrazeny jejich simulací. Simulované zařízení neodpovídají v tomto případě realitě a slouží zde hlavně jako náhrada za skutečná zařízení za účelem vývoje.

Klient bude fungovat jako publisher i subscriber v MQTT systému. Bude odebírat topic *"topic/device1/in"* pro přijímání zpráv a odesílat zprávy bude na topic *"topic/device1/out"*. Funkcionalitu tohoto klienta budou studenti moct ověřit s využitím univerzální nástroje pro testování MQTT klientů, který vznikl v rámci této práce jako dodateční nástroj.



Obrázek 29. Schéma systému pro úkol MQTT klient [vlastní zdroj]

8.3 Způsob testování a hodnocení úloh

Všechny vybrané úlohy budou testovány black box technikou. Za tímto účelem v rámci této práce vnikl nástroj, umožňující provádění testů bez závislosti na interní implementaci řešení daných úloh. Důvodem pro volbu této techniky je zajištění výši volnosti při implementování řešení úloh. V neposlední řadě také potřeba zaměřit se primárně na chování aplikace vůči svému okolí, zejména na síťovou komunikaci. A v takové případě je black box technika jedna z možností, která se pro tento typ hodnocení úloh hodí.

Při testování bude sledováno reálné chování aplikace v různých situacích a podmínkách. Bude se tedy hodnotit, zda se aplikace chová vůči svému okolí v oblasti síťové komunikace dle definovaných požadavků.

Každá úloha bude mít definováno své maximum bodů, kterých studenti budou moci dosáhnout. Tento počet bodu se bude odvíjet od počtu úspěšných testovacích případů.

8.4 Postup při vytváření projektů

Tato část se zabývá postupem při vytváření struktury projektu pro dané úlohy a přípravy repozitáře.

Každá úloha je organizována do svého vlastního repozitáře. V každém repozitáři se nachází zadání úlohy v souboru README.md a šablonu Java projektu, připravenou pro implementaci daného úkolu.

Pro vytvoření projektu jsem využil nástroj Gradle. Prvním krokem bylo spuštěním příkazu gradle init, který následně inicializoval nový projekt podle požadavků. Bližší postup se nachází na obrázku č. 30.

```
martin@pop-os ~/Desktop/AutomaticAssessmentOfTasksInNetworkProgramming/Tasks/1_Email_Sender  main gradle init
Starting a Gradle Daemon (subsequent builds will be faster)

Select type of build to generate:
  1: Application
  2: Library
  3: Gradle plugin
  4: Basic (build structure only)
Enter selection (default: Application) [1..4] 1

Select implementation language:
  1: Java
  2: Kotlin
  3: Groovy
  4: Scala
  5: C++
  6: Swift
Enter selection (default: Java) [1..6] 1

Enter target Java version (min: 7, default: 21): 11

Project name (default: 1_Email_Sender): Email Sender

Select application structure:
  1: Single application project
  2: Application and library project
Enter selection (default: Single application project) [1..2] 1

Select build script DSL:
  1: Kotlin
  2: Groovy
Enter selection (default: Kotlin) [1..2] 2

Select test framework:
  1: JUnit 4
  2: TestNG
  3: Spock
  4: JUnit Jupiter
Enter selection (default: JUnit Jupiter) [1..4] 1

Generate build using new APIs and behavior (some features may change in the next minor release)? (default: no) [yes, no]


> Task :init
To learn more about Gradle by exploring our Samples at https://docs.gradle.org/8.7/samples/sample_building_java_applications.html

BUILD SUCCESSFUL in 4m 8s
1 actionable task: 1 executed
```

Obrázek 30. Inicializace Java projektu pomocí nástroje Gradle [vlastní zdroj]

9 TESTOVACÍ SADY PRO JEDNOTLIVÉ ÚLOHY

Každá z vybraných úloh vyžaduje specifické testovací scénáře, aby bylo možné ověřit, zda splňuje požadavky a funguje správně v různých situacích. V této kapitole se zaměříme na testovací scénáře, které byly pro každou z úloh v rámci této práce vytvořeny. Při popisu bude vždy v textu název testovací sady a pod ní budou v bodech popsány její testovací případy.

9.1 Email sender

Testovací scénář u této úlohy se zaměřuje na základní testování funkčnosti odesílání emailů. Obsahuje testovací případy, které ověřují správné odeslání emailu a kontrolu obsahu příchozích emailů.

1. **Testovací sada**: **Základní test odesílání emailů**
    a. **Test odeslání emailu** – Testuje se odeslání jednoho e-mailu. Samotný obsah odeslaného emailu nebude nijak ověřován, pouze se ověří, zda dorazil na server.
    b. **Test obsahu emailu 1** – Testuje se odeslání jednoho e-mailu. Po přijetí emailu je testována správnost i jeho obsah.
    c. **Test obsahu emailu 2** – Jedná se o stejný případ testování jak předchozí jen s jinými odesílanými daty.
    d. **Test odeslání neplatného emailu** – Prostřednictvím testované aplikace je odeslán email se neplatnými údaji. Testuje se, zda email nebyl odeslán.

9.2 Telnet klient

Scénář testování k této úloze je oproti předchozí úloze rozsáhlejší. Výsledná aplikace má více funkcionalit a tak je zapotřebí ověřit všechny z nich. Je rozdělena celkem do tří testovacích sad.

1. **Testovací sada: Testování Telnet klienta**
    a. **Test odeslání zprávy 1** – Testovaný telnet klient odešle na server několik testovacích zpráv a ověří, zda byly tyto zprávy doručeny správně.
    b. **Test odeslání zprávy 2** – Testovaný telnet klient odešle na server mnoho zpráv v krátký okamžik. Bude ověřováno, zda byly doručeny všechny z nich.

c. **Test odeslání zprávy 3** – Bude odesláno několik zpráv, ovšem bude testováno, zda seznam doručených zpráv neobsahuje zprávy, které neměly být odeslány.

2. **Testovací sada: Testování přijímání zpráv**
   a. **Test přijímání zpráv 1** – Virtuální server odešle připojenému telnet klientovi zprávu a bude testováno, zda tuto zprávu obdržel ve správné podobě.
   b. **Test přijímání zpráv 2** – Stejné jak u předchozího případu, ale budou odesílány jiné zprávy.
3. **Testovací sada: Test příkazu pro ukončení**
   a. **Test ukončení** – Na standardní stream testované aplikace telnet klienta bude odeslán příkaz "/QUIT" pro jeho ukončení. Bude ověřeno, zda se aplikace opravdu ukončila.

9.3 Telnet server

Testovaní u této úlohy je velice podobné jak u úlohy telnet klient. Jen namísto klienta je testovat server s funkcí echo. Obsahuje celkem tři testovací sady, které si nyní popíšeme.

1. **Testovací sada: Testování serveru s jedním klientem**
   a. **Test komunikace jednoho klienta 1** – Bude testována komunikace mezi serverem a jedním telnet klientem. Klient odešle jednu zprávu a ověří se, zda testovaný echo server odeslal tu stejnou zprávu zpět.
   b. **Test komunikace jednoho klienta 2** – Opět bude testována komunikace mezi serverem a jedním telnet klientem, jen s jinými parametry.
2. **Testovací sada: Testování serveru s více klienty**
   a. **Test komunikace více klientů 1** – Bude testována komunikace mezi serverem a více telnet klienty. Každý klient odešle jednu zprávu a ověří se, zda testovaný echo server odeslal klientům tu stejnou zprávu zpět.
   b. **Test komunikace více klientů 2** – Bude testována komunikace mezi serverem a více telnet klienty. Každý klient odešle několik zpráv a bude ověřováno, zda odpověděl na všechny z nich.
3. **Testovací sada: Testování omezení počtu připojení k serveru**
   a. **Testování omezení počtu klientů 1** – Server byl spuštěn s parametrem, který určuje, že k serveru se může připojit pouze 5 klientů. K tomuto serveru bylo následně připojeno 6 klientů. U posledního klienta 'client-6' se očekává, že

nebude zpět přijímat zprávy od serveru, protože se nemůže připojit skrz omezení serveru.

9.4 Instant Message server

Při testování této úlohy jde v podstatě také o testování telnet serveru. Rozdíl je ale v tom, že už vykonává pokročilejší činnost, než jen jednoduché odeslání přijaté zprávy zpět stejnému klientovi.

1. **Testovací sada: Testování běžné komunikace**
    a. **Zaklaní test komunikace 1** – Bude testováno, zda doručování zpráv ostatním klientům probíhá správně. Odesílání zpráv probíhá v základní diskuzní místnosti „public“.
    b. **Zaklaní test komunikace 2** – Stejné jako v předchozím případě, jen s jinými odesílanými daty a podmínkami.
    c. **Test odeslání soukromé zprávy** - V tomto testovacím případě bude testováno, zda server umožňuje posílání soukromých zpráv.
2. **Testovací sada: Testování diskuzních místností**
    a. **Test vstoupení do diskuzní místnosti** – V rámci tohoto testovacího případu bude ověřena funkčnost příkazů #join a #groups. Tyto příkazy slouží pro připojení se do diskuzní místnosti a zobrazení místností, ve kterých se uživatel aktuálně nachází.
    b. **Test komunikace v diskuzní místnosti** – Tento testovací případ závisí na výsledku předchozího, kdy se dva klienti připojili k soukromé místnosti. Bude ověřeno, že komunikace probíhá správně a členové, kteří nejsou v této diskuzní místnosti, zprávu neobdrží.
3. **Testovací sada: Testování příkazu pro změnu jména**
    a. **Test změny jména** – Testuje funkčnost příkazu pro změnu jména uživatele. Jméno bude unikátní v rámci IM serveru.
    b. **Test změny neunikátního jména** – Testuje funkčnost příkazu pro změnu jména uživatele. Jméno bude neunikátní v rámci IM serveru a tak se předpokládá, že tato změna uživateli nebude umožněna.

9.5 Web crawler

Testování v této úloze spočívá v tom, že se studentské řešení spustí se stejnými parametry jako interní web crawler modul, který bude součástí testovacího nástroje. Výsledky obou budou porovnány pro všech 20 výstupních slov a také i pro jejich absolutní četnosti výskytu. U testování jejich absolutní četnosti je brána určitá tolerance, jelikož u některých webových stránek, ne vždy musí být jejich obsah úplně identický při znovu načtení.

1. **Testovací sada: Testování Web crawleru pro hloubku 0**
    a. **Test utb.cz s hloubkou 0** – Testuje funkcionalitu web crawleru na webové stránce utb.cz s hloubkou prohledávání 0.
    b. **Test jlcpcb.com s hloubkou 0** – Testuje funkcionalitu web crawleru na webové stránce jlcpcb.com s hloubkou prohledávání 0.
2. **Testovací sada: Testování Web crawleru pro hloubku 1**
    a. **Test wokwi.com s hloubkou 1** – Testuje funkcionalitu web crawleru na webové stránce wokwi.com s hloubkou prohledávání 1.
    b. **Test jlcpcb.com s hloubkou 1** – Testuje funkcionalitu web crawleru na webové stránce jlcpcb.com s hloubkou prohledávání 1.

9.6 RESTful API Server

U této úlohy se testuje funkčnost serverové aplikace, prostřednictvím odesílání a přijímání požadavku od jeho REST API. V rámci testovacího scénáře je postupně prověřena funkce všech end pointů.

1. **Testovací sada: Testování vytváření uživatelů**
    a. **Test vytvoření uživatele** – Ověřuje, zda je možné vytvořit nového uživatele v systému pomocí požadavku odeslaného na REST API.
    b. **Test vytvoření uživatele s neplatným emailem** – Testuje, zda není možné vytvořit uživatele s neplatnou e-mailovou adresou.
    c. **Test vytvoření uživatele s neplatným telefonem** – Testuje, zda není možné vytvořit uživatele s neplatným telefonním číslem.
2. **Testovací sada: Testování aktualizace uživatelů**
    a. **Test aktualizace uživatele** – Ověřuje, zda lze aktualizovat údaje existujícího uživatele v systému.

b. **Test aktualizace uživatele s neplatnými údaji 1** – Testuje, zda nelze aktualizovat uživatele s neplatnou e-mailovou adresou.
c. **Test aktualizace uživatele s neplatnými údaji 2** – Ověřuje, zda nelze aktualizovat uživatele s nevalidním jménem.

3. **Testovací sada: Testování požadavků pro odstranění**
   a. **Test odstranění jednoho uživatele** – Testuje, zda je možné úspěšně odstranit jednoho uživatele ze systému.
   b. **Test odstranění všech uživatelů** – Ověřuje, zda je možné úspěšně odstranit všechny uživatele ze systému.

9.7 SOAP webová služba

Testování u této úlohy je velmi podobné předchozí úloze. Jediný rozdíl je v tom, že se nejedná o REST, ale o SOAP a tak se komunikuje prostřednictvím XML. Aplikace samotná je složitější i po aplikační stránce, kde ukládaná data mají mezi sebou vazbu.

1. **Testovací sada: Testování správy autorů**
   - **Test vytvoření autora** - Vytvoří jednoho autora v systému pomocí odeslaného požadavku a testuje se, zda byl autor úspěšně vytvořen.
   - **Test vytvoření autora s neplatnými údaji** - Pokusí se vytvořit autora se špatně vyplněným jménem a ověří, že tato akce selže.
   - **Test odstranění autora** - Ověří, zda je možné úspěšně odstranit autora ze systému.
2. **Testovací sada: Testování správy knih**
   - **Test vytvoření knihy** – Odešle požadavek na vytvoření jedné knihy v systému a ověří, zda byla úspěšně vytvořena.
   - **Test vytvoření knihy s neplatným údajem** - Zkusí vytvořit knihu s neplatnými údaji a ověřuje se, že tato akce selhala.
   - **Test změny dat knihy** - Mění data již uložené knihy a ověřuje, zda jsou změny úspěšně provedeny.
   - **Test odstranění knihy** - Ověří, zda je možné úspěšně odstranit knihu ze systému pomocí odeslaného požadavku.

9.8 MQTT Klient

Při testování této úlohy se testuje správnost komunikace testovaného MQTT klienta a také jeho chovaní, které bylo definováno v zadání úlohy. Před zahájením samotného testování je nutné mít lokálně spuštěný MQTT broker, který zprostředkovává přenos zpráv mezi klienty v sítí.

1. **Testovací sada: Testování přijímání zpráv vlhkosti půdy**
    a. **Test automaticky odesílaných zpráv** – Ověřuje, zda v definovaných desetisekundových intervalech přicházejí automatické zprávy o naměřené úrovni vlhkosti půdy.
    b. **Test vyžádání odesílání zprávy** – Ověřuje, zda testovaný klient na vyžádání odešle hodnotu vlhkosti půdy.
2. **Testovací sada: Testování řízení závlahy**
    a. **Test získaní stavu zavlažování** – Testuje se, zda testovaný klient reaguje na zprávy pro získání aktuálního stavu zavlažování.
    b. **Test řízení zavlažování** – Ověřuje, zda testovaný klient reaguje na zprávy pro řízení zavlažování.
    c. **Test automatického zastavení zavlažování** – Ověřuje, zda testovaný klient automaticky zastaví zavlažování po 30 sekundách.
3. **Testování chybových zpráv**
    a. **Test odeslání chybové hlášky čidla** – Ověřuje, zda dojde k automatickému odeslání chybové hlášky při poruše čidla vlhkosti.
    b. **Test odeslání chybové hlášky zavlažování** – Ověřuje, zda dojde k automatickému odeslání chybové hlášky při poruše zavlažování.

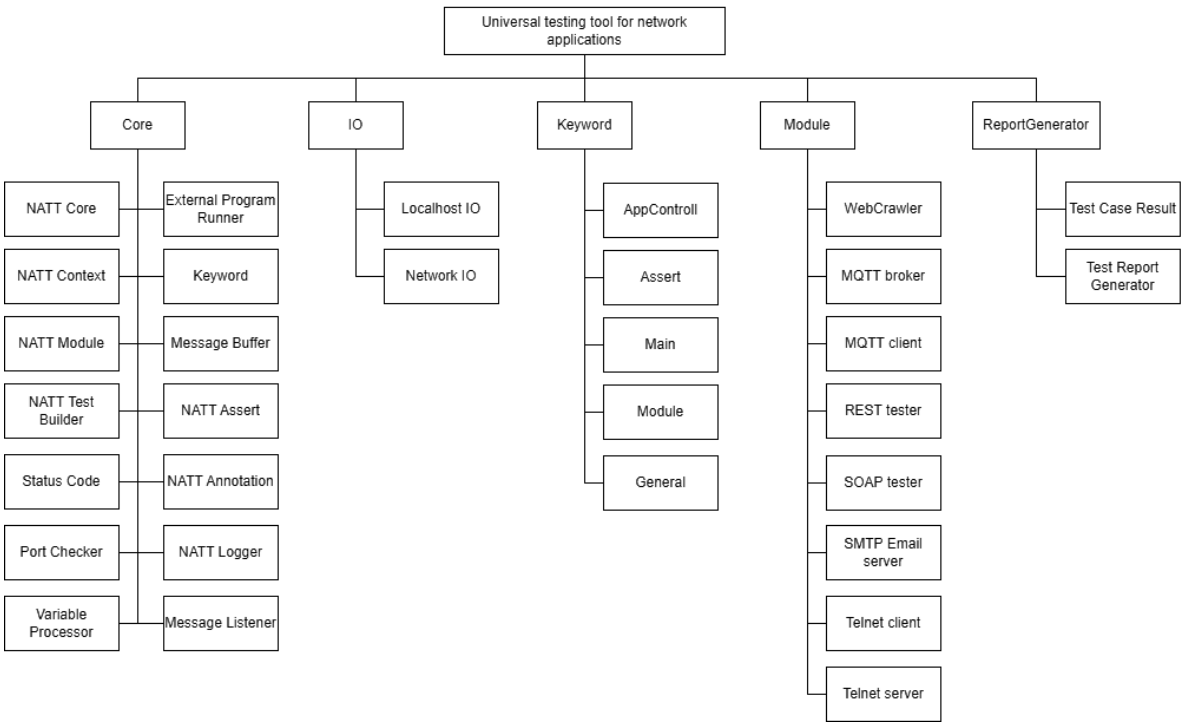
10 POPIS BLACK BOX TESTOVACÍHO NÁSTROJE

Pro automatizaci testování a hodnocení úloh byla vybrána metoda black box testování. Tato metoda klade důraz na testování softwaru bez znalosti jeho vnitřní struktury nebo implementace. Důvody volby black box techniky testování a vytvoření vlastního nástroje na míru daným požadavkům byly následující.

Prvním důvodem je univerzálnost této metody, která umožňuje testování různých typů softwarových aplikací bez ohledu na jejich implementaci. Dalším důležitým faktorem bylo pokud možno co nejvíce oddělit hodnotící aplikaci a její interní logiku od studentských řešení úkolů. Mezi velké výhody nástroje rovněž patří nezávislost na externích síťových prostředcích. Nástroj si je schopen sám vytvářet vlastní virtuální servery a klienty za účelem testování. Nástroj také umožňuje velice jednoduše definovat nové testovací sady. Všechno toto je možné velice jednoduše pomocí specifických klíčových slov v konfiguraci. Konfigurace bude podrobněji popsána v kapitole 12.

10.1 Hlavní struktura nástroje

Zdrojový kód nástroje je rozdělen celkem do pěti hlavních java balíčků. Těmi jsou Core, IO, Keyword, Module a ReportGenerator. Všechny z nich budou blíže popsány v následujících částech této kapitoly.



Obrázek 31. Softwarová struktura nástroje pro hodnocení úloh [vlastní zdroj]

10.2 Core balíček

Tato část nástroje pro hodnocení úloh zahrnuje nejdůležitější část zdrojového kódu, která je zodpovědná za zpracování a vykonávání testovacích scénářů. Obsahuje klíčové prvky pro sestavování testovací struktury z konfigurace, vykonávání testů, vyhodnocování výsledků a generování výstupních reportů. Nyní bude blíže popsán význam jeho jednotlivých komponent.

1. **NATT Core** – Je hlavní komponenta celého nástroje. Zde probíhá načítání konfigurace, sestavení testovací struktury, její vykonání a na konec generování reportů. Pro všechny tyto operace využívá funkcionalitu dalších komponent, které jsou v tomto nástroji definované.
2. **NATT Context** – Slouží k udržování kontextu běhu testu, jako jsou proměnné využívané při testování, nastavení a stav testování. Poskytuje rozhraní pro ukládání a získávání informací o běhu testu.
3. **NATT Module** – Jedná se o abstraktní třídu, od které dědí všechny moduly. Modulem je v rámci tohoto nástroje myšlen síťový prvek, který umožňuje komunikovat s testovanou aplikací. Může se jednat například o klienta nebo server.
4. **NATT Test Builder** – Tato komponenta zajišťuje sestavení testovací struktury z předané konfigurace, která je načítána v YAML formátu.
5. **External Program Runner** – Slouží k spouštění externích programů nebo skriptů jako součást testovacích scénářů. Je primárně určena pro spouštění testované aplikace studenta.
6. **Keyword** – Jde o abstraktní třídu, od které dědí všechny klíčové slova, které tento nástroj poskytuje. Poskytuje univerzální rozhraní pro všechny klíčové slova.
7. **Message Buffer** – Slouží k ukládání přijatých zpráv od všech komunikačních modulů. Každý modul má k dispozici svou vlastní paměť s přijatými zprávami. Při testování je možné s obsahem těchto přijatých zpráv pracovat. Každá zpráva je ukládána i atributem tag. Jedná se o libovolné označení dané zprávy, které usnadňuje její vyhledávání. Hodnota tag závisí na konkrétním modulu, který zprávu přijal.
8. **NATT Assert** – Poskytuje sadu funkcí umožňujících definovat tvrzení v rámci testovacího případu.
9. **NATT Annotation** – Obsahuje definice anotací, které jsou v rámci tohoto nástroje využívány.

10. **NATT Logger** – Poskytuje funkce pro zaznamenávání událostí a informací během testování.
11. **Status Code** – Jsou zde definovány stavové kódy, se kterými může být nástroj ukončen.
12. **Port Checker** – Slouží k ověřování dostupnosti portu na zařízení.
13. **Variable Processor** – Zajišťuje zpracování proměnných a substituci hodnot v testovacích scénářích. Ve všech string atributech umožňuje vložit aktuální hodnotu proměnné.
14. **Message Listener** – Poskytuje mechanismus pro naslouchání a zpracování zpráv při událostech generovaných během testování. Událostí je v tomto případě myšleno přijetí zprávy specifickým komunikačním modulem.

10.3 IO balíček

Tento balíček obsahuje nástroje potřebné pro načítání a ukládání souborů. Je podporováno ukládání a načítání dat ze souboru ve formátu YAML nebo čistě jen běžného textu.

1. **Localhost IO** – Zajišťuje ukládání a načítání dat ze souboru na zařízení, na kterém je nástroj aktuálně spuštěný.
2. **Network IO** – Zajišťuje stejnou funkcionalitu jako Localhost IO, ale umožňuje toto provádět ze vzdálených síťových zařízeních. Je například možné konfiguraci testovacích scénářů načíst ze serveru k tomu určeného a tak případně zabránit manipulaci s testovací sadou ze strany studentů.

10.4 Keyword balíček

Obsahem tohoto balíčku je sada všech klíčových slov, které nástroj podporuje. Bližší popis všech klíčových slov bude v kapitole 12.

1. **AppControll** – Obsahuje sadu klíčových slov, které umožňují řízení testované aplikace.
2. **Assert** – Obsahuje sadu klíčových slov, které umožňují definovat tvrzení.
3. **Main** – Obsahuje sadu klíčových slov, které definují hlavní testovací strukturu. Jde o klíčová slova pro definici testovací sady a testovacích případů.
4. **Module** – Obsahuje sadu klíčových slov, které umožňují vytvářet síťové moduly a pracovat s nimi.

5. **General** – Obsahuje obecnou sadu klíčových slov. Jsou zde klíčové slova pro práci s proměnnými, čekáním, podmíněným čekáním, čtením dat ze souborů a další.

10.5 Module balíček

Balíček obsahuje implementaci všech komunikačních modulů. Tyto moduly je možné vytvářet s definovanou konfigurací, ukončovat, a pokud to daný modul podporuje tak i přijímat a odesílat síťové zprávy.

1. **Web Crawler** – Modul obsahuje implementaci web crawleru. Umožňuje analyzovat obsah procházených webových stránek. V aktuální verzi nástroje nabízí jeden analyzátor obsahu pro hledání a počítání nejčetnějších slov.
2. **MQTT Broker** – Využívá knihovnu Moquette pro vytvoření MQTT Brokeru.
3. **MQTT Client** – Obsahuje implementaci MQTT klienta.
4. **REST Tester** – V tomto modulu se nachází implementace klienta, který s využitím knihovny od Apache umožňuje testovat libovolné REST API. Umožňuje odesílat požadavky GET, POST, PUT, DELETE a přijímat odpovědi serveru. Podporuje nastavování hodnot query parametrů a odesílání dat v těle požadavku.
5. **SOAP Tester** – Obsahuje implementaci testera pro SOAP webovou službu. Umožňuje odesílat požadavky a přijímat odpovědi ve formátu XML. Pro jednodušší práci jsou přijaté odpovědi vždy převáděny do formátu JSON.
6. **SMTP Email Server** – S využitím knihovny Green Mail umožňuje lokálně spustit email server za účelem testování.
7. **Telnet Client** – Tento modul obsahuje implementaci telnet klienta. Umožňuje přijímat a odesílat zprávy.
8. **Telnet Server** – Obsahuje implementaci telnet serveru. Tento server sám o sobě nevykonává žádnou činnost. Jen přijímá zprávy od připojených klientů. Jeho chování tedy už závisí na konfiguraci v testovacím scénáři. Server také umožňuje odesílat zprávy připojeným klientům.

10.6 ReportGenerator balíček

Obsahuje komponenty, které zajišťují správné generování reportu o provedeném testování. Výstupní reporty jsou generovány jako jeden HTML soubor, ve kterém je přehledně zaznamenán celý proces testování.

1. **Test Case Result** – Uchovává výsledek testovacího případu. Umožňuje zaznamenávat vykonávané akce a stav o tom zda uspěli nebo ne.
2. **Test Report Generator** – Umožňuje vygenerovat výsledný report, který obsahuje výsledky a průběh celého testování. Za tímto účelem využívá knihovnu ExtentReports.

11 UKÁZKA IMPLEMENTACE NÁSTROJE

V této kapitole budou představeny vybrané části implementace nástroje pro automatizované hodnocení úloh. Bude popsáno, které knihovny byly použity, část implementace sestavení testovací struktury a vykonávání testů. V další části bude popsán způsob spouštění externích aplikací a ukázky implementace spouštění komunikačních modulů. Na závěr bude předvedeno, jak je celý projekt sestaven a následně spouštěn v rámci GitLab CI/CD pipeline.

11.1 Použité knihovny

Nástroj pro automatizované hodnocení využívá několik knihoven. Ty jsou specifikovány v konfiguračním souboru nástroje Gradle. Nyní budou popsány všechny knihovny, které jsou využívány v rámci tohoto projektu a pro jaké účely zde jsou.

```
dependencies {
    // Use JUnit test framework.
    testImplementation libs.junit

    // This dependency is used by the application.
    implementation libs.guava

    implementation 'commons-cli:commons-cli:1.4'
    implementation 'com.aventstack:extentreports:5.0.5'
    implementation 'org.yaml:snakeyaml:1.29'
    implementation 'com.icegreen:greenmail:1.6.1'
    implementation 'com.sun.mail:javax.mail:1.6.2'
    implementation 'org.apache.httpcomponents:httpclient:4.5.13'
    implementation 'org.jsoup:jsoup:1.14.3'
    implementation 'org.eclipse.paho:org.eclipse.paho.client.mqttv3:1.2.5'
    implementation 'com.fasterxml.jackson.core:jackson-databind:2.13.1'
    implementation 'com.fasterxml.jackson.dataformat:jackson-dataformat-xml:2.13.1'
    implementation 'io.moquette:moquette-broker:0.15'
}
```

Obrázek 32. Použité knihovny [vlastní zdroj]

Seznam všech použitých knihoven:

- **JUnit** – V projektu je využíván pro automatizované testování jeho nejdůležitějších komponent.
- **Guava** – Je to obecná knihovna, poskytující množství pomocných nástrojů. Standardně přidáváno do projektu Gradle.
- **Commons CLI** – Používá se pro zpracování příkazové řádky a argumentů.
- **ExtentReports** – Knihovna pro generování sofistikovaných reportů o provedených testech. Je využívaná pro generování reportů.

- **SnakeYAML** – V projektu je využívána pro zpracování YAML souborů. Využíváno primárně pro zpracování konfigurace.
- **GreenMail** – Knihovna je využívána pro spuštění email serveru za účelem testování studentských aplikací.
- **JavaMail API** – Umožňuje posílat a přijímat emaily. Je využíváno při testování.
- **HttpClient** – Je využíváno pro vytváření a odesílání HTTP požadavků. Toho je využíváno v modulech pro testování REST API a SOAP.
- **JSoup** – Umožňuje analyzovat HTML dokumenty. Tato knihovna je využívána v interním modulu web crawleru.
- **Eclipse Paho** – Tato knihovna je využívána při testování MQTT klientů. Je využívána v modulu, který implementuje MQTT klienta pro testování.
- **Jackson Databind a Jackson Dataformat XML** – Poskytují funkce pro serializaci a deserializaci dat mezi java objekty a JSON/XML. Využíváno primárně u testování REST API a SOAP.
- **Moquette** – Umožňuje spustit jednoduchý MQTT broker, který může být použit při testování MQTT klientů.

11.2 Sestavení testovací struktury

Jednou z prvních velmi důležitých částí v procesu vykonávání testů u tohoto nástroje je sestavení testovací struktury. Ta je sestavována z konfigurace načtené ze souboru, který musí obsahovat data ve formátu YAML.

Ze všeho první je načtena samotná konfigurace a s využitím knihovny SnakeYAML je převedena do objektové podoby. Tyto jsou následně při sestavovaní převedeny do požadované testovací struktury, která se celá skládá pouze z objektů klíčových slov, definovaných v rámci tohoto testovacího nástroje. Výstupem sestavené struktury je její hlavní kořenové klíčové slovo, které obsahuje všechny ostatní klíčové slova testovacího scénáře. Každé klíčové slovo může mít libovolný počet parametrů a dalších klíčových slov. To jaké parametry a potomky bude klíčové slovo mít, závisí na konfiguraci.

Na obrázku č. 33 se nachází ukázka kódu, kde je zahájen proces sestavování. Jde o poměrně rozsáhlý algoritmus, jehož celá implementace se nachází ve volané metodě buildTests(). Sestavování má s využitím JUnit definováno několik testů, které zaručují jeho spolehlivost. Při sestavování je kontrolována neplatná syntaxe, a chybějící parametry.

```java
/**
 * Podle nactene konfigurace sestavi testovaci sadu. Bude sestavena jeji
 * struktura a budou take nacteny vsechny parametry.
 *
 * @throws InvalidSyntaxInConfigurationException
 * @throws NonUniqueModuleNamesException
 * @throws NonUniqueTestNamesException
 * @throws InternalErrorException
 */
public void buildTestsFromYaml()
        throws InvalidSyntaxInConfigurationException, NonUniqueModuleNamesException, NonUniqueTestNamesException,
        InternalErrorException {
    logger.info(message:"Start building test structure according to the configuration ...");

    if (this.configurationData == null) {
        throw new InternalErrorException(message:"Configuration not loaded!");
    }

    // Sestaveni testovaci sady dle konfigurace probehne v externim modulu. Dojde k
    // vytvoreni struktury trid a naplneni data z konfiguracniho souboru
    Keyword root = NATTTestBuilder.buildTests(this.configurationData);

    if (root != null) {
        this.rootKeyword = root;
    } else {
        throw new InternalErrorException(message:"Failed to generate tests. Root keyword is null!");
    }

    // vypis testovaci stuktury na system.out
    this.logger.info(message:"Test structure ...");
    root.printToStructure(offset:0);

    logger.info(message:"Test structure building done");
}
```

Obrázek 33. Metoda pro sestavení testovací struktury [vlastní zdroj]

11.3 Vykonávání testů

Vykonávání testu je další část, která následuje ihned po dokončení sestavování testovací struktury. Před samotným spuštěním je nejprve vygenerován prázdný objekt, do kterého se budou v průběhu testování zapisovat výsledky testů. V dalším kroku je inicializována celá testovací struktura. To spočívá v alokaci potřebné paměti a vytvoření všeho potřebného, co jednotlivé klíčové slova vyžadují pro své vykonání.

Následně se už začíná se samotným vykonáváním. Vykonávání testu začíná hlavním klíčovým slovem. Systém vykonávání spočívá v tom, že je nejprve vykonaná požadovaná funkce daného klíčového slova a pak se dále pokračuje ve vykonávání jeho vnořených klíčových slov.

Každé klíčové slovo může vykonávat libovolné činnosti, závisející na jeho interní implementaci. V některých případech však klíčové slovo provede nějaké akce, které jsou trvalé a mohly by tak ovlivňovat výsledky v dalším průběhu testování. Za tímto účelem ještě každé klíčové slovo po volání jeho hlavní metody execute() musí volat také metodu deleteAction(). Ta tyto trvalé změny, které klíčové slovo provedlo, zase vrátí do původního stavu. Může jít například o ukončení serveru, klienta, nebo odstranění uložené hodnoty

v paměti. Každé klíčové slovo má platnost jen v rámci kontextu, ve kterém bylo vytvořeno. Když kontext, v rámci kterého bylo klíčové slovo vytvořeno, končí, je volána i jeho ukončující akce. Jinými slovy, pokud je určité klíčové slovo definované v rámci testovacího případu, platnost jeho akcí, které byly vykonány, bude platná pouze v rámci tohoto testovacího případu.

```
/**
 * Vykona vsechny keywordy ve vygenerovane testovaci sade. Jako prvni se vsechny
 * keywordy v testovaci sade inicializuji a nasledne se zacnou vykonavat v
 * definovanem poradi. Nakonec se vykonaji ukoncujici akce.
 *
 * @throws TestedAppFailedToRunException
 * @throws InternalErrorException
 * @throws InvalidSyntaxInConfigurationException
 * @throws NonUniqueModuleNamesException
 */
public void executeAllTests()
        throws TestedAppFailedToRunException, InternalErrorException, InvalidSyntaxInConfigurationException,
        NonUniqueModuleNamesException {
    logger.info(message:"Start test executing ...");

    if (this.rootKeyword == null) {
        throw new InternalErrorException(message:"The test_root element is not initialized!");
    }

    // vytvoreni instacne trvaleho modulu pro komonikaci s externe testovanou
    // aplikaci
    this.programRunner = new ExternalProgramRunner();

    // vygenerovani extentu pro generovani reportu testovani
    NATTContext.instance().setReportExtent(TestReportGenerator.generateReportExtent(NATTCore.REPORT_PATH,
            this.testReportName == null ? "Test report" : this.testReportName));

    // clear test caseses results
    NATTContext.instance().getTestCaseResults().clear();

    // inicializace vsech keyword v sestavene testovaci sade
    NATTTestBuilder.initTestScructure(this.rootKeyword);

    // pokud je vyzadovana jen validace konfiguracniho souboru tak zde ukonci
    // testovaci nastroj. Pokud vse az do tohoto kroku probehlo v poradku bez
    // vyvolani vyjimky tak je konfigurace testovaci sady platna z hlediska
    // syntakticke stranky.
    if (this.validateOnly) {
        System.exit(0);
    }

    // zahaji testovani (testovani je zahajeno vykonavanim hlavni keywordy a pak se
    // pokracuje na jejich potomcich)
    logger.info(message:"Starts execution on the root keyword  ...");
    this.rootKeyword.execute();

    // volani ukoncujicich akci
    this.rootKeyword.deleteAction();

    // ukonci runner pokud jeste stale bezi
    this.programRunner.stopExternalProgram();

    logger.info(message:"Test executing finished");
}
```

Obrázek 34. Metoda pro zahájení vykonávání testů [vlastní zdroj]

11.4 Spouštěč externích aplikací

Velmi důležitou částí je spouštěč externích aplikací. Ten může být využit libovolně, ale primárně je určen na spouštění testované aplikace. Umožňuje spouštět aplikaci, ukončovat ji nebo ji jen ukončit a hned znovu spustit. Je možné také spouštěné aplikaci předávat libovolné argumenty. Jediné omezení tohoto interního nástroje je, že umožňuje v jeden okamžik

spustit pouze jednu aplikaci. Ve většině případech není však třeba spouštět jiné aplikace než tu, kterou je třeba testovat.

U spuštěné aplikace je umožněna komunikace. Je možné ji přes standardní stream odesílat potřebná data a také je i od ní přijímat. V případě nečekaného ukončení je oznámena případná chyba, která to způsobila a kód, se kterým aplikace skončila.

11.5 Modul SMTP email serveru

Jde o jeden z komunikačních modulů, který tento black box testovací nástroj nabízí. Jedinou funkcí tohoto modulu je spustit email server, který využívá SMTP protokol a následně přijímat emaily od klientů. Na obrázku pod tímto textem se nachází implementace abstraktní metody modulu, kde se spouští server a zpracovávají příchozích emaily.

```
@Override
public void runModule() throws InternalErrorException {
    if (!PortChecker.isPortAvailable(host:"localhost", this.port)) {
        throw new InternalErrorException(
                String.format("Failed to start SMTP email server because port %d is already taken", this.port));
    }

    smtpServer.start();
    logger.info(super.getNameForLogger() + "SMTP Email Server is listening on port: " + this.port);
    this.isRunning = true;

    // vlakno pro zpracovani prichozich emailu
    Thread emailHandlerThread = new Thread(() -> {
        try {
            while (isRunning) {
                // zpracuje vsechny prijate emaily
                Message[] messages = smtpServer.getReceivedMessages();
                for (Message message : messages) {
                    try {
                        processEmail(message);
                    } catch (IOException e) {
                        logger.warning(super.getNameForLogger() + "Failed to process email: " + e.getMessage());
                    }
                }
                // odstrani prijate emaily z pameti
                if (messages.length > 0) {
                    try {
                        smtpServer.purgeEmailFromAllMailboxes();
                    } catch (FolderException e) {
                        logger.warning(super.getNameForLogger() + "Failed to remove emails from mail box");
                    }
                }
                // uspany vlakna na urcity interval ... pokud dojde email je probuzeno okamzite
                smtpServer.waitForIncomingEmail(timeout:5000, emailCount:1);
            }
        } catch (MessagingException e) {
            logger.info(super.getNameForLogger() + "SMTP Email Server closed. Message: " + e.getMessage());
        } catch (Exception e) {
            logger.warning(super.getNameForLogger() + "Unhandled exception occurred in email handler thread: "
                    + e.getMessage());
        } finally {
            logger.info(super.getNameForLogger() + "Email handler thread terminated.");
        }
    });
    emailHandlerThread.start();
    super.setRunning(isRunning:true);

}
```

Obrázek 35. Spuštění modulu s SMTP email serverem [vlastní zdroj]

V této metodě se jako první spustí server samotný. Následně je vytvořeno jedno vlákno, které se stará o zpracování emailů, přijímaných tímto serverem. V nekonečné smyčce tohoto vlákna jsou jako první zpracovávány příchozí emaily. Je získán jejich obsah a další potřebné údaje. Přijaté data jsou pak vkládaný do message bufferu, se kterými testovací nástroj dále může pracovat. Jelikož všechny emaily stále zůstávají na serveru, tak po jejich zpracovaní jsou hned odstraňovány ze schránky přijatých emailů.

11.6 Modul MQTT brokeru

Princip implementace u tohoto modulu je relativně podobný jako u modulu, který vytváří email server. Jediný rozdíl je v tom, že je spouštěna na serveru jiná služba a to v tomto případě MQTT broker. Modul za tímto účelem využívá knihovnu Moquette a jediná úloha modulu je jeho správné spuštění se specifickými parametry a také možnost jeho ukončení.

11.7 Modul MQTT klienta

Modul umožňuje vytvořit plnohodnotného MQTT klienta. Je možné nastavit, ze kterých topic bude odebírat zprávy a také stanovit libovolnou adresu a port MQTT brokeru. Samozřejmostí je odesílání zpráv na libovolný topic. V metodě na obrázku pod tímto textem se nachází metoda, která tohoto klienta spouští a vše zmíněné nastaví.

```
@Override
public void runModule() throws InternalErrorException {
    try {
        // vytvoreni klienta
        this.mqttClient = new MqttClient(brokerURL == null ? "tcp://localhost:1883" : brokerURL,
                MqttClient.generateClientId());

        // pripojeni
        MqttConnectOptions options = new MqttConnectOptions();
        options.setCleanSession(cleanSession:true);
        this.mqttClient.connect(options);

        // nastaveni callbacku
        this.mqttClient.setCallback(new MQTTCallback(this));

        // subscribed topics
        if (subscribedTopics != null) {
            for (String topic : subscribedTopics) {
                this.mqttClient.subscribe(topic);
            }
        }

        logger.info(super.getNameForLogger() + "MQTT Client module is running.");
        super.setRunning(isRunning:true);

    } catch (MqttException e) {
        logger.warning(super.getNameForLogger() + "Failed to initialize MQTT client: " + e.getMessage());
    }
}
```

Obrázek 36. Spuštění modulu s MQTT klientem [vlastní zdroj]

11.8 Modul telnet klienta

V tomto modulu je obsažena implementace telnet klienta. Jde o jednoduchého klienta, který umožňuje odesílat a přijímat zprávy v nešifrované podobě pouze s využitím třídy Socket, která je standardně k dispozici v JDK.

V metodě na obrázku č. 37 se nachází implementace metody pro jeho spuštění. Jako první je otevřeno spojení s komunikující protistranou. Dále jsou vytvořeny instance readeru a writeru, které umožnují čtení přijatých zpráv a odesílání zpráv. Pro přijímání zpráv je vytvořeno vlákno, které přijaté zprávy vkládá v původní podobě do bufferu zpráv pro další zpracování.

```
@Override
public void runModule() throws InternalErrorException {
    try {
        // navaze spojeni s hostem
        socket = new Socket(host, port);
        reader = new BufferedReader(new InputStreamReader(socket.getInputStream()));
        writer = new PrintWriter(socket.getOutputStream(), true);
        logger.info(super.getNameForLogger() + String.format(
                "Telnet client connected to host with the address: '%s' on port '%d'", host, port));

        // zacne naslouchat prichozim zpravam
        Thread messageListener = new Thread(() -> {
            try {
                String message;
                while ((message = reader.readLine()) != null) {
                    // prijatou zpravu vlozi do bufferu
                    NATTContext.instance().getMessageBuffer().addMessage(this.getName(), tag:"", message);
                    // notifikace o prijate zprave
                    super.notifyMessageListeners(tag:"", message);
                }
            } catch (IOException e) {
                logger.warning(super.getNameForLogger() + String.format("Connection closed"));
            }
            super.setRunning(isRunning:false);
        });
        messageListener.start();
        super.setRunning(isRunning:true);
    } catch (IOException e) {
        logger.warning(super.getNameForLogger() + String.format(
                "Failed to establish a connection with the host '%s' on port '%d'", host, port));
    }
}
```

Obrázek 37. Spuštění modulu s telnet klientem [vlastní zdroj]

11.9 Modul telnet serveru

Modul telnet severu umožňuje vytvořit server, který může naslouchat na libovolném portu, stanoveném při vytváření. Nejdříve je vytvořen serverový soket, který přijímá nové spojení. Pro každé ze spojení je vytvořen handler, který obsluhuje daného klienta. Třída handleru dědí od třídy vlákna a tak je možné spustit zpracování v odděleném vlákně. V tomto vlákně probíhá přijímání zpráv, které jsou po zpracování následně vkládány do bufferu zpráv. Implementace spouštěcí metody se nachází na obrázku č. 38.

```
@Override
public void runModule() throws InternalErrorException {
    if (!PortChecker.isPortAvailable(host:"localhost", this.port)) {
        throw new InternalErrorException(
                String.format("Failed to start Telnet server because port %d is already taken", this.port));
    }

    try {
        // vytvori server soket pro novazovani prichozich spojeni
        serverSocket = new ServerSocket(this.port);
        logger.info(super.getNameForLogger() + "Server socket is listening on port " + this.port);

        // navazovani spojeni s klienty spusti v novem vlakne
        Thread acceptClientsThread = new Thread(() -> {
            try {
                long id = 1;
                while (true) {
                    Socket clientSocket = serverSocket.accept();
                    ClientHandler clientHandler = new ClientHandler(clientSocket, "client-" + id, this);
                    id++;
                    clients.add(clientHandler);
                    clientHandler.start();
                }
            } catch (IOException e) {
                logger.info(super.getNameForLogger() + "Server socket closed");
            }
            super.setRunning(isRunning:false);
        });
        acceptClientsThread.start();
        super.setRunning(isRunning:true);
    } catch (IOException e) {
        throw new InternalErrorException("Failed to start Telnet server: " + e.getMessage());
    }
}
```

Obrázek 38. Spuštění modulu s telnet serverem [vlastní zdroj]

11.10 Modul pro testování REST API

Účelem tohoto modulu je testování REST API. Modul umožňuje vytvořit HTTP klienta, který odesílá požadavky na specifikovaný end point REST API. Tento modul může odesílat požadavky a přijímat odpovědi od serveru. Podporuje vytváření GET, POST, PUT, DELETE požadavků. Při odesílání zprávy podporuje definovat libovolný počet parametrů v query stringu. U POST a PUT požadavků podporuje odesílání dat v těle požadavku.

Na obrázku č. 39 se nachází část kódu, která se stará o vytváření požadavku, který je pak následně odesílán na konkrétní end point. V kódu je switch, který podle nastaveného typu požadavku určuje, jaká metoda bude využita pro generování požadavku. Jsou zde celkem čtyři další metody, každá pro jiný typ požadavku. Všem těmto metodám jsou předávaný parametry odesílané zprávy. Parametry mohou obsahovat parametry query stringu nebo i obsah, který bude vložen do těla požadavku.

Po úspěšném vygenerování požadavku je v další části kódu odeslán a čeká se na jeho odpověď. Maximální doba, kterou může čekat je nastavena na 5000 milisekund.

Pokud je zpráva úspěšně obdržena, je zpracována a její obsah je uložen do message bufferu pro další zpracovaní.

```
// vytvori pozadavek
HttpUriRequest request = null;
switch (this.requestType.toLowerCase()) {
    case "get":
        request = buildGetRequest(this.endpoint, params);
        break;
    case "post":
        try {
            request = buildPostRequest(this.endpoint, params);
        } catch (UnsupportedEncodingException e) {
            logger.warning(super.getNameForLogger() + "Failed to build request: " + e.getMessage());
            return false;
        }
        break;
    case "put":
        try {
            request = buildPutRequest(this.endpoint, params);
        } catch (UnsupportedEncodingException e) {
            logger.warning(super.getNameForLogger() + "Failed to build request: " + e.getMessage());
            return false;
        }
        break;
    case "delete":
        request = buildDeleteRequest(this.endpoint, params);
        break;
    default:
        throw new InternalErrorException("Invalid type of request: " + this.requestType);
}
```

Obrázek 39. Sestavování požadavku pro testování REST API [vlastní zdroj]

11.11 Modul pro testování SOAP

Princip funkce tohoto modulu je podobný jako u modulu, který testuje REST API. Rozdíl je zde jen v tom, že všechny odesílané požadavky jsou typu POST a odesílají na SOAP webovou službu XML obsah, který obsahuje daný požadavek i s jeho daty.

Po sestavení požadavku je odeslán a stejně jako u modulu, který požadavek odesílal pro REST API tak i zde se na odpověď od serveru čeká maximálně 5000 milisekund. Po úspěšném přijetí odpovědí je zpráva z XML formátu převedena na JSON. Do message bufferu je vkládána tedy jen zpráva už ve formátu JSON a to jen data, které se nachází v těle dané odpovědi serveru. Pokud dojde k nějaké chybě na straně serveru, je testovací nástroj o tom informován v podobě http status kódu.

```java
@Override
public boolean sendMessage(String message) throws InternalErrorException {
    if (message == null || message.isEmpty()) {
        return false;
    }

    // vytvoreni pozadavku
    HttpPost httpPost = new HttpPost(url);

    // nastavi SOAP XML jako obsahu pozadavku
    StringEntity entity = new StringEntity(message, ContentType.create(mimeType:"text/xml", charset:"UTF-8"));
    httpPost.setEntity(entity);

    // odeslani pozadavku na SOAP sluzbu
    logger.info(super.getNameForLogger() + "Sending a request on URL: " + this.url);
    try (CloseableHttpResponse response = httpClient.execute(httpPost)) {

        // zpracovani odpovedi
        HttpEntity responseEntity = response.getEntity();
        if (responseEntity != null) {
            int statusCode = response.getStatusLine().getStatusCode();
            if (statusCode >= 400) {
                logger.warning(super.getNameForLogger() + "HTTP request returned an error status: " + statusCode);
                NATTContext.instance().getMessageBuffer().addMessage(getName(), tag:"",
                        SOAPTester.ERROR_CODE_PREFIX + String.valueOf(statusCode));
                super.notifyMessageListeners(tag:"", RESTTester.ERROR_CODE_PREFIX + String.valueOf(statusCode));
                return true;
            }

            try (BufferedReader rd = new BufferedReader(new InputStreamReader(responseEntity.getContent()))) {
                StringBuilder result = new StringBuilder();
                String line;
                while ((line = rd.readLine()) != null) {
                    result.append(line);
                }
                String jsonResponse = convertXmlToJson(result.toString(), taget:"Body");
                NATTContext.instance().getMessageBuffer().addMessage(this.getName(), tag:"", jsonResponse);
                super.notifyMessageListeners(tag:"", jsonResponse);
            }
        }
          You, 48 minutes ago • update
    } catch (IOException e) {
        logger.warning(super.getNameForLogger() + "Failed to send request: " + e.getMessage());
        return false;
    } catch (Exception e) {
        logger.warning(super.getNameForLogger() + "Failed process response: " + e.getMessage());
        return false;
    }

    return true;
}
```

Obrázek 40. Metoda pro odesílání SOAP požadavků [vlastní zdroj]

11.12 JUnit automatizované testy

Součástí vývoje je i testování. Za tímto účelem využívá tento projekt JUnit. V projektu jsou testovány jeho nejdůležitější komponenty, aby ve výsledku bylo dosaženo jeho vyšší spolehlivosti a případné chyby byly odhaleny hned při sestavování projektu.

Pro tento projekt je aktuálně definováno celkem 65 unit testů. Na obrázku č. 41 se nachází jedna z ukázek kódu, kde je možné vidět dva unit testy, které testují funkčnost modulu pro testování REST API. Obdobným způsobem jsou navrženy i zbývající testy pro další komponenty hodnotícího nástroje.

```
You, 1 second ago | 1 author (You)
public class RESTTesterTest {

    private RESTTester restTester;

    @Before
    public void setUp() throws NonUniqueModuleNamesException, InternalErrorException {
        restTester = new RESTTester(name:"http://example.com", endpoint:"TestModule", requestType:"GET", contentType:null);
        restTester.runModule();
    }

    @After
    public void cleanUp() throws InternalErrorException {
        restTester.terminateModule();
    }

    @Test
    public void testBuildGetRequest() {
        String[] params = { "param1=value1", "param2=value2" };
        HttpGet getRequest = restTester.buildGetRequest(endpoint:"http://example.com", params);

        assertNotNull(getRequest);
        assertEquals(expected:"GET", getRequest.getMethod());
        assertEquals(expected:"http://example.com?param1=value1&param2=value2", getRequest.getURI().toString());
    }

    @Test
    public void testBuildPostRequest() throws UnsupportedEncodingException {
        String[] params = { "param1=value1", "param2=value2", "#body={\"data\": \"value\"}" };
        HttpPost postRequest = restTester.buildPostRequest(endpoint:"http://example.com", params);

        assertNotNull(postRequest);
        assertEquals(expected:"POST", postRequest.getMethod());
        assertEquals(expected:"http://example.com?param1=value1&param2=value2", postRequest.getURI().toString());
    }
```

Obrázek 41. Ukázka unit testů pro modul testování REST API [vlastní zdroj]

11.13 Sestavení

Projekt využívá nástroj Gradle. Sestavení je provedeno voláním příkazu, který se nachází na obrázku pod tímto textem. Při volání tohoto příkazu dojde k sestavení celého projektu a následně i provedení všech testů.

```
martin@pop-os ~/Desktop/AutomaticAssessmentOfTasksInNetworkProgramming/NetworkAppTestingTool  main ± ./gradlew build
Path for java installation '/usr/lib/jvm/openjdk-17' (Common Linux Locations) does not contain a java executable
Path for java installation '/usr/lib/jvm/openjdk-11' (Common Linux Locations) does not contain a java executable

BUILD SUCCESSFUL in 17s
12 actionable tasks: 1 executed, 11 up-to-date
```

Obrázek 42. Sestavení projektu pomocí Gradle [vlastní zdroj]

11.14 Konfigurace GitLab CI/CD pipeline

Výsledný nástroj pro hodnocení síťových aplikací musí umožňovat vykonávání testů i na GitLab serveru. Za tímto účelem se v daném repozitáři, u kterého je třeba testovat výslednou síťovou aplikaci, vytvořit soubor s názvem „.gitlab-ci.yml“. Ten bude obsahovat konfiguraci CI/CD pipeliny. Ta se nachází na obrázku č. 43. V pipeline jsou definované následující kroky.

1. Specifikuje se Docker image, který se má použít pro běh. V tomto případě jde o gradle:alpine.

2. V dalším kroku je nastavena proměnná prostředí GRADLE_USER_HOME na lokální složku „.gradle“ ve workspace.
3. V následujícím kroku jsou definovány fáze pipeline. Jde konkrétně o fáze build a test.
4. Dalším krokem je sestavení aplikace, která bude testována. Jedná o aplikaci studenta, která je řešení zadaného úkolu.
5. V posledním kroku bude vykonáno testování s využitím nástroje pro automatické hodnocení úloh, který vznikl v rámci této práce. Tento nástroj se musí nacházet v kořenové složce repozitáře.

Tato konfigurace byla přebrána z tohoto zdroje [34]. Následně byla úprava na míru tak, aby umožnovala integraci testovacího nástroje.

.gitlab-ci.yml 474 B

```yaml
image: gradle:alpine

before_script:
  - GRADLE_USER_HOME="$(pwd)/.gradle"
  - export GRADLE_USER_HOME

stages:
  - build
  - test

build_application:
  stage: build
  script:
    - ./gradlew build
  artifacts:
    paths:
      - app/build/libs/*.jar
  allow_failure: false

black_box_testing:
  stage: test
  script:
    - java -jar NATT.jar -c test-config.yaml
  artifacts:
    paths:
      - test_report.html
  dependencies:
    - build_application
  allow_failure: false
```

Obrázek 43. CI/CD pipline pro sestavení a provedení testů [vlastní zdroj]

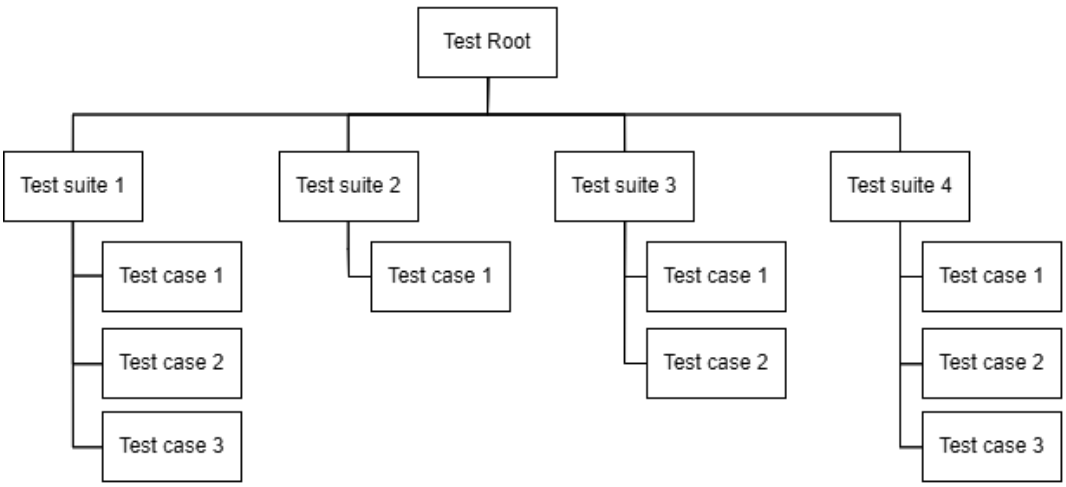
12 ZPŮSOB KONFIGURACE TESTOVACÍHO NÁSTROJE

Aby byla umožněna univerzálnost black box testovacího nástroje, je třeba zajistit způsob, kterým nástroji budou předány potřebné informace o tom, co přesně má vykonávat. Za tímto účelem jsou nástroji předávány konfigurace testovacích sad ve formátu YAML. Z těchto konfigurací nástroj dokáže číst a spolehlivě vykonávat to co je v nich definováno.

Konfigurační jazyk tohoto nástroje se skládá ze sady klíčových slov, které budou podrobně popsány v následující části v této kapitoly. Způsob psaní konfigurace pro tento nástroj je velice podobny jako psaní konfigurací pro GitLab CI/CD pipelny.

12.1 Testovací struktura

Na začátku je nejdříve nutné si popsat, jak jsou testy strukturovány. Na obrázku č. 44 se nachází schéma testovací struktury, který testovací nástroj vyžaduje. Je vidět, že struktura se celá odvíjí od uzlu s názvem Test Root. V každé konfiguraci musí být tento prvek definován hned jako první. Od něj už se dále odvíjejí všechny testovací sady. Test root obsahuje základní nastavení celého testování.



Obrázek 44. Struktura testů v nástroji pro automatické hodnocení [vlastní zdroj]

12.2 Proměnné a přijaté zprávy

Při testování je možné pracovat s obsahem proměnných nebo s bufferem zpráv. Ten obsahuje všechny zprávy, které byly během testování obdrženy. Obsahu bufferu přijatých zpráv je vždy automaticky mazán po dokončení daného testovacího případu. Proměnné jsou vždy dostupné jen v rámci tohoto kontextu, ve kterém byly vytvořeny. Pokud tedy byla

proměnná vytvořena v rámci testovacího případu, bude dostupná pouze v daném testovacím případě. Proměnnou není třeba nijak deklarovat před zápisem do ní. Stačí do ní jen zapsat určitý obsah, a pokud neexistovala, bude automaticky vytvořena.

12.2.1 Vkládání proměnných do řetězců

K obsahu proměnných je možné přistoupit pomocí určitých klíčových slov, které jsou přímo k tomu uzpůsobeny, nebo je jejich hodnotu možné přímo vložit do libovolného parametru libovolného klíčového slova v konfiguraci. Musí se však jednat o parametr, který je typu řetězec. Do daného řetězce stačí jen vložit název dané proměnné s prefixem *‚$‘*. Například následujícím způsobem: *„Text $var1 Text“*.

12.3 Klíčové slova

Obsahem této části bude popis všech klíčových slov, které nástroj umožňuje vykonávat. Z důvodu velikého rozsahu se zde bude nacházet pouze základní popis. Jejich detailní popis a popisem jejich parametru se nachází v dokumentaci.

Každé klíčové slovo má svůj unikátní název a sadu parametrů, které musí být nastaveny. Klíčové slovo může obsahovat libovolný počet parametrů. Parametrem může být string, boolean, double, long nebo list. U list musí být všechny jeho hodnoty stejného typu. Hodnota parametru může také obsahovat další vnořené klíčové slova.

12.3.1 Hlavní klíčové slova

Tato sada obsahuje všechny klíčové slova, které umožnují přímo definovat testovací strukturu.

1. **test_root** – Označuje kořenový prvek konfigurace testu. Musí se nacházet na začátku testovací konfigurace. Od tohoto bodu se začínají vykonávat testy.
2. **test_suite** – Slouží k definici testovací sady.
3. **test_case** – Umožňuje definovat jednotlivé testovací případy.

12.3.2 Klíčové slova pro řízení externí aplikace

Tato sada obsahuje klíčové slova, které umožnují pracovat s externí aplikací. Primárně je určeno pro spouštění a komunikaci s testovanou aplikací.

1. **run_app** – Spustí aplikaci. V jednu chvíli může běžet pouze jedna externí aplikace. Umožňuje definovat argumenty, které jsou ji předány při jejím spuštění.

2. **run_app_later** – Spustí aplikaci s časovým zpožděním. Tato operace je asynchronní. Opět platí, že v jednu chvíli může běžet pouze jedna externí aplikace.
3. **reload_app** – Zastaví momentálně běžící aplikaci a spustí ji znovu.
4. **standard_stream_send** – Odešle zprávu spuštěné aplikaci prostřednictvím standardního proudění.

12.3.3 Obecné klíčové slova

Tato sada zahrnuje klíčové slova pro práci s proměnnými, čekáni, podmíněné čekání a další.

1. **wait** – Pozastaví provádění testu na definovanou dobu.
2. **wait_until** – Čeká, dokud nenastane určitá akce. Akce je vyvolána přijetím zprávy určitým komunikačním modulem. Je možné rozšiřovat o filtrující podmínky. Obsah zprávy, která vyvolala požadovanou akci je automaticky uložena do proměnné pro případné testování.
3. **store_to_var** – Vyhledá a uloží obsah jedné konkrétní zprávy z bufferu zpráv do zvolené proměnné. Pokud je pro specifikované vyhledávací podmínky nalezeno více zpráv, je do proměnné uložena první nalezená. Tedy, ta která byla přijata dřív.
4. **count_and_store** – Počítá počet přijatých zpráv během jednoho testovacího případu a tento počet ukládá do proměnné.
5. **read_file** – Přečte obsah z určeného souboru na lokálním zařízení a jeho hodnotu uloží do definované proměnné.
6. **set_var** – Specifikované proměnné nastaví definovaný obsah.
7. **replace** – Získá obsah specifické proměnné, ve kterém nahradí všechny požadované slova za jejich náhrady. Výsledek je uložen do jiné proměnné.
8. **write_file** – Zapíše definovaný obsah do souboru na lokáním zařízení.
9. **clear_buffer** – Vymaže obsah bufferu zpráv. Je možné vymazat obsah bufferu u všech modulů nebo jen u jednoho specifického.
10. **json_get** – Extrahuje hodnotu jednoho definovaného atributu z obsahu proměnné, který je ve formátu JSON. Je možné přistupovat k indexům pole, nebo i v jednom kroku více do hloubky.
11. **buffer_get** – Získá obsah jedné zprávy z bufferu zpráv, jednoho konkrétního modulu. K zprávě je přistoupeno pomocí indexu a její obsah je uložen do definované proměnné.

12.3.4 Klíčové slova pro definici tvrzení

Tato sada obsahuje klíčové slova, které umožnují definovat tvrzení, které musí být při testování splněno.

1. **assert_string** – Ověřuje, zda proměnná obsahuje očekávaný řetězec.
2. **assert_lower** – Ověřuje, zda numerická proměnná je nižší než očekávaná hodnota.
3. **assert_larger** – Ověřuje, zda numerická proměnná je vyšší než očekávaná hodnota.
4. **assert_equals** – Ověřuje, zda proměnná je rovna očekávanému číslu. Je možné nastavit i určité rozmezí tolerance.
5. **assert_range** – Ověřuje, zda sekvence přijatých zpráv ze dvou modulů jsou ve stanoveném segmentu shodné. Pro její porovnávání je možné definovat i jednoduché porovnávací pravidlo.
6. **assert_app_is_running** – Ověřuje, zda externí aplikace momentálně běží.
7. **assert_module_is_running** – Ověřuje, zda konkrétní modul momentálně běží.
8. **assert_json** – Umožňuje ověřit, zda JSON objekt v proměnné je stejný jako očekávaný JSON objekt.

12.3.5 Klíčové slova pro práci s moduly

Tato sada obsahuje klíčové slova, které umožňují pracovat s komunikačními moduly.

1. **create_telnet_client** – Vytvoří modul, který spustí nové virtuální telnet klienta.
2. **create_telnet_server** – Vytvoří modul, který spustí virtuální telnet server.
3. **create_web_crawler** – Vytvoří modul, který spustí web crawler.
4. **create_email_server** – Vytvoří modul, který spustí virtuální emailový server.
5. **create_rest_tester** – Vytvoří modul, který spustí http klienta pro testování REST API.
6. **create_soap_tester** – Vytvoří modul pro testování SOAP služeb.
7. **create_mqtt_client** – Vytvoří modul, který spustí virtuálního MQTT klienta.
8. **create_mqtt_broker** – Vytvoří modul, který spustí MQTT broker.
9. **termite_module** – Ukončí běžící modul, který již není potřeba.
10. **module_send** – Odesílá zprávu s využitím určitého modulu.
11. **create_filter_action** – Vytváří filtr pro akce, které jsou vyvolávány při přijetí zprávy. Je možné filtrovat textový obsah.
12. **clear_filter_actions** – Odstraní všechny filtry akcí u specifického modulu.

12.4 Popis práce s komunikačními moduly

Každý z modulů má své specifické chováni a způsoby, jak je nutné s nimi pracovat. Obsahem této části bude popis každého z nich a způsoby, jak správně využívat jejich funkcionality při psaní testovací konfigurace.

Po obdržení zprávy libovolným modulem, je její obsah příslušně zpracován a vložen do bufferu zpráv. Ve všech případech je i zároveň obsah této poslední doručené zprávy ukládán do proměnné s následujícím názvem: *„<module-name>-last-msg“*. Použití této proměnné v některých případech výrazně usnadňuje psaní konfigurace.

12.4.1 Spouštěč externích aplikací

Jedná se o modul, který není nutné vytvářet a je vytvořen automaticky sám ještě před samotným zahájením testování. Není možné vytvořit více jeho instancí.

Pro odesílání zpráv je nutné použít klíčové slovo "standard_stream_send' místo standardně používaného 'module_send'.

Zprávy přijaté tímto modulem se vkládají do vyrovnávací paměti zpráv. Tag každé vkládané zprávy je nastavena na prázdnou hodnotu.

12.4.2 Modul telnet klient

Jedná se o modul, který musí být vytvářen před tím, než budou využívány jeho funkcionality při testování. V rámci testovací sady může existovat i více jeho instancí. Toto tvrzení platí i pro všechny ostatní moduly, pokud není řečeno jinak.

Obsah zprávy odesílané pomocí klíčového slova ‚module_send‘ nemusí mít nijak specifický formát. Odešle libovolně předaný textový řetězec.

Zprávy, které tento modul obdrží, jsou vkládaný do bufferu zpráv. Tag každé vkládané zprávy je nastaven na prázdnou hodnotu.

12.4.3 Modul telnet server

U telnet serveru obsah zprávy odesílané pomocí klíčového slova ‚module_send‘ nemusí mít nijak specifický formát. Odešle libovolně předaný textový řetězec všem připojeným klientům.

Všechny zprávy od klientů, které tento modul obdrží, jsou vkládaný do bufferu zpráv. Tag každé vkládané zprávy je nastaven na ID klienta, který serveru tuto zprávu poslal.

ID klienta je řetězec tohoto formátu *„client-#“* kde # představuje číslo klienta. Číslo daného klienta závisí na pořadí, ve kterém byl k serveru připojen. První připojeny klient bude mít ID *„client-1“*, druhý bude mít *„client-2“* a tak dále.

12.4.4 Modul SMTP email server

Tento modul nepodporuje odesílání zpráv pomocí klíčového slova ‚module_send'. Jedná se o modul, který pouze spustí email server a přijímá emaily.

Obsah všech emailů, které tento modul obdrží, je vkládán do bufferu zpráv. Tag každé vkládané zprávy je nastaven na hodnotu předmětu daného emailu.

12.4.5 Modul REST tester

U modulu REST tester obsah zprávy odesílané pomocí klíčového slova ‚module_send' musí mít specifický formát odesílaného obsahu. Text odesílaný tímto modulem může obsahovat více parametrů. Za tímto účelem musí být text ve zprávě v následujícím formátu: *„<name_1>=<value_1>;<name_2>=<value_2>“*.

Kde „name“ je název parametru, který bude vkládán do query stringu. Pokud tento název začíná prefixem *„#“*, bude jeho obsah vložen do těla požadavku.

Všechny odpovědí na požadavky od serveru, které tento modul obdrží, jsou vkládány do bufferu zpráv ve formátu JSON. Tag každé vkládané zprávy je nastaven na adresu endpointu, ze kterého byla odpověď obdržena.

12.4.6 Modul SOAP tester

U modulu REST tester obsah zprávy odesílané pomocí klíčového slova „module_send“ musí mít specifický formát odesílaného obsahu. Odesílaný obsah požadavku musí být ve formátu XML a jeho struktura musí být taková, jakou vyžaduje daná testovaná SOAP webová služba. Požadavek je možné zadat ručně nebo ho načíst přímo z předpřipraveného souboru a načtený obsah pomocí této metody odeslat.

Všechny odpovědí na požadavky od serveru, které tento modul obdrží, jsou vkládány do bufferu zpráv. Každá zpráva před vložením je XML převedena do JSON formátu. Tag každé vkládané zprávy je nastaven na adresu endpointu, ze kterého byla odpověď obdržena.

12.4.7 Modul MQTT klient

U modulu MQTT klienta obsah zprávy odesílané pomocí klíčového slova ‚module_send' musí mít specifický formát odesílaného obsahu. Ve zprávě je nutné specifikovat topic a obsah dané zprávy. Formát je následující: *„ <topic>:<Message>"*.

Zprávy obdržené tímto modulem jsou vkládány do bufferu zpráv v takové podobě, v jaké přišly. U každé zprávy je tag nastaven na hodnotu topic, ze kterého byla zpráva obdržena.

12.4.8 Modul MQTT broker

Modul MQTT broker nepodporuje odesílání zpráv pomocí klíčového slova ‚module_send. Modul pouze spustí MQTT broker a vykonává jeho činnost. Modul neumožňuje ani přijímání příchozí komunikace. Tento modul je pouze zprostředkovatelem MQTT komunikace mezi klienty, kteří jsou k němu připojení.

12.5 Ukázka konfigurace testů

Na další straně na obrázku č. 45 se nachází počáteční část konfigurace pro testování serveru s REST API. Na samotném začátku se spouští Spring serverová aplikace, která bude testována. Je zde definována podmínka, která čeká, dokud se tato Spring aplikace plně nespustí. V dalších krocích přípravy před samotným testováním je odstranění všech uživatelů z databáze pomocí DELETE požadavku. Dále následuje definice první testovací sady, která má taky své inicializační kroky, které budou provedeny před tím, než začne s vykonáváním testovacích případu. Ve spodní části obrázku se nachází definice prvního testovacího případu. Konfigurace samotná je rozsáhlá a pokračuje i dále.

```yaml
test_root:
    max_points: 10.0
    initial_steps:
        # spusti spring rest api server
        - run_app: "java -jar build/libs/RESTAPIServer-0.0.1-SNAPSHOT.jar"
        # ceka nez se server spusti (vypis textu: Started RestapiServerApplication in)
        - create_filter_action:
            name: "app-std-out"
            text: "Started RestapiServerApplication in"
            mode: "contains"
            case_sensitive: false
        - wait_until:
            module_name: "app-std-out"
            time_out: 30000
        # cisteni databaze na zacatku
        - create_rest_tester:
            name: "module-1"
            url: "http://localhost:8080/deleteAll"
            request_type: "DELETE"
        - module_send:
            name: "module-1"
            message: ""
    test_suites:
        - test_suite:
            name: "Testování vytváření uživatelů"
            delay: 500
            initial_steps:
                # vytovreni REST testeru pro POST & GET
                - create_rest_tester:
                    name: "user-create"
                    url: "http://localhost:8080/createUser"
                    request_type: "POST"
                    content_type: "application/json"
                - create_rest_tester:
                    name: "user-get"
                    url: "http://localhost:8080/getUser"
                    request_type: "GET"
            test_cases:
                - test_case:
                    name: "Test vytvoření uživatele"
```

Obrázek 45. Ukázka části konfigurace testovacího scénáře [vlastní zdroj]

13 UKÁZKA VÝSLEDNÉ PRÁCE

V této závěrečné části práce budou předvedeny dosažené výsledky. V této kapitole bude popsáno, co je výstupem práce, ukázka finální verze nástroj pro automatizované hodnocení síťových aplikací, ukázky výstupní report a editor konfigurací.

13.1 Výstup práce

Zadáním této práce bylo zajistit automatizované hodnocení úloh v předmětu síťových aplikací. Dalším úkolem bylo provést rozbor aktuálních úloh a pro vybrané z nich navrhnout jejich testovaní a doplnit je o podrobné zadaní. A nakonec pro všechny tyto úkoly, připravit strukturu projektu pro GitLab repozitář.

Výstupem této práce je:

1. Přepracování a rozšíření sady úloh
2. Podrobné zadání pro každou úlohu napsané v Markdown
3. Vzorové řešení pro každou úlohu – Určeno pouze pro učitele předmětu
4. Připravená struktura projektů pro GitLab repozitáře
5. Nástroj pro automatizované hodnocení síťových aplikací
6. Konfigurace obsahující testovací sady pro všechny úlohy
7. Editor konfigurací
8. Dokumentace nástroje v Markdown

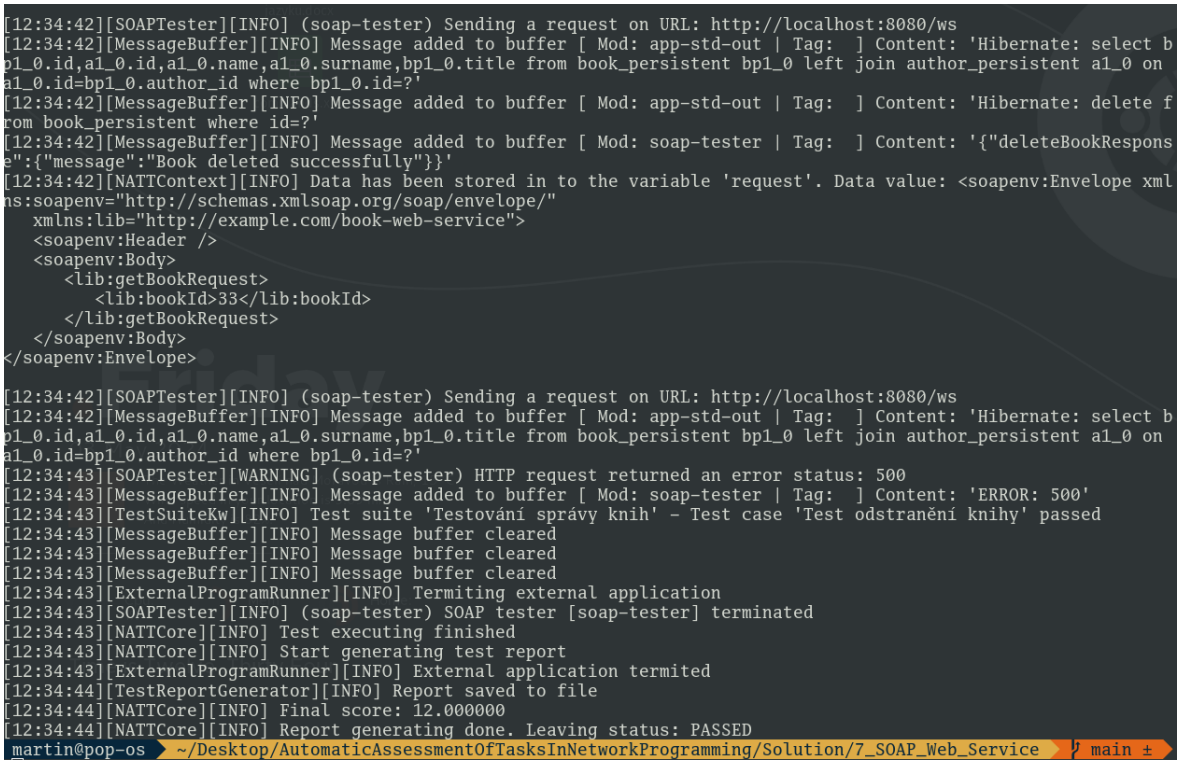
13.2 Nástroj pro automatizované hodnocení síťových aplikací

Výsledný nástroj pro automatizované hodnocení síťových aplikací je univerzální nástroj, který je možné spouštět z terminálu na libovolném zařízení. Je napsán v jazyce Java a tak je zaručena jeho multiplatformní nezávislost. Nástroj je možné ovládat pomocí několika parametrů. Je možné nastavovat cestu k souboru kde je definována testovací sada. Tento soubor může být načten z lokálního zařízení nebo také z jiného síťového zařízení. Dalšími atributy je možné definovat název výstupního reportu nebo vyžádat si pouze validaci konfigurace testů bez jejich provedení. Podrobnější popis se nachází v dokumentaci.

13.2.1 Ukázka spuštění na počítači uživatele

Na obrázku č. 46 se nachází ukázka spuštění nástroje pomocí terminálu na operačním systému Linux. Nástroj hned po spuštění vypisuje na terminál nejdůležitější operace, které

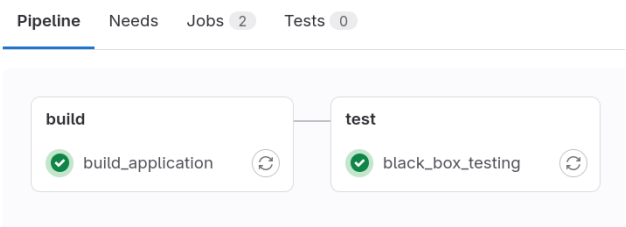
provádí. Po dokončení testování je vypsáno výsledné bodové hodnocení testované úlohy a výsledný stav testování. Pokud testování bylo úspěšné, je nástrojem navrácen status kód 0. V opačném případě je navrácen jiný kód. Po dokončení testovaní je také vygenerován report, který obsahuje průběh a výsledky testování.



Obrázek 46. Spuštění nástroje na počítači uživatele [vlastní zdroj]

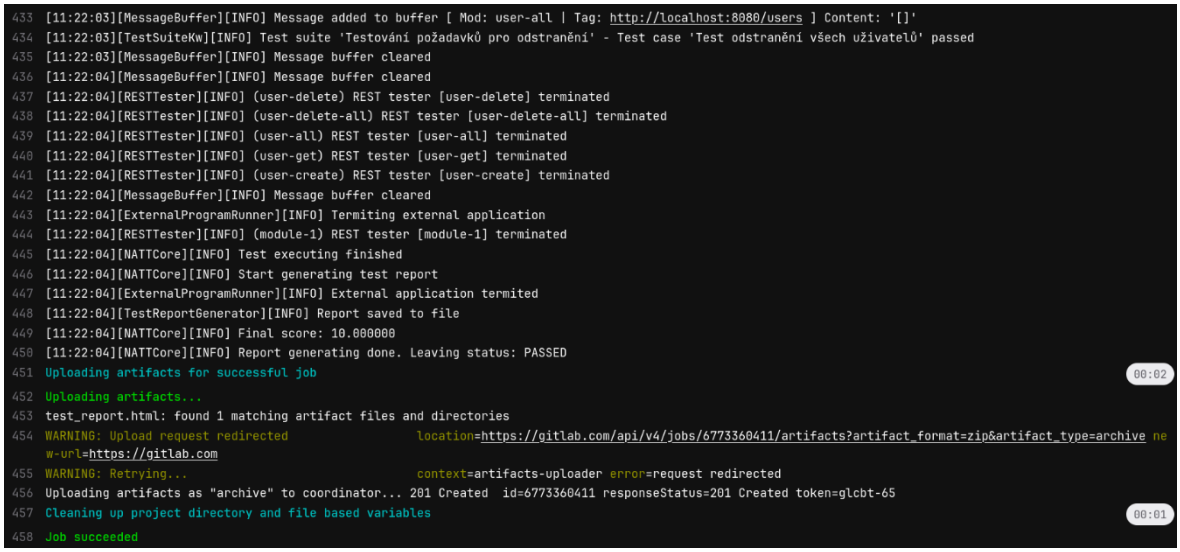
13.2.2 Ukázka spuštění na GitLab server

Nástroj je možné používat také v GitLab CI/CD pipeline. Repozitáře všech úloh už obsahují konfigurace pipeline, které zajišťují, že testování s využitím tohoto nástroje bude automaticky zahájeno hned po nahrání kódu do repozitáře. Princip spuštění je úplně stejný jako na počítači uživatele. V pipeline hraje však důležitou roli status kód, který je vždy navracen při ukončení nástroje. Ten určuje, zda testování proběhlo úspěšně nebo ne.



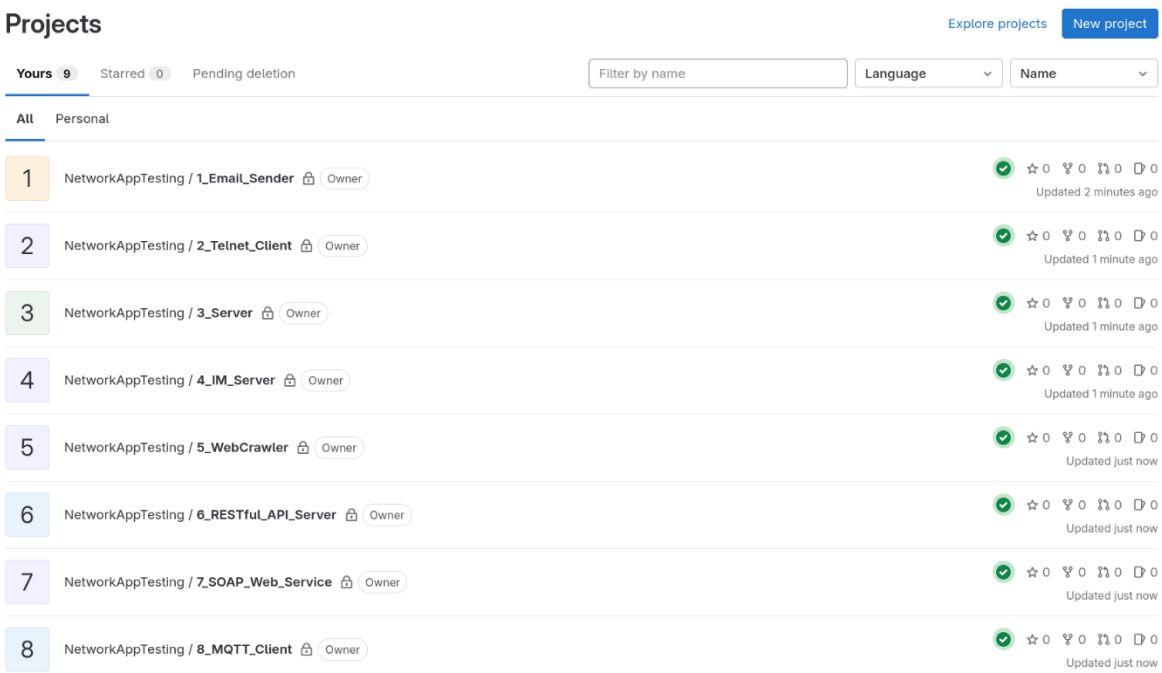
Obrázek 47. Pipelina s úspěšně provedenými testy [vlastní zdroj]

Na obrázku č. 48 se nachází terminál s výstupem testovacího nástroje. Výstup z nástroje je stejný jako na počítači uživatele. I zde po dokončení testování je zanechaný výsledný report, který si studenti mohou, stáhnout z URL odkazu na jedno z posledních řádků výpisu v terminálu.



Obrázek 48. Spuštění nástroje v GitLab pipeline [vlastní zdroj]

Všechny vzorové řešení byli otestování nástrojem. Na obrázků níže se nachází obrázek z GitLab serveru se všemi těmito úlohami a výsledky jejich testování.

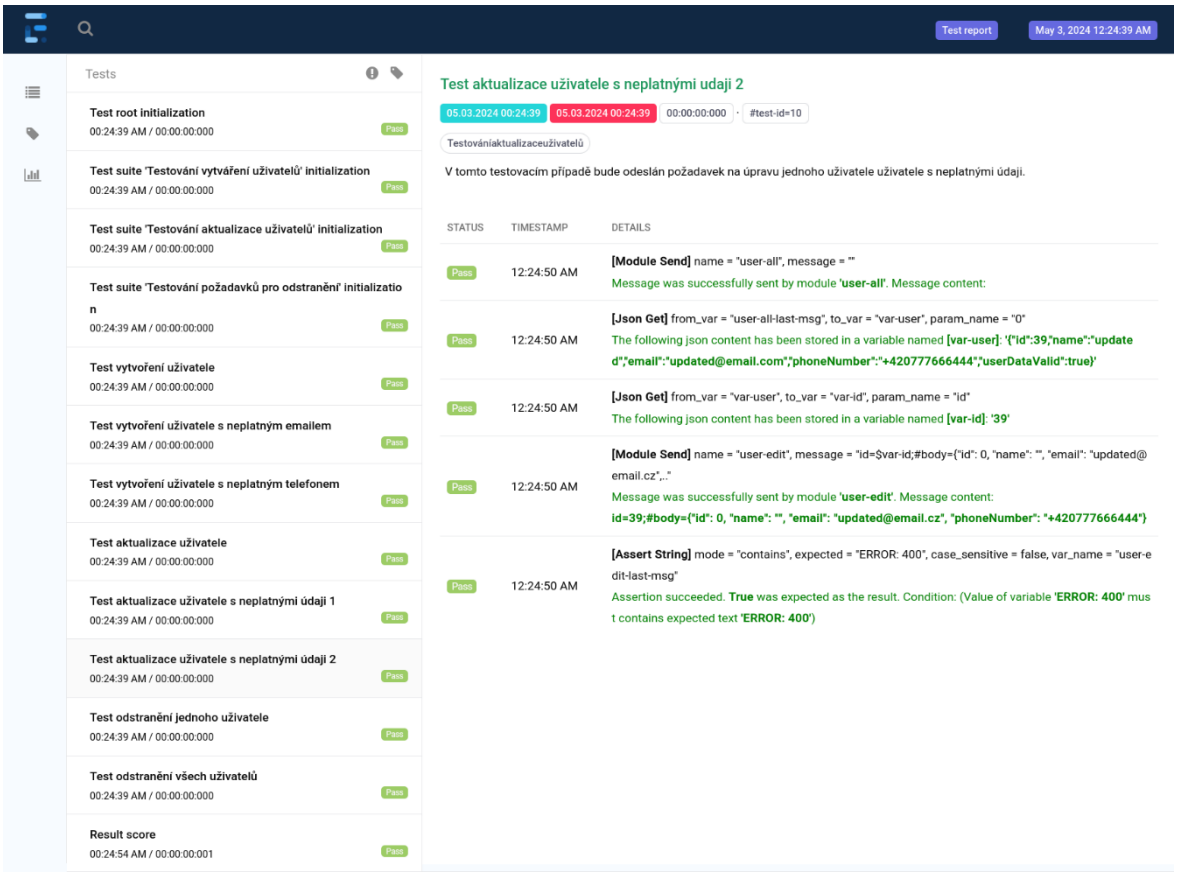


Obrázek 49. Výsledky testování všech úkolů [vlastní zdroj]

13.3 Výstupní report

Všechny reporty jsou generovány jako HTML stránka, kterou si student jednoduše může zobrazit v libovolném prohlížeči. Výsledný report je přehledný, jednoduše čitelný.

Po levé straně je zde seznam všech testovacích případů, které byly vykonány. Výsledné bodové hodnocení úlohy se i zde v reportu nachází pod názvem Result score. Po pravé straně se nachází informace o aktuálně označeném testovacím případu. V horní části je jeho název a popis. Pod ním se nacházejí všechny kroky, který byli v rámci tohoto testovacího případu vykonány. U každého kroku je také zaznamenán jeho stav, zda uspěl nebo ne. V reportu je možné vyhledávat nebo také seskupovat testovací případy do testovacích sad.

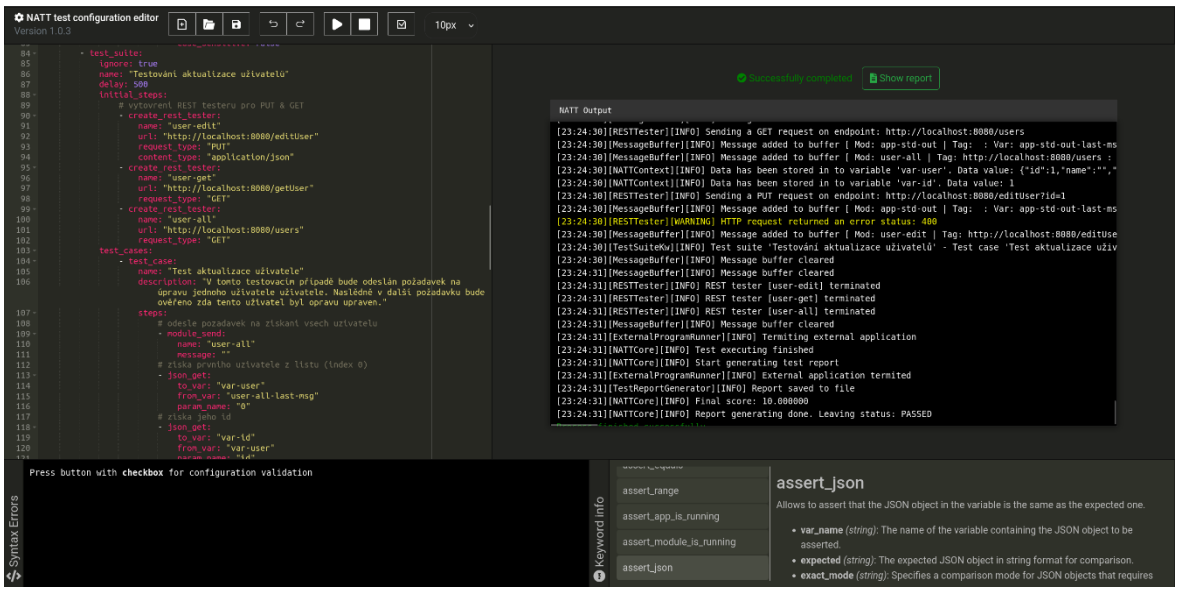


Obrázek 50. Výstupní report generovaný testovacím nástrojem [vlastní zdroj]

13.4 Editor konfigurací

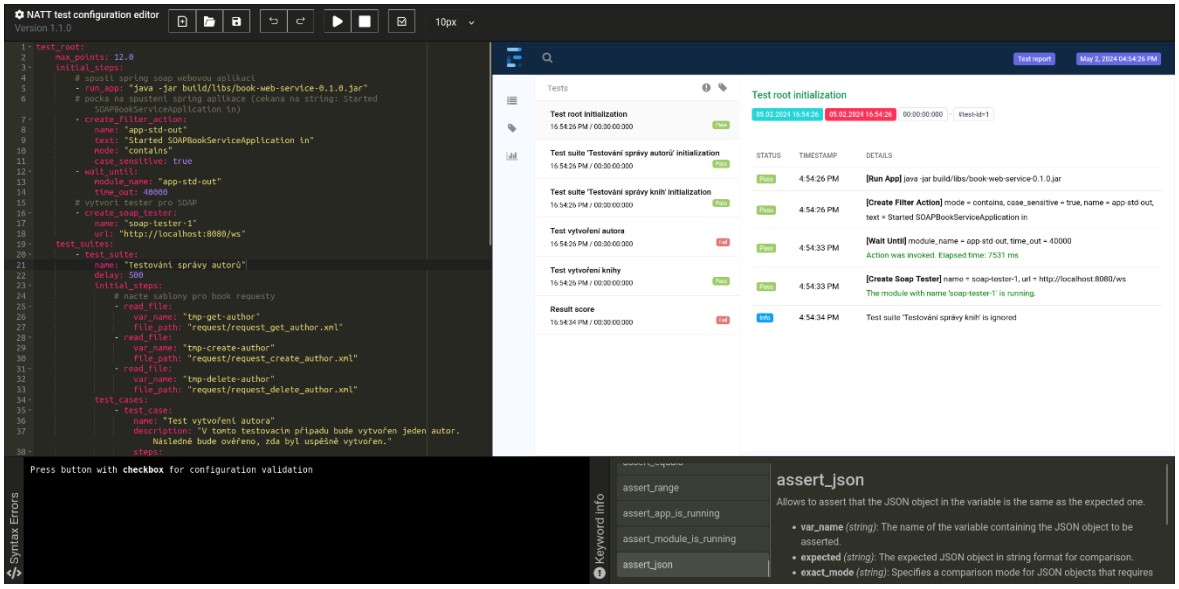
Jelikož konfigurační jazyk k tomuto nástroji obsahuju už relativně hodně klíčových slov a parametrů, které je nutné nastavovat, bylo by komplikované všechny testovací konfigurace psát jen v obyčejném editoru pro YAML.

Za tímto účelem k nástroji vzniklo jednoduché webové prostředí, které umožňuje velice jednoduše psát testovací konfigurace a efektivně je i zde testovat. Toto prostředí je možné spustit na jakémkoliv zařízení. Vyžaduje mít jen nainstalovaný Python a python knihovnu Flask. Pokud uživatel chce i testovací konfigurace spouštět je vyžadována Java v minimální verzi 17.



Obrázek 51. Editor konfigurací s náhledem na výstupní terminál [vlastní zdroj]

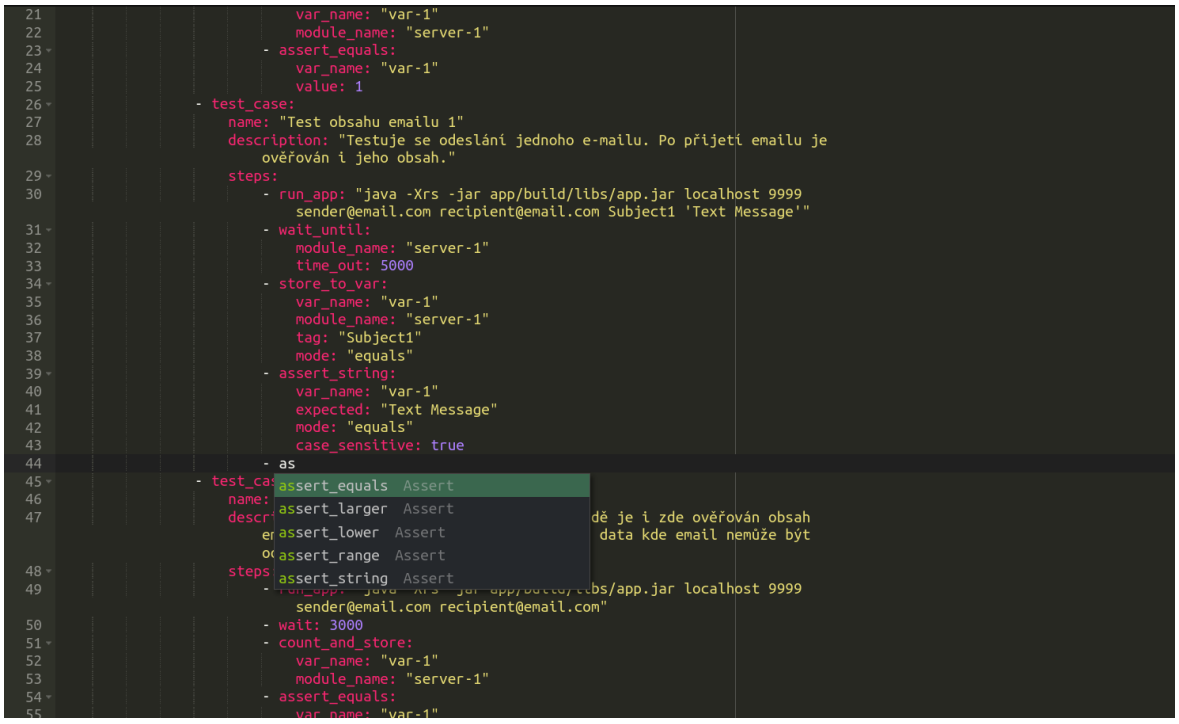
Uživatelské rozhraní editoru se skládá z pěti částí. V levé části se nachází pole, ve kterém se píše samotný kód konfigurace ve formátu YAML. Tato komponenta využívá open-source textového editoru Ace. Na pravé straně se nachází výstup testování. Může se zde nacházet terminál, do kterého je vypisován výstup z nástroje stejně, jak tomu bylo třeba u jobu v GitLab pipeline. Po dokončení testovaní je možné v této časti také zobrazit výsledný report testování. Ve spodní levé části se nachází pole, ve kterém se zobrazují chybové hlášky informující uživatele o případných syntaktických chybách v konfiguraci. Ve spodní pravé části se nachází nápověda ke všem klíčovým slovům a jejich parametrům, které je nutné nastavovat. V horní části se nachází lišta nástrojů, který umožňuje vytvořit novou konfiguraci, otevřít konfiguraci, uložit konfiguraci, krok zpět a vpřed, spuštění testů, zastavení testů, nalezení syntaktických chyb a také možnost změny velikosti písma v editoru.



Obrázek 52. Editor konfiguraci s výsledným reportem [vlastní zdroj]

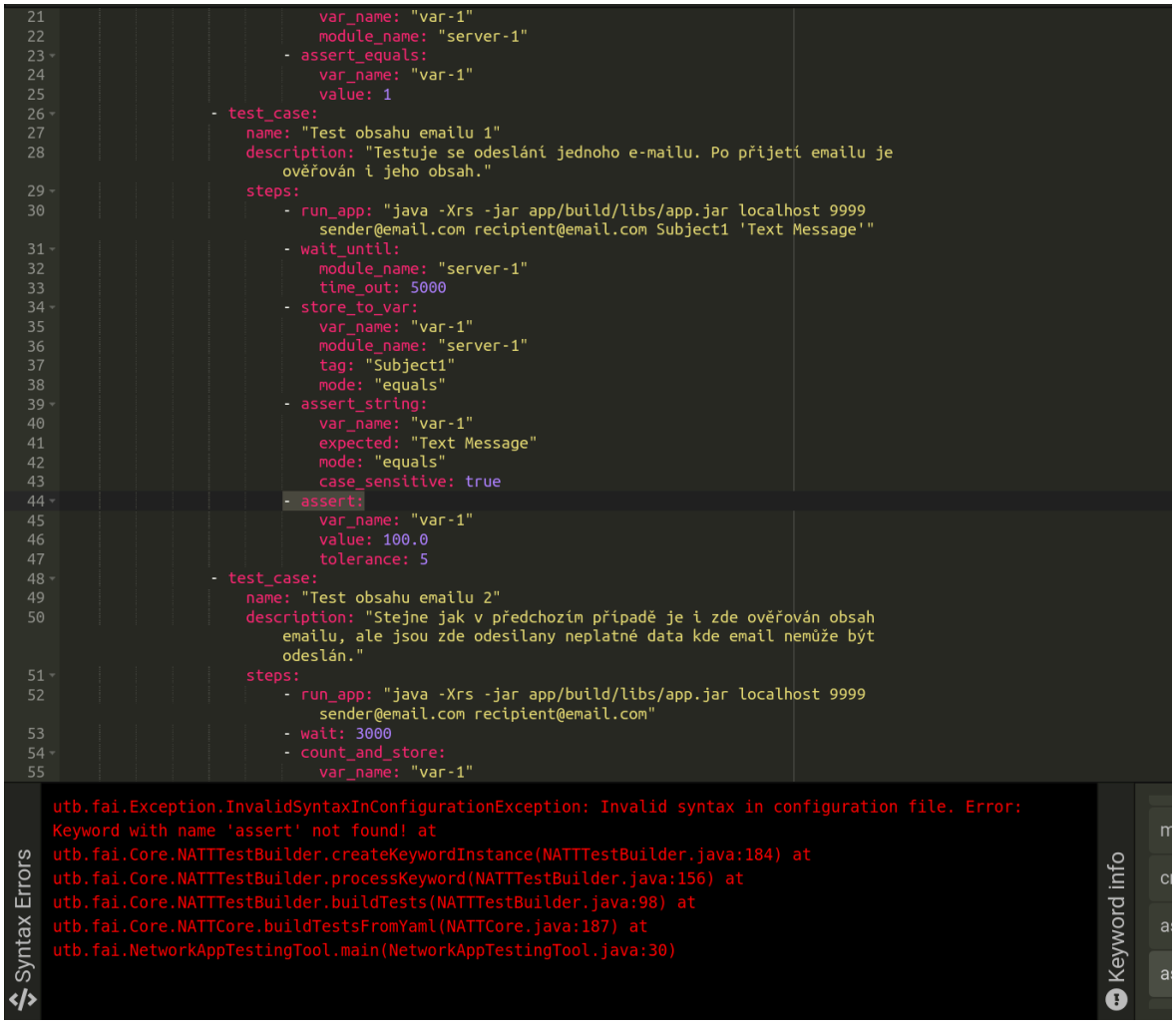
13.4.1 Možnosti editoru

Editor konfigurací nabízí několik funkcí, které výrazně ulehčují a urychlují psaní konfigurací testovacích sad. Jednou z prvních užitečných funkcí, je automatické doplňování kód. To doplní do konfigurace samotné klíčové slovo a i jeho potřebné parametry, kterým také nastaví základní hodnoty. Ty si pak uživatel může podle potřeby pozměnit. Kód, který je psán, se zároveň automaticky ukládá, tak se nemůže stát, že by došlo ke ztrátě kódu.



Obrázek 53. Automatické doplňovaní kód v editoru konfiguraci [vlastní zdroj]

Jednou již ze zmiňovaných mozností je možnost odhalovaní syntaktických chyb v konfiguraci. Tuto kontrolu je možné spustit pomocí tlačítka v horní liště nástrojů. Při této kontrole je využíváno samotného nástroje pro hodnocení síťových aplikací. Tomu je předán samotný konfigurační kód s parametrem pro vykonání validace. V případě nějaké neplatné syntaxe je uživatel editoru informován a je mu přímo řečeno, kde je problém.



Obrázek 54. Zobrazovaní syntaktických chyb v konfiguraci [vlastní zdroj]

ZÁVĚR

Tato diplomová práce se zaměřovala na automatizované hodnocení síťových aplikací v rámci předmětu Programování síťových aplikací. Hlavním cílem bylo implementovat software, který by umožňoval automatizované hodnocení studentských řešení úkolů. Při dosahování tohoto cíle byly provedeny následující kroky.

Nejprve byla provedena analýza stávající sady úkolů v rámci předmětu. Vybrané úlohy byly přepracovány a doplněny o podrobnější zadání. Tato sada úloh byla ještě doplněna o nové úlohy. Pro každou z těchto úloh byla vytvořena šablona projektu a připraveno vzorové řešení pro učitele tohoto předmětu. Šablony projektů byly uzpůsobeny takovým způsobem, aby byly připravené pro použití v repozitářích na GitLab serveru.

V další části byl vyvinut univerzální black-box testovací nástroj přímo na míru, který umožňuje efektivní hodnocení síťových aplikací. Pro všechny úlohy byly napsány testovací sady v konfiguračním jazyce, který hodnotící nástroj dokáže automatizovaně vykonávat. Všechny vzorové řešení úloh byly nástrojem testovány a všechny testy byly provedeny úspěšně ve všech případech. Testování bylo provedeno na GitLab serveru i počítači.

V budoucnu by tento nástroj pro automatizované hodnocení mohl být rozšířen o další funkcionality umožňující testovat širší spektrum síťových aplikací. Dalším vhodným vylepšením nástroje by mohla být funkce automatické detekce plagiátorství. Tato funkce by byla vítaná hlavně pro využití ve výuce. Nástroj by v budoucnu mohl umožňovat definici vlastních klíčových slov nebo podporovat rozšíření v podobě pluginů.

Na závěr je možné říct, že implementace automatizovaného hodnocení síťových aplikací v rámci předmětu Programování síťových aplikací se úspěšně podařila. Toto řešení umožňuje poskytnout studentům přesnější a okamžitou zpětnou vazbu při jejich práci na úkolech, což přispívá k efektivnější výuce.

SEZNAM POUŽITÉ LITERATURY

[1] Eclipse vs. IntelliJ vs. NetBeans: Which Java IDE is Right for You? MOIZ, Muhammad. Medium [online]. 2023 [cit. 2024-02-12]. Dostupné z: https://medium.com/@moiz.patujo/eclipse-vs-intellij-vs-netbeans-which-java-ide-is-right-for-you-390c8c4cbcd7

[2] Client-Server vs. Peer-to-Peer Networks: Similarities and Differences. Online. HOWARD. FS. 2022. Dostupné z: https://community.fs.com/article/client-server-vs-peer-to-peer-networks.html. [cit. 2024-02-11].

[3] Třívrstvá architektura. Management Mania [online]. 2015 [cit. 2024-02-25]. Dostupné z: https://managementmania.com/cs/trivrstva-architektura-three-tier-architecture

[4] Vícevrstvá architektura: tenký, tlustý a chytrý klient. ČERMÁK, Miroslav. Clever and Smart [online]. 2012 [cit. 2024-02-12]. Dostupné z: https://www.cleverand-smart.cz/vicevrstva-architektura-tenky-tlusty-a-chytry-klient/

[5] SOA (Service Oriented Architecture). Management Mania [online]. 2017 [cit. 2024-02-12]. Dostupné z: https://managementmania.com/cs/service-oriented-architecture

[6] TCP/IP vs. OSI: What's the Difference Between the Two Models? FS [online]. 2021 [cit. 2024-02-13]. Dostupné z: https://community.fs.com/article/tcpip-vs-osi-whats-the-difference-between-the-two-models.html

[7] TCP/IP. In: DATA COMMUNICATIONS AND NETWORK TECHNOLOGIES. Singapore: Huawei Technologies Co., 2023, s. 15-72. ISBN 978-981-19-3028-7.

[8] APPLICATION LAYER PROTOCOLS. Online. PATIL, Chaitanya. Medium. 2023. Dostupné z: https://medium.com/@chaituapatil3/application-layer-protocols-62ded821264e. [cit. 2024-02-16].

[9] Difference Between SSL & TLS. Online. DATTA, Subham. Baeldung. 2023. Dostupné z: https://www.baeldung.com/cs/ssl-vs-tls. [cit. 2024-02-16].

[10] KRČMÁRIK, Petr. Animace směrování v IP sitích [online]. Brno, 2012 [cit. 2024-02-16]. Dostupné z: https://is.muni.cz/th/evu4c/krcmibakule060.pdf. Bakalářská práce. Masarykova univerzita, Fakulta informatiky.

[11] What Is a Socket? Oracle Java Documentation [online]. c1995-2022 [cit. 2024-02-17]. Dostupné z: https://docs.oracle.com/javase/tutorial/networking/sockets/definition.html

[12] ORACLE. Java™ Platform, Standard Edition 8 API Specification. Online. C1993-2024. Dostupné z: https://docs.oracle.com/javase/8/docs/api/. [cit. 2024-02-17].

[13] Spring Framework. Online. Spring. C2005 - 2024. Dostupné z: https://spring.io/projects/spring-framework. [cit. 2024-02-19].

[14] J2EE, Java EE a Jakarta EE - Vývoj podnikové Javy. Itnetwork [online]. c2024 [cit. 2024-02-19]. Dostupné z: https://www.itnetwork.cz/java/jee/j2ee-java-ee-a-jakarta-ee

[15] The battle of Version Control: Git vs. SVN vs. Mercurial. Linux Concept [online]. c2019-2023 [cit. 2024-02-19]. Dostupné z: https://linuxconcept.com/git-vs-svn-vs-mercurial/

[16] POWER, Vince. CircleCI vs. GitHub Actions vs. GitLab: Key Differences. Linux Concept [online]. 2023 [cit. 2024-02-19]. Dostupné z: https://saucelabs.com/resources/blog/circleci-vs-github-actions-vs-gitlab-key-differences

[17] LUTHRA, Tarun. How Java Program Works? Scaler Topics [online]. 2023 [cit. 2024-02-20]. Dostupné z: https://www.scaler.com/topics/java/how-java-program-works/

[18] JUnit - Overview. Tutorials point [online]. c2024 [cit. 2024-02-20]. Dostupné z: https://www.tutorialspoint.com/junit/junit_overview.htm

[19] Mockito Framework Tutorial. Java T point [online]. c2011-2021 [cit. 2024-02-20]. Dostupné z: https://www.javatpoint.com/mockito

[20] What is Selenium? Online. BrowserStack. C2024. Dostupné z: https://www.browserstack.com/selenium. [cit. 2024-02-21].

[21] Gradle User Manual. Online. Gradle. C2023. Dostupné z: https://docs.gradle.org/current/userguide/userguide.html. [cit. 2024-02-21].

[22] What is a Container? | Docker. Online. Docker. C2024. Dostupné z: https://www.docker.com/resources/what-container/. [cit. 2024-02-22].

[23] TERŠL, Adam. Paralelní programování a datové struktury v C++ [online]. Brno, 2016 [cit. 2024-02-22]. Dostupné z: https://is.muni.cz/th/ugint/BP.pdf. Masarykova Univerzita Fakulta Informatiky.

[24] Semaphore vs. Mutex. Online. Baeldung. 2023. Dostupné z: https://www.baeldung.com/cs/semaphore-vs-mutex. [cit. 2024-02-22].

[25] Parallel Programming Primer. Online. Princeton Research Computing. 2024. Dostupné z: https://researchcomputing.princeton.edu/support/knowledge-base/parallel-code. [cit. 2024-02-22].

[26] CHOPRA, RAJIV. Software Testing: A Self-Teaching Introduction. Online. Mercury Learning and Information, 2018. ISBN 978-1-683921-66-0. Dostupné z: https://terrorgum.com/tfox/books/softwaretestingprinciplesandpractices.pdf. [cit. 2024-02-26].

[27] Software Testing Life Cycle (STLC). Online. Geeks for Geeks. C2024. Dostupné z: https://www.geeksforgeeks.org/software-testing-life-cycle-stlc/. [cit. 2024-02-26].

[28] Tutorial: Create and run your first GitLab CI/CD pipeline. Online. GitLab | Docs. [2024]. Dostupné z: https://docs.gitlab.com/ee/ci/quick_start/. [cit. 2024-03-01].

[29] Get started with GitLab CI/CD. Online. GitLab | Docs. [2024]. Dostupné z: https://docs.gitlab.com/ee/ci/index.html. [cit. 2024-03-01].

[30] YAML Tutorial: Everything You Need to Get Started in Minutes. Online. ClouBbees. 2023. Dostupné z: https://www.cloudbees.com/blog/yaml-tutorial-everything-you-need-get-started. [cit. 2024-03-01].

[31] REST API Introduction. Online. GeeksForGeeks. 2023. Dostupné z: https://www.geeksforgeeks.org/rest-api-introduction/. [cit. 2024-03-27].

[32] SOAP (Simple Object Access Protocol). Online. TechTarget. 2023. Dostupné z: https://www.techtarget.com/searchapparchitecture/definition/SOAP-Simple-Object-Access-Protocol. [cit. 2024-03-27].

[33] What Is the MQTT Protocol: A Beginner's Guide. Online. MQ. 2024. Dostupné z: https://www.emqx.com/en/blog/the-easiest-guide-to-getting-started-with-mqtt. [cit. 2024-03-27].

[34] Gradle.gitlab-ci.yml. Online. GitLab. 2022. Dostupné z: https://gitlab.com/gitlab-org/gitlab/-/blob/master/lib/gitlab/ci/templates/Gradle.gitlab-ci.yml. [cit. 2024-05-05].

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

IoT	Internet věcí
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
HTTP	Hypertext Transfer Protocol
DNS	Domain Name Systém
DHCP	Dynamic Host Configuration Protocol
SSH	Secure Shell
SMTP	Simple Mail Transfer Protocol
FTP	File Transfer Protocol
Telnet	Teletype network
CI	Kontinuální integrace
CD	Kontinuální nasazení
JDK	Java Development Kit
GUI	Grafické uživatelské rozhraní
IDE	Vývojové prostředí
P2P	Peer-to-Peer
IS	Informační systém
OSI	Open Systems Interconnection
IP	Internet Protocol
MAC	Media Access Control
RIP	Routing Information Protocol
BGP	Border gateway protocol
URL	Uniform Resource Locator
MVC	Model-view-controller

JVM	Java Virtual Machine
SE	Standard Edition
EE	Enterprise Edition
JVM	Java Virtual Machine
JRE	Java Runtime Environment
CPU	Centrální procesorová jednotka
NATT	Network Application Testing Tool

SEZNAM OBRÁZKŮ

SEZNAM TABULEK

SEZNAM PŘÍLOH

PŘÍLOHA P I: ZPROVOZNĚNÍ AUTOMATIZOVANÉHO HODNOCENÍ ÚLOH

V této příloze bude popsáno jak připravit všechny repozitáře, aby v nich bylo zprovozněno automatizované hodnocení. Aby bylo umožněno automatizované vyhodnocování úloh, je nutné distribuovat sestavený black box nástroj do všech repozitářů úloh. To lze provést pomocí skriptu, který se nachází v adresáři s nástrojem 'NetworkAppTestingTool'.

Pro operační systém Linux:

```
cd ./NetworkAppTestingTool

chmod +x distribute_tool_to_tasks.sh

./distribute_tool_to_tasks.sh
```

Pro operační systém Windows:

```
cd NetworkAppTestingTool
start distribute_tool_to_tasks.bat
```

Spuštění nástroje pro automatizované hodnocení:

V rámci GitLab pipeline je nástroj spuštěn automaticky. Na běžné počítači je nástroj možné spustit následujícím příkazem:

```
java -jar NATT.jar -c <cesta-ke-konfiguraci>
```

Ve všech repozitářích s úkoly jsou již předpřipravené skripty, které umožnují nástroj spustit bez psaní tohoto příkazu. Pro operační systém Linux se jedná o skript *„run_local_test.sh“* a pro Windows je zde *„run_local_test.bat“*.

Požadavky:

- Java v minimální verzi 17
- Gradle
- Python

PŘÍLOHA P II: ZADÁNÍ ÚLOHY Č. 1

Zadání úkolu Email Sender

Vaším prvním úkolem je doplnit implementaci třídy EmailSender, která umožní odesílat emailové zprávy pomocí SMTP protokolu. Třída EmailSender by měla umožnovat následující:

1. **Konstruktor EmailSender(String host, int port):** Konstruktor otevře spojení se zadaným SMTP serverem na daném portu.
2. **Metoda send(String from, String to, String subject, String text):** Metoda odesílá emailovou zprávu z určeného emailu na zadaný email příjemce s daným předmětem a textem.
3. **Metoda close():** Metoda odesílá příkaz QUIT na SMTP server a zavírá spojení.

```
package utb.fai;

import java.net.*;
import java.io.*;

public class EmailSender {
    /*
     * Constructor opens Socket to host/port. If the Socket throws
     * an exception during opening,
     * the exception is not handled in the constructor.
     */
    public EmailSender(String host, int port) throws
            UnknownHostException, IOException {

    }
    /*
     * Sends email from an email address to an email address with some
     * subject and text.
     * If the Socket throws an exception during sending, the exception is
     * not handled by this method.
     */
    public void send(String from, String to, String subject, String text)
            throws IOException {

    }

    /*
     * Sends QUIT and closes the socket
     */
    public void close() {

    }
}
```

Druhým úkolem bude doplnění implementace ve třídě Main pro zpracování parametrů předaných aplikaci při spuštění. Při testování aplikace bude testovací nástroj tímto způsobem aplikaci předávat parametry a nasledně hodnotit chování vaší aplikace.

Struktura pole argumentů parametrů

- **args[0]** - Adresa SMTP serveru *(String)*
- **args[1]** - Číslo portu SMTP serveru *(int)*
- **args[2]** - Email odesílatele *(String)*
- **args[3]** - Email příjemce *(String)*
- **args[4]** - Předmět emailu *(String)*
- **args[5]** - Obsah emailu *(String)*

```
package utb.fai;

public class App {

    public static void main(String[] args) {
        // TODO: Implement input parameter processing

        try {
            EmailSender sender = new EmailSender("smtp.utb.cz", 25);
            sender.send("you@utb.cz", "you@utb.cz", "Email from Java",
                  "Funguje to?\nSnad...");
            sender.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Poznámka: *Implementace a struktura kódu aplikace je libovolná a je zcela na vás, jak tento problém vyřešíte. Je však důležité, aby aplikace splňovala zadané požadavky.*

🏆 Způsob hodnocení

Vaše implementace bude hodnocena na základě chování aplikace při testování různých scénářů. Automatizovaný testovací nástroj bude předávat vaší aplikaci různé parametry, včetně platných a neplatných, aby otestoval její chování za různých podmínek. V případě testování síťové komunikace mezi více klienty, testovací nástroj bude vytvářet virtuální klienty/servery za účelem ověření funkcionality.

Výsledné hodnocení bude záviset na celkovém počtu úspěšných testovacích případů. Počet bodů získaných z úlohy bude tedy záviset na celkové úspěšnosti při testování. Váš výsledný počet bodů bude určen následujícím vzorcem.

VP = MB * (UT / CPT)

Popis symbolů:

- **VP:** Výsledný počet bodů.
- **MB:** Maximální počet bodů pro danou úlohu.
- **UT:** Počet úspěšných testovacích případů.
- **CPT:** Celkový počet testovacích případů.

Jak spustit automatizované hodnocení lokálně na svém počítači?

Automatizované hodnocení můžete spustit lokálně za účelem ověření funkčnosti vaší aplikace. K tomu slouží předpřipravený skript, který je dostupný v repozitáři tohoto úkolu. Výsledný report testování se bude nacházet v souboru: `local_test.xml`.

Pro uživatele systému Linux:

Spusťte skript s názvem `run_local_test.sh`.

Pro uživatele systému Windows:

Spusťte skript s názvem `run_local_test.bat`.

Tato část popisující způsob hodnocení a bodování úlohy je pro každou úlohu stejná. Je součástí zadání každé úlohy. Dále už se nebude znovu opakovat.

PŘÍLOHA P III: ZADÁNÍ ÚLOHY Č. 2

Zadání úkolu Telnet klient

Implementujte telnet klienta, který vytvoří soket pro komunikaci s jiným telnet klientem na specifikované IP adrese a portu. Následně na tento soket bude posílat vše, co uživatel napíše na standartní vstup *System.in*. Na standartní výstup *System.out* bude vypisovat vše, co odpoví komunikující protistrana. Tento text vypisujte v nepozměněné formě.

Požadavky:

- Program bude přijímat IP adresu a port jako parametry při spouštění aplikace.
- Zamezte blokování při operaci InputStream.read() pomocí volání InputStream.available(). Snižte zátěž CPU ve smyčce aktivního čekání pomocí volání Thread.sleep(20).
- Implementujte příjem a odesílání znaků ze Socketu v nezávislých vláknech.
- Zpráva bude odeslána ve chvíli, kdy uživatel zmáčkne na klávesnici ENTER
- Pokud uživatel napíše "/QUIT" a zmáčkne ENTER, aplikace ukončí spojení a ukončí se.

Typ: *Aplikaci telnet klienta můžete vyzkoušet například s využitím nástroje netcat, kdy následujícím příkazem na localhostu spustíte server, ke kterému se váš klient připojí.* `nc -lv 127.0.0.1 4444`

```
package main.java.utb.fai;

import java.io.*;
import java.net.*;

public class TelnetClient {

    private String serverIp;
    private int port;

    public TelnetClient(String serverIp, int port) {
        this.serverIp = serverIp;
        this.port = port;
    }

    public void run() {
        try {
            Socket socket = new Socket(serverIp, port);
            // Implementation of receiving and sending data
            // Implement processing of input from the user and sending
            // data to the server
            // Implement response processing from the server and output
            // to the console
```

```
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

V hlavní třídě main implementujte zpracování vstupních parametrů, které jsou aplikaci předávány při spuštění.

Struktura pole vstupních parametrů

- **args[0]** - IP adresa klienta, se kterým bude naše aplikace komunikovat *(String)*
- **args[1]** - Číslo portu, na kterém bude aplikace komunikovat. Standardně 23 pro telnet. *(int)*

```
package main.java.utb.fai;

public class App {

    public static void main(String[] args) {
        // TODO: Implement input parameter processing

        TelnetClient telnetClient = new TelnetClient("127.0.0.1", 23);
        telnetClient.run(); // run telnet client
    }
}
```

Poznámka: *Implementace a struktura kódu aplikace je libovolná a je zcela na vás, jak tento problém vyřešíte. Je však důležité, aby aplikace splňovala zadané požadavky.*

PŘÍLOHA P IV: ZADÁNÍ ÚLOHY Č. 3

Zadání úkolu Telnet server

Naprogramujte jednoduchý server, který bude schopný přijímat spojení od neomezeného počtu klientů. Server bude vykonávat funkci (tzv. “echo” funkce). To znamená, že zprávu, kterou obdrží od libovolného klienta, mu ji přepošle zpět. Pro obsluhu více klientů vytvořte jedno vlákno pro každé připojení. Jako klienta využijte aplikaci telnet klienta, kterého jste implementovali již v předchozí úloze.

Požadavky:

- Server musí být schopen přijímat spojení od neomezeného počtu klientů.
- Pro obsluhu každého klienta bude samostatné vlákno. Implementace ve třídě `ClientThread`.
- Klientovi server pošle zpět zprávu, kterou od něj přijme (Echo).
- Možnost omezení počtu vláken s využitím ThreadPoolExecutor.

```
package utb.fai;

import java.io.*;
import java.net.*;

public class ClientThread extends Thread {

    private Socket clientSocket;

    public ClientThread(Socket clientSocket) {
        this.clientSocket = clientSocket;
    }

    @Override
    public void run() {
        // Implementation of processing incoming communication from the
        // telnet client
    }
}
```

V hlavní třídě aplikace implementujte hlavní smyčku serveru. Tedy otevírání nových spojení mezi klientem a serverem. Využijte třídu `ServerSocket`. Při spouštění aplikace budou aplikaci serveru předávány následující parametry.

Struktura pole vstupních parametrů

- **args[0]** - Port, na kterém bude server naslouchat *(int)*
- **args[1]** - Maximální počet vláken. V tomto případě číslo bude i odpovídat maximálnímu počtu klientů, kteří budou moci být v jeden okamžik připojení. *(int)*

```java
package utb.fai;

import java.io.*;
import java.net.*;
import java.util.concurrent.*;

public class App {

    public static void main(String[] args) {
        int port = 12345;
        int max_threads = 10;

        // Implement input parameter processing

        // Implementation of the main server loop
        ServerSocket serverSocket;
    }
}
```

Poznámka: *Implementace a struktura kódu aplikace je libovolná a je zcela na vás, jak tento problém vyřešíte. Je však důležité, aby aplikace splňovala zadané požadavky.*

PŘÍLOHA P V: ZADÁNÍ ÚLOHY Č. 4

Zadání úkolu IM Server

Cílem této úlohy bude implementovat instant message server. Princip je v zásadě podobný jako u předchozí úlohy při implementaci jednoduchého echo serveru. Pouze budou přidány další funkcionality, které budou popsány dále v zadání této úlohy. Jako klienta můžete použít telnet klienta, kterého jste implementovali již v jedné z předchozích úloh.

Požadavky:

- **"Identifikace uživatelů"** - Uživatel bude schopen zasláním speciálního řetězce `#setMyName <jméno>` změnit své jméno, které se zobrazuje ostatním klientům. Uživatel bude moct ostatním posílat zprávy až ve chvíli, kdy bude mít nastavené jméno. Jméno musí být bez mezer! Hned po připojení klienta na IM server bude vyzván k zadání svého jména a to bez nutnosti psaní příkazu *#setMyName*.
- **Možnost zasílání soukromých zpráv** - Možnost zaslání soukromé zprávy pomocí speciálního řetězce `#sendPrivate <jméno> <zpráva>`
- **Možnost připojit se do diskuzní místnosti** - Uživatel bude mít možnost připojit se do libovolné diskuzní místnosti. To provede tím, že zadá následující příkaz `#join <název>`. Pokud taková místnost ještě na serveru neexistuje, tak vytvoří novou. Ve chvíli kdy uživatel píše zprávy tak je vidí jen ti uživatelé, kteří jsou ve stejné skupině jak uživatel, který je odesílá. Při připojení nového uživatele na server bude automaticky zařazen do diskuzní místnosti s názvem ***"public".***
- **Možnost odpojit se z diskuzní místnosti** - Uživatel bude mít také možnost z místnosti odejít. To provede následujícím příkazem `#leave <název>`.

Základní implementace již funkčního IM serveru naleznete v tomto repozitáři. Vaším úkolem bude tedy jen jeho rozšíření o výše zmíněné funkcionality.

Popis základní implementace IM serveru

Jedná se o jednoduchý server, který zprávu = řetězec zaslaný po síti v kódování UTF-8 od klienta rozesílá všem ostatním klientům.

O každého klienta se stará jedna instance třídy SocketHandler, v níž jsou definovány 2 vnitřní třídy, tasky pro ThreadPool OutputHandler a InputHandler. Oba tyto tasky běží paralelně, jeden se stará o příchozí data ze socketu klienta, druhý o odchozí data.

Jedná se o komunikaci klient1-server-klient2 a ta funguje tak, že klient1 pošle data, jeho InputHandler je přijme a vloží je do všech front (proměnná "messages") všech aktivních SocketHandlerů.

OutputHandler klienta2 čeká na data, která přijdou do jeho fronty zpráv. Jakmile přijdou, probudí se, data z fronty vezme a pošle je svému klientovi.

Všichni právě připojení klienti (resp. jejich SocketHandler-s) jsou uchovávání v množině activeHandlers.

Typ

Pokud bychom chtěli přidat funkci pro doručování zpráv jen vybraným klientům, bylo by lepší místo množiny HashSet použít ConcurrentHashMap<String, SocketHandler>, v níž bychom jako klíč používali řetězec clientID. Zprávy, které mají jít jen jednomu klientovi, by pak mohly začínat řetězcem clientID - pokud by server na začátku zprávy našel clientID, našel by si v HashMapě podle něj referenci na SocketHandler, který tohoto klienta obsluhuje, a zprávu by přidal do fronty pouze jemu.

Struktura pole vstupních parametrů

- **args[0]** - Port, na kterém bude server naslouchat *(int)*
- **args[1]** - Maximální počet klientů, kteří se mohou k serveru v jeden okamžik připojit. *(int)*

Seznam všech příkazů

Příkaz	Popis
`#setMyName <jméno>`	Změní jméno uživatele na zadané `<jméno>`. Jméno musí být bez mezer.
`#sendPrivate <jméno> <zpráva>`	Zasílá soukromou zprávu zadanému `<jméno>` s obsahem `<zpráva>`.

`#join <název>`	Připojí uživatele do diskuzní místnosti se zadaným `<název>`. Vytvoří novou místnost, pokud ještě neexistuje. Klient je po připojení automaticky připojen do místnosti "public"**.**
`#leave <název>`	Odpojí uživatele z diskuzní místnosti se zadaným `<název>`.
`#groups`	Vypíše seznam všech diskuzních místností, ve kterých se klient nachází. Výpis bude v tomto formátu: `<místnost1>,<místnost2>,…`

Formát zprávy

Když uživatel napíše ostatním nějakou zprávu, bude se ostatním uživatelům zobrazovat následujícím způsobem: `[<jméno>] >> <zpráva>`

Poznámka: *Implementace a struktura kódu aplikace je libovolná a je zcela na vás, jak tento problém vyřešíte. Je však důležité, aby aplikace splňovala zadané požadavky.*

PŘÍLOHA P VI: ZADÁNÍ ÚLOHY Č. 5

Zadání úkolu Web crawler

Naprogramujte web crawler pro prohledávání webu a identifikaci 20 nejčastěji se vyskytujících slov. Aplikaci bude možné ovládat přes příkazový řádek a umožní specifikovat parametry pro hloubku zanoření a úroveň ladění programu. Implementujte parsování HTML stránek, procházení odkazů na stránce a analýzu četnosti slov.

Požadavky:

- **Parsování HTML stránek:** K implementaci tohoto úkolu můžete využít buď HTML parser přímo v JDK v balíčku *javax.swing.text.html.parser*, nebo open source parser pro HTML 4.0, např. *JSoup*. JSoup knihovna je již v tomto repozitáři zahrnuta a připravena na případné použití.
- **Procházení odkazů:** Program musí procházet všechny odkazy na stránce a ukládat je do seznamu nalezených odkazů. Musí také udržovat množinu všech navštívených stránek, aby se stejné stránky neanalyzovaly vícekrát.
- **Analýza četnosti slov:** Aplikace bude umožnovat analýzu četnosti slov. Pro implementaci je doporučené využít datovou strukturu "vyhledávací tabulka" (HashMap nebo TreeMap).
- **Ošetření kódování stránek:** Získání kódování stránek z HTTP hlavičky nebo značky META.
- **Multithreading:** Načítání a analýza každé stránky v nezávislém vláknu pomocí Executoru pro využití více jádrových procesorů. Seznam "visitedURIs" implementujte pomocí ConcurrentHashMap.

Formát výstupu aplikace

Po spuštění aplikace prohledá web a po dokončení analýzy vypíše slova s jejich absolutní četností seřazená sestupně na standardní výstup *System.out*. Aplikace musí vypisovat pouze tyto informace a nic jiného navíc a to za celou dobu jejího běhu.

Ukázkový výstup

```
the;47861
to;42777
and;37061
of;31670
your;29636
or;27962
you;22063
a;19607
data;19221
for;17302
information;14858
on;14341
our;12409
in;12381
is;11457
we;11143
with;10905
that;10462
as;9936
may;8774
```

Struktura pole vstupních parametrů

1. **args[0]** - URL stránky, na které program zahájí analýzu.
   Například: http://yahoo.com *(String)*
2. **args[1]** - Tento parametr bude udávat maximální hloubku zanoření při analýze. Tento parametr bude nepovinný. Pokud při spustění aplikace nebude zadán, bude nastaven na 2. (Hloubka 0 znamená, že bude analyzován obsah jen stránky, na které je analýza zahájena). *(int)*
3. **args[2]** - Tento parametr bude udávat, v jaké hloubce zanoření budou vypisovány případné debug informace na System.err. Tento parametr bude nepovinný. Můžete využít pro vlastní účely debuggování. Při hodnocení nebude brán v úvahu. *(int)*

Základní kostra programu se nachází v tomto repozitáři. Vaším úkolem ji bude předělat tak, aby splňovala výše uvedené požadavky.

Poznámka: *Implementace a struktura kódu aplikace je libovolná a je zcela na vás, jak tento problém vyřešíte. Je však důležité, aby aplikace splňovala zadané požadavky.*

PŘÍLOHA P VII: ZADÁNÍ ÚLOHY Č. 6

Zadání úkolu RESTful API Server

Naprogramujte RESTful API Server pomocí frameworku Spring Boot, který bude poskytovat služby pro správu seznamu uživatelů. Tato aplikace bude spuštěna na portu 8080 a bude využívat http připojení. Vaše aplikace bude podporovat následující operace:

- **Získání seznamu uživatelů:** API endpoint pro získání seznamu všech uživatelů v systému.
- **Získání jednoho uživatele:** API endpoint pro získání jednoho konkrétního uživatele podle jeho id. Pokud není uživatel nalezen, tak navrátí HTTP status kód 404 Not Found.
- **Vytvoření nového uživatele:** API endpoint pro vytvoření nového uživatele v systému. Endpoint bude validovat zadané data (email a telefoní číslo). Při úspěšném vytvoření navratí HTTP status kód 201. Pokud tyto data nebudou splňovat požadovaný formát, nebude uživatel v systému vytvořen a server navrátí HTTP status kód 400 Bad Request.
- **Aktualizace existujícího uživatele:** API endpoint pro aktualizaci informací o existujícím uživateli v systému. Pokud takový uživatel neni nalezen, tak navratí HTTP status kód 404 Not Found. Pokud dojde k jiné chybě, navrátí HTTP status kód 400.
- **Smazání uživatele:** API endpoint pro smazání existujícího uživatele ze systému. Pokud dojde k nějaké chybě, navrátí HTTP status kód 400.
- **Smazání všech uživatelů:** API endpoint, který odstraní všechny uživatele z databáze. Pokud dojde k nějaké chybě, navrátí HTTP status kód 400.

Typ požadavku	Název endpointu	URL adresa endpointu	Předávané parametry	Formát odpovědi
GET	Získání seznamu uživatelů	/users	N/A	JSON
GET	Získání informací o jednom uživateli	/getUser	ID uživatele	JSON
POST	Vytvoření nového uživatele	/createUser	Data nového uživatele	JSON

PUT	Aktualizace existujícího uživatele	/editUser	ID uživatele, Data aktualizovaného uživatele	JSON
DELETE	Smazání uživatele	/deleteUser	ID uživatele	JSON
DELETE	Smazání všech uživatelů	/deleteAll	N/A	JSON

Poznámka: *Funkcionalitu vaší aplikace můžete manuálně otestovat pomocí nástroje cURL. https://curl.se/* Pro získání všech uživatelů můžete použít tento příkaz:

```
curl -X GET http://localhost:8080/users -H 'Content-Type: applica-
tion/json'
```

Datová struktura uživatele:

Do této třídy budou v aplikaci ukládany data o uživateli. Bude potřeba dokončit její implementaci a i vhodně zvolit anotace, aby ji bylo možné ukládat do databáze prostřednictvím JPA repozitáře.

```
package utb.fai.RESTAPIServer;

class MyUser {

    private Long id;
    private String name;
    private String email;
    private String phoneNumber;

    public User() {}

    public User(String name, String email, String phoneNumber) {
        this.name = name;
        this.email = email;
        this.phoneNumber = phoneNumber;
    }

    public boolean isUserDataValid() {
        // Add your validation logic here (e.g., email and phone number
        // format validation)
        return true;
    }

    // TODO: Getters and setters
}
```

Pro ukládání uživatelských dat použijte libovolný volně dostupný databázový systém (např. MySQL, PostgreSQL, H2, MongoDB atd.), který bude vaše aplikace využívat. Můžete využít například následující službu, která nabízí free hosting databází: https://aiven.io/

Přístup k vaší databázi bude třeba nakonfigurovat v souboru `application.properties`, který se nachází ve složce `src/main/resources`.

Základní kostra programu se nachází v tomto repozitáři. Vaším úkolem bude implementovat všechny výše zmíněné endpointy.

Poznámka: *Implementace a struktura kódu aplikace je libovolná a je zcela na vás, jak tento problém vyřešíte. Je však důležité, aby aplikace splňovala zadané požadavky.*

PŘÍLOHA P VII: ZADÁNÍ ÚLOHY Č. 7

Zadání úkolu SOAP webová služba

Implementujte SOAP webovou službu pro správu knih a jejich autorů v systému. Webová služba by měla umožňovat základní práci s knihami a autory. To zahrnuje vytvoření, aktualizaci, mazání a také získávání informací. Aplikace bude data ukládat do relační databáze MySQL. Tato aplikace bude spuštěna na portu 8080 a bude využívat http připojení.

Požadavky:

- Implementujte SOAP webovou službu v jazyce Java.
- Pro tvorbu SOAP webových služeb využijte framework Spring boot.
- Pro ukládání knih a autorů v systému použijte technologii JDBC a relační databázi MySQL. Můžete využít například následující službu, která poskytuje free hosting MySQL databaze: https://aiven.io/
- Implementujte zpracovaní pro všechny níže zmíněné SOAP requesty. Všechny požadavky budou zasílány na tento endpoint `http://localhost:8080/ws`.
- U požadavků, které do databáze vkládají data, musí být jejich parametry validované. V případě jména/příjmení je nutné zajistit, aby se skládalo pouze z jednoho slova. U všech string parametru zároveň platí, že musí vždy obsahovat nějakou hodnotu.
- SOAP requesty a respons definujte v xml souboru `library.xsd`, který se nachází ve složce `src/main/resources`.

Seznam SOAP requestů pro tuto aplikaci

Název požadavku	Popis	Atributy	Typy atributů
`getBookRequest`	Získání informací o knize	`bookId`	`long`
`createBookRequest`	Vytvoření nové knihy	`book`	`Book`
`updateBookRequest`	Aktualizace existující knihy	`bookId`, `book`	`long`, `Book`
`deleteBookRequest`	Smazání knihy	`bookId`	`long`
`getAuthorRequest`	Získání informací o autorovi	`authorId`	`long`
`createAuthorRequest`	Vytvoření nového autora	`author`	`Author`
`deleteAuthorRequest`	Smazání autora	`authorId`	`long`

Seznam SOAP response pro tuto aplikaci

Název odpovědi	Popis	Atributy	Typy atributů
`getBookResponse`	Odpověď s informacemi o knize	`book`	`Book`
`createBookResponse`	Odpověď s informacemi o vytvořené knize	`book`	`Book`
`updateBookResponse`	Odpověď s informacemi o aktualizované knize	`book`	`Book`
`deleteBookResponse`	Potvrzení smazání knihy	`message`	`string`
`getAuthorResponse`	Odpověď s informacemi o autorovi	`author`	`Author`
`createAuthorResponse`	Odpověď s informacemi o vytvořeném autorovi	`author`	`Author`
`deleteAuthorResponse`	Potvrzení smazání autora	`message`	`string`

Datová struktura knihy a autora

Do relační databáze budou ukládány dvě datové struktury, jejichž struktura a názvy atributů jsou následující. Jedna ze struktur bude pro autora knihy. Autor bude na knize nezávislý. Každá kniha bude mít svůj název a také autora, který už bude v databázi mít záznam. Jeden autor může mít více knih a kniha nemůže být bez autora. Při odstranění autora z databáze budou také odstraněny i jeho knihy.

```
public class Author {

    private Long id;

    private String name;

    private String surname;

    // TODO: Getters and setters
}

public class Book {

    private Long id;
```

```
    private String title;

    private Author author;

    // TODO: Getters and setters
}
```

Návod na otestování SOAP webové služby

Pro vyzkoušení této aplikace můžete využít nástroj cURL. Pro ukázku zde je požadavek na vytvoření nového autora knihy. Ostatní požadavky se provádějí stejným způsobem.

```
curl --header "content-type: text/xml" -d @request_create_author.xml
http://localhost:8080/ws
```

Zde je obsah souboru `request_create_author.xml`, který posíláme na server pomocí curl příkazu. Server tento požadavek zpracuje a následně odpoví taky formou xml.

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/enve-
lope/"
   xmlns:lib="http://example.com/book-web-service">
   <soapenv:Header />
   <soapenv:Body>
      <lib:createAuthorRequest>
         <lib:author>
            <lib:id>1</lib:id>
            <lib:name>Your name</lib:name>
            <lib:surname>Your surname</lib:surname>
         </lib:author>
      </lib:createAuthorRequest>
   </soapenv:Body>
</soapenv:Envelope>
```

Poznámka: *Implementace a struktura kódu aplikace je libovolná a je zcela na vás, jak tento problém vyřešíte. Je však důležité, aby aplikace splňovala zadané požadavky.*

PŘÍLOHA P IX: ZADÁNÍ ÚLOHY Č. 8

Zadání úkolu MQTT klient pro monitorování vlhkosti půdy a řízení závlahy

Implementujte MQTT klienta pro monitorování vlhkosti půdy a řízení závlahy v zemědělských prostředích. Klient bude pravidelně odesílat zprávy obsahující aktuální naměřené hodnoty vlhkosti půdy a přijímat příkazy pro řízení zavlažovacích systémů. Senzor vlhkosti a zavlažovací systém nebude v rámci této úlohy fyzicky k dispozici, ale budou virtuálně simulovány v programu. Pro obě tyto zařízení budete mít již v repozitáři připravené funkční softwarové komponenty.

Pro vlastní testování je i součásti repozitáře testovací MQTT klient, kterého můžete spustit příkazem `java -jar TestingMQTTClient.jar`. Ten umožňuje zobrazovat a manuálně odesílat zprávy na vámi programovaného klienta. Bližší návod na používání v tomto nástroji zadání příkazů `/help` nebo v tomto souboru.

Náš klient bude v MQTT systému publisher a zároveň i subscriber. A to z tohoto důvodu, že tento klient umožňuje jak čtení hodnot z čidel, tak i zajišťuje řízení závlahy z jiného zařízení. Hlavní řídicí systém a zároveň i webové rozhraní bude toto zařízení monitorovat a i řídit jeho chod. V rámci této úlohy však je cílem implementace pouze klienta, který je ve schématu na pravé straně (Obrázek č.).

Klient, kterého budete implementovat, bude odebírat tento topic *"topic/device1/in"*. Tento topic bude u našeho klienta vyhrazen čistě pro příchozí zprávy. Pro odchozí zprávy zde bude využíván topic *"topic/device1/out"*.

Požadavky:

- Klient se připojí k MQTT brokeru, který v rámci tohoto úkolu bude virtuálně hostovaný na lokálním zařízení. Adresu připojení nastavte následující `tcp:/localhost:1883`.
- Implementujte zpracování přijimání a odesílání zpráv a také logiku jejich následného zpracovaní.
- MQTT klient bude hodnotu vlhkosti naměřené z čidla pravidelně odesílat po 10 sekundovém intervalu. První odeslaná hodnota bude hned po *spuštění* klienta.

Systém, který bude zpracovávat tyto naměřené data, může také zažádat o okamžité zaslání hodnoty.

- Při příchodu zprávy pro spuštění závlahy klient tuto akci okamžitě provede. Zavlažování bude trvat tak dlouho, dokud neuplyne časový limit 30 sekund od poslední přijaté zprávy pro zahájení závlahy. Zavlažování bude ukončeno okamžitě v případě, kdy přijde zpráva pro okamžité zastavení.
- Klient bude řídicí systém informovat o úspěšném spuštění zavlažování a jeho zastavení prostřednictvím odesílaných stavových zpráv.
- Klienta bude možné požádat o zaslání stavu o zavlažování. Tedy zda je zavlažování v aktuální okamžik aktivováno nebo ne.
- Pokud dojde k nějaké poruše čidla nebo zavlažovacího systému, musí klient tuto skutečnost oznámit odeslání stavové zprávy s touto chybou.

Zprávy a jejich struktura

V následující tabulce jsou popsány všechny zprávy, prostřednictvím kterých bude tento MQTT klient komunikovat s okolními systémy. Data odesílaných zpráv budou vždy v následujícím formátu:

`<název zprávy>;<hodnota parametru zprávy>.`

Zpráva	**Popis**	**Struktura**	**Typ zprávy**
Odeslání hodnoty vlhkosti	Zpráva obsahující hodnotu naměřené vlhkosti z čidla.	`humidity;<hodnota vlhkosti>`	OUT
Okamžité zaslání vlhkosti	Zpráva pro vyžádání okamžitého zaslání aktuální hodnoty vlhkosti.	`get-humidity`	IN
Vyžádání stavu	Zpráva pro vyžádání zaslání stavu zavlažování. Zda je, nebo není aktivní.	`get-status`	IN

Spuštění závlahy	Zpráva pro spuštění zavlažování.	`start-irrigation`	IN
Zastavení závlahy	Zpráva pro okamžité zastavení zavlažování.	`stop-irrigation`	IN
Stavová zpráva - Závlaha aktivována	Zpráva informující o tom, že je zavlažování aktivováno.	`status;irrigation_on`	OUT
Stavová zpráva - Závlaha deaktivována	Zpráva informující o tom, že je zavlažování deaktivováno.	`status;irrigation_off`	OUT
Stavová zpráva - Chyba	Zpráva informující o výskytu chyby čidla nebo zavlažovacím systému.	`fault;<typ chyby>`	OUT

Tento repozitář již obsahuje základní strukturu. Typy chyb a jednotky, ve kterých čidlo vlhkosti měří, se nachází v šabloně připraveny k použití. Při odesílání hodnoty vlhkosti do zprávy zapište RAW hodnotu, kterou vám poskytne předpřipravené API tohoto čidla. API k čidlu a zavlažovacímu systému a jejich popis se nacházejí ve složce API ve zdrojovém kódu.

Struktura pole vstupních parametrů

1. **args[0]** - Seed pro generování náhodných čísel. Využíváno k simulaci modulů. *(long)*
2. **args[1]** - Poruchovost čidla vlhkosti. Číslo je v rozsahu od 0 po 1 a udává pravděpodobnost, že při čtení hodnoty vlhkosti dojde k jeho poškození. *(float)*
3. **args[2]** - Poruchovost zavlažovacího systému. Číslo je v rozsahu od 0 po 1 a udává pravděpodobnost, že při libovolné manipulaci (spuštění/zastavení) dojde k jeho poškození. *(float)*

Do následující třídy implementujte klienta

```
package utb.fai;

import org.eclipse.paho.client.mqttv3.MqttClient;
import utb.fai.API.HumiditySensor;
import utb.fai.API.IrrigationSystem;

/**
 * Trida MQTT klienta pro mereni vhlkosti pudy a rizeni zavlazovaciho
 * systemu. V teto tride implementuje MQTT klienta
 */
public class SoilMoistureMQTTClient {

    private MqttClient client;
    private HumiditySensor humiditySensor;
    private IrrigationSystem irrigationSystem;

    /**
     * Vytvori instacni tridy MQTT klienta pro mereni vhlkosti pudy a
     * rizeni zavlazovaciho systemu
     *
     * @param sensor     Senzor vlhkosti
     * @param irrigation Zarizeni pro zavlahu pudy
     */
    public SoilMoistureMQTTClient(HumiditySensor sensor, IrrigationSystem
          irrigation) {
        this.humiditySensor = sensor;
        this.irrigationSystem = irrigation;
    }

    /**
     * Metoda pro spusteni klienta
     */
    public void start() {

    }

}
```

Poznámka: *V této úloze dodržujte danou strukturu kódu aplikace. Součástí vašeho klienta jsou také moduly obsahující API k čidlu a zavlažovacímu systému. U modulů jde o simulaci a některé jevy, které mohou nastat, jsou pseudonáhodné. Tyto moduly vyžadují při spuštění klienta konfiguraci v podobě počátečních podmínek a seedu. Hodnotící nástroj si tyto podmínky v průběhu testování nastaví sám.*