Univerzita Pardubice
Fakulta elektrotechniky a informatiky

Systém pro automatickou kontrolu odevzdaných zdrojových kódů

Bc. Petr Váňa

Diplomová práce

2021

Univerzita Pardubice
Fakulta elektrotechniky a informatiky
Akademický rok: 2020/2021

ZADÁNÍ DIPLOMOVÉ PRÁCE
(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Bc. Petr Váňa**
Osobní číslo:	**I19292**
Studijní program:	**N0613A140007 Informační technologie**
Studijní obor:	**Informační technologie**
Téma práce:	**Systém pro automatickou kontrolu odevzdaných zdrojových kódů**
Zadávající katedra:	**Katedra softwarových technologií**

Zásady pro vypracování

Cílem diplomové práce je navrhnout, implementovat a nasadit systém pro automatickou kontrolu odevzdaných zdrojových kódů.
Předpokladem je návrh systému je zaměřením na automatickou kontrolu zdrojových kódů pro jazyk Java SE s možností tvorby testovacích scénářů vůči kterým budou zdrojové kódy testovány. Nutná je podpora jak pro práci s primitivními, tak i referenčními datovými typy s cílem poskytnout podporu a zpětnou vazbu jak vyučujícím, tak studentům v podobě zpětné vazby k odevzdanému kódu, výstižných reportů, statistik apod.
Cílem je vytvářený systém nasadit do výuky několika předmětů se zaměřením na výuku programování s využitím jazyka Java SE – výsledky diplomové práce tedy budou uplatněny v univerzitním prostředí.
Předpokladem je implementační platforma Java s využitím vhodných frameworků.

Rozsah pracovní zprávy: **50-60 normostran**
Rozsah grafických prací:
Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam doporučené literatury:

*TURNQUIST L. G.Learning Spring Boot 2.0 – Second Edition: Simplify the development of lightning fast applications based on microservices and reactive programming. Packt Publishing, 2017, 370 pp. ISBN978-1786463784.
*PORCELLO E, BANKS A. Learning React: Modern Patterns for Developing React Apps . O'Reilly Media; 2nd Edition, 2020, 310 pp. ISBN: 978-1492051725.

Vedoucí diplomové práce: **doc. Ing. Michael Bažant, Ph.D.**
Katedra softwarových technologií

Datum zadání diplomové práce: **6. listopadu 2020**
Termín odevzdání diplomové práce: **15. května 2021**

L.S.

Ing. Zdeněk Němec, Ph.D. v.r.
děkan

prof. Ing. Antonín Kavička, Ph.D. v.r.
vedoucí katedry

V Pardubicích dne 30. listopadu 2020

Prohlašuji:

Tuto práci jsem vypracoval samostatně. Veškeré literární prameny a informace, které jsem v práci využil, jsou uvedeny v seznamu použité literatury.

Byl jsem seznámen s tím, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorský zákon, zejména se skutečností, že Univerzita Pardubice má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona, a s tím, že pokud dojde k užití této práce mnou nebo bude poskytnuta licence o užití jinému subjektu, je Univerzita Pardubice oprávněna ode mne požadovat přiměřený příspěvek na úhradu nákladů, které na vytvoření díla vynaložila, a to podle okolností až do jejich skutečné výše.

Beru na vědomí, že v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších předpisů, a směrnicí Univerzity Pardubice č. 9/2012, bude práce zveřejněna v Univerzitní knihovně a prostřednictvím Digitální knihovny Univerzity Pardubice.

V Pardubicích dne 20. 5. 2021

Petr Váňa

PODĚKOVÁNÍ

Tímto bych rád poděkoval panu doc. Ing. Michaelu Bažantovi, Ph.D. za vedení práce a rady při vypracovávání práce. Dále děkuji také svým rodičům za umožnění studia a podporu.

ANOTACE

Diplomová práce je zaměřena na vytvoření a nasazení webové aplikace, která bude automaticky kontrolovat odevzdané zdrojové kódy a vyhodnocovat je na základě vytvořených testovacích scénářů pro jednotlivé úlohy.

Teoretická část se věnuje problematice webových aplikací, zejména jsou zde popsány používané technologie a architektury. Také jsou zde rozebrány techniky testování zdrojových kódů.

Praktická část se zaměřuje na analýzu, návrh, implementaci a nasazení aplikace. Dále je zde podrobně popsán systém tvoření testovacích scénářů.

KLÍČOVÁ SLOVA

Spring boot, MySQL, React, webová aplikace, testování zdrojové kódu.

TITLE

Application system for automatic check of submitted source code.

ANNOTATION

Master's thesis is focused on creating and deploying web application. Application checks submitted source codes and evaluates them based on created test scenarios for each assignment.

Theoretical part is focused on problematics of web applications especially there are described technologies and architectures which are commonly used. There are also analyzed techniques used for source code testing.

Practical part is focused on analysis, design, implementation, and deployment of the application. There is also described system of creating test scenarios in detail.

KEYWORDS

Spring boot, MySQL, React, web application, source code testing.

OBSAH

SEZNAM OBRÁZKŮ

SEZNAM TABULEK

SEZNAM UKÁZEK ZDROJOVÝCH KÓDŮ

Seznam zkratek

API	Application Programming Interface
CRUD	Create, read, update, delete
CSS	Cascading Style Sheets
CORS	Cross-origin resource sharing
DAO	Data Access Object
DDL	Data Definition Language
DI	Dependency Injection
DS	DispatcherServlet
GUI	Graphical User Interface
HTML	Hyper Text Markup Language
IoC	Inversion of Control
IoT	Internet of things
JDK	Java Development Kit
JPQL	Java Persistence Query Language
JSON	JavaScript Object Notation
JTW	JSON Web Token
JVM	Java Virtual Machine
MVC	Model-view-controller
ORM	Object-Relational Mapping
PDF	Portable Document Format
REST	Representational State Transfer
SOAP	Simple Object Access Protocol
SQL	Structured Query Language
W3C	World Wide Web Consortium
WSDL	Web Services Description Language

ÚVOD

Diplomová práce je zaměřena na vytvoření systému, který bude umožňovat automatickou kontrolu odevzdaných zdrojových kódů. Cílem je navržení, implementace a nasazení webové aplikace do výuky několika předmětů se zaměřením na základy programování. Základní funckionalita zahrnuje tvorbu a kontrolu testovacích scénářů pro jazyk Java SE, vůči kterým budou zdrojové kódy testovány, správu předmětů a jejich cvičení, správu uživatelů a poskytnutí zpětné vazby vyučujícím i studentům k odevzdaným kódům v podobě výstižných reportů a statistik. Při vytváření testovacích scénářů musí být zajištěna podpora pro práci s primitivními i referenčními datovými typy.

Výsledná aplikace se skládá z klientské a serverové části. Klientská část neboli frontend je aplikace naprogramovaná v jazyce JavaScript s použitím frameworku React, která zajišťuje generování HTML stránek, interakci s uživatelem a serverem. Vzhled webu je upravován pomocí CSS. Serverová část se skládá z backendu a databáze. Backend zahrnuje aplikaci vyvíjenou v jazyce Java s použitím frameworku Spring Boot, která zajišťuje přijímání, zpracování a odpovídání na požadavky a komunikaci s databází pomocí frameworku Hiberante. Jako databázový systém je použit MySQL.

První část práce popisuje základní koncept testování zdrojových kódů se zaměřením na vybrané metody a techniky používané při testování. Dále je zde rozebrána problematika webových aplikací, zejména popis používaných technologií a vybraných softwarových architektur.

Druhá část je nejprve zaměřena na analýzu podobných řešení a návrh aplikace. Následně je zde rozebrána implementace serverové i klientské části aplikace s podrobným popisem implementace testovacího systému a závěrem je popsán postup nasazení aplikace.

1 TESTOVÁNÍ ZDROJOVÝCH KÓDŮ

Tato kapitola je zaměřená na problematiku testování zdrojových kódů, zaměřuje se na obecný koncept testování a na specifické způsoby testování a jejich techniky.

1.1 Koncept testování zdrojových kódů

Testování zdrojových kódů je proces při, kterém je zjišťováno, zda software splňuje požadavky a neobsahuje chyby a nedostatky. Správné testování také pomáhá vést ke stabilitě kódu, jelikož žádný nový commit („zásah do kódu“) není nasazen bez toho, aniž by úspěšně prošel vytvořenými testy. U přístupu vývoje řízeného testy je dokonce nejdříve napsán test, který sám o sobě dostačuje na selhání testu. Dále jsou prováděny změny v kódu do té doby, než bude test úspěšný. Tento cyklus se poté opakuje, kdy jsou nejdříve psány testy, dokud jeden z nich neselže a pak jsou opět prováděny změny kódu, dokud testy nejsou úspěšné.

[5], [53]

1.2 JUnit testy

JUnit je framework pro psaní jednotkových testů, které slouží pro kontrolu správnosti kódu. Nejnovější verzí je v současné době JUnit5. Jméno jednotkové testy je odvozeno od způsobu testování, jeden test, v tomto případě myšleno jako jedna testovací metoda, je vždy zaměřena pouze na jednu věc. Dohromady pak tvoří celek pomocí, kterého je možno otestovat veškeré funkce.

Důležité je, aby každá testovací metoda byla samostatná a nezávislá. Názorná ukázka tohoto konceptu bude představena na následujícím příkladu. Mějme:

- Třídu *A*, obsahující metody:
    - *B*,
    - *C*.
- Testovací třídu *X*, obsahující metody:
    - *Y* – testuje metodu *B*,
    - *Z* – testuje metodu *C*.

Musí platit, že průběh jedné testovací metody nesmí ovlivnit průběh druhé a opačně. Také nesmí nastat případ, kdy chyba v metodě *B* ovlivní výsledek v metodě *Z*. To samé platí i pro metody *C* a *Y*.

Testovací třída

Testovací třída je taková, která není abstraktní, není privátní, má jeden konstruktor a obsahuje alespoň jednu testovací metodu. Může obsahovat dva typy metod:

- testovací,
- lifecycle.

Oba typy metod nesmí být abstraktní a nesmí vracet hodnotu. Jejich deklarace probíhá buď lokálně v dané třídě, nebo jsou děděny z předků nebo rozhraní.

Lifecycle metody

Všechny tyto metody jsou děděny z předka, pokud nejsou přepsány. V tomto případě se pak označují těmito anotacemi:

- @BeforeAll – Statická metoda provede se pouze jednou a jako první před všemi ostatními metodami. Slouží především pro inicializaci statických atributů, které je výhodné inicializovat jednou.
- @BeforeEach – Provede se před každou testovací metodou. Užitečná například pro nastavování instančních atributů před provedením každé metody.
- @AfterEach – Provede se po každé testovací metodě. Slouží pro úklid instančních atributů/zdrojů.
- @AfterAll – Statická metoda provede se jako poslední po všech metodách. Většinou se používá na uklízení.

Testovací metody

Jsou všechny metody, které kontrolují správnost kódu. Označují se anotacemi:

- @Test – Označení pro běžnou testovací metodu. Do argumentu *expected* je možno nastavit očekávání vyhození specifikované výjimky, v případě že nastane test bude úspěšný.
- @RepeatedTest – Zpracovává test podle specifikovaného počtu opakování. Chovaní jednotlivého testu je stejné jako u @Test.

- @ParametrizedTest – Umožňují provést test s různými parametry. Musí zde být uveden zdroj, který obsahuje alespoň jeden argument. Test a jeho je prováděn zvlášť pro každý argument.
- @TestTemplate – Označuje šablonu pro testovací příklady. Příkladem @TestTemplate jsou vbudované @ReapeatedTest a @ParametrizedTest.
- @TestFactory – Označení pro dynamické testy, které jsou generovány za běhu programu.

Další anotace

Zde jsou popsány další běžně užívané anotace:

- @TestMethodOrder – Zajistí vykonání metod v určeném pořadí.
- @DisplayName – Umožňuje deklarovat vlastní název pro třídu nebo metodu, které se zobrazí na výstupu.
- @DisplayNameGenerator – Implementováním třídy *DisplayNameGenerator* umožňuje vytvářet název podle specifikovaných pravidel. Příkladem výchozí implementace je ReplaceUnderscores, která nahradí všechna podtržítka mezerami.
- @Disabled – Používaná pro deaktivování testovacích tříd nebo metod.
- @Timeout – Po překročení zadaného času je test označen jako chybný. Čas je specifikován přes argument *timeout*.
- @Tag – Používaná pro označení metody nebo třídy tagem pro filtrování testů.

Vyhodnocení testů

Ve většině případů se vyhodnocení testu určí pomocí tvz. assertions. JUnit Jupiter přišel se třídou *Assertions*, ve které se nacházejí statické metody, pomocí nichž se určuje úspěšnost testů. Většina metod v této třídě je založena na principu porovnání očekávané hodnoty / očekávaného objektu s aktuální hodnotou / aktuálním objektem. Příkladem takové metody je *assertEquals*. Další metody jsou např.:

- *assertArrayEquals* – Porovnává pole na základě hodnot/objektů v poli. V případě použití metody *assertEquals* na pole se porovnávají jejich reference.
- *assertNull* – Porovná rovnost objektu s hodnotou null.
- *assertSame* – Označí test za úspěšný, pokud jsou objekty stejné na základě jejich reference.

- *assertTrue* – Označí test za úspěšný, pokud je předaná hodnota true.
- *assertNotEquals*, *assertNotNull, assertNotSame, assertFalse* – Záporné assert metody.
- *fail* – Označí test za chybný.

Žádná testovací metoda nespecifikuje, kolik assert metod by měla obsahovat. Klíč k čistému kódu není přímo o počtu, ale spíš o použití. Každá metoda by měla testovat pouze jednu akci, ta pak může být ověřena více assert metodami.

[18]

1.3 Integrační testy

Integrační testování je druhý level softwarového testovacího procesu po unit testech. V tomto případě jsou komponenty testovány ve skupině. Zjištění funkčnosti jednotlivých komponent by měla být zajištěna pomocí unit testů. Dalším krokem je pak integrační testování, kde cílem je narazit a odhalit chyby, které vznikaní při interakci integrovaných komponent.

Postup pro integrační testy

Základním předpokladem pro vytváření integračních testů je kompletní otestování pomocí unit testů.

1. Příprava plánu integračních testů.
2. Návrh testových scénářů, případů a skriptů.
3. Provedení samotného testování a záznam chyb.
4. Vypořádání se s chybami a opětovné testování.
5. Opakování kroku 3 a 4, dokud není integrace úspěšná.

Důvody integračních testů

I přesto že všechny komponenty jsou testovány pomocí unit testů, stále mohou nastat případy, kdy vznikne chyba kvůli níže zmíněným důvodům:

- Každá komponenta může být navržena a implementována někým jiným a každý může použít rozdílnou programovací logiku. Je nezbytné zkontrolovat, zda logika implementovaná vývojářem odpovídá očekáváním.
- Interakce s nástroji/komponentami třetí strany.
- Interakce s databází.

- Při vývoji může často docházet ke změně nebo k přidání požadavků. V případě že vývojář nasadí změny bez unit testování, mohou nastat chyby a v takové situaci jsou integrační testy velmi důležité.

Typy způsobu implementace integračních testů

Pozitivní způsob integračního testování je založen na základě toho, že se ověřuje správnost chování při předložení validních datových sad. Jednoduchým příkladem může být vstupní pole, které přijímá jednu nezápornou číslici. Pozitivní testování pak ověřuje, zda systém správně přijímá čísla 0-9.

Negativní způsob integračního testování je založen na základě toho, že ověřuje správnost chování při předložení nevalidních datových sad. Příkladem může být již výše zmíněné vstupní pole, které přijímá nezápornou číslici. Negativní testování ověřuje, zda systém správně neakceptuje např. jiné znaky než čísla, záporné hodnoty, prázdnou hodnotu nebo dvou a více ciferná čísla.

Typy integračních testů

Inkrementální integrační testování je přístup, ve kterém se moduly přidávají postupně po jednom, nebo i po více podle potřeby. V zásadě se začíná se dvěma moduly, které jsou otestovány a k nim se postupně přidávají a testují další. K Inkrementálnímu integračnímu testování se používají následující přístupy:

- Top-Down,
- Bottom-Up,
- Hybrid.

Top-Down přístup se vypořádává s testováním tak, že vyšší level moduly (rodiče) jsou postupně testovány s nižšími level moduly (potomky). Díky této metodě mohou být hlavní nedostatky odhaleny a opraveny jako první. Výhody a nevýhody:

+ Kritické moduly jsou testovány první.
+ Možnost brzkého prototypu aplikace.
- Identifikace chyb je složitá.
- Struktura může být komplikovaná.
- Nižší level moduly nejsou testovány dostatečně.

Bottom-Up přístup je opakem výše zmíněného Top-Up, testuje postupně nižší level moduly s vyššími level moduly. Výhody a nevýhody:

+ Identifikace chyb je snadná.
+ Není třeba čekat na vývoj všech modulů.
- Kritické moduly jsou testovány jako poslední.
- Není zde možnost brzkého prototypu aplikace.

Hybridní přístup kombinuje přístupy Top-Down a Bottom-Up. V tomto přístupu se provádí testování vyšších level modulů s nižšími a zároveň nižší level moduly s vyššími. Výhody a nevýhody:

+ Šetří nejvíce času.
+ Poskytuje kompletní testování všech modulů.
- Komplikovaná metoda.
- Vyžaduje více režie, jelikož proces probíhá z obou stran najednou.

Neinkrementální integrační testování je přístup, který se používá v případě, kde je velmi komplexní tok dat a je těžké určit, kdo je rodič a kdo je potomek. Využívá tzv. Big bang metodu. V tomto přístupu se testování provádí pomocí integrace všech modulů, nebo skoro všech modulů najednou, na rozdíl od inkrementálních metod, kde se moduly přidávají postupně.

[14], [15], [16], [54]

1.4 Selenium testy

Selenium je jeden z nejrozšířenějších open source frameworků pro automatizované testování webového rozhraní. Podporuje různé operační systémy, webové prohlížeče a programovací jazyky jako jsou:

- Operační systémy – Windows, Linux, Macintosh, Android, iOS.
- Webové prohlížeče – Google Chrome, Mozilla Firefox, Safari, Edge.
- Programovací jazyky – Java, C#, Python, PHP.

Selenium podporuje paralelní zpracování, což redukuje potřebný čas a zvyšuje tak efektivitu testů. Může být integrován s frameworky jako jsou Ant a Maven pro kompilaci zdrojového kódu. Také umožňuje práci s frameworky jako JUnit, TestNG pro testování a generování

reportů. Další výhodou je, že vyžaduje tolik zdrojů jako jiné frameworky pro automatizované testování. Selenium web driver nepotřebuje žádnou instalaci, testovací skripty interagují přímo s webovým prohlížečem. Příkazy pro testování jsou rozděleny do různých tříd, což zjednodušuje jejich pochopení a práci s nimi. Selenium je integrován s WebDriver API.

Selenium nepodporuje testování desktopových aplikací a webových služeb jako jsou SOAP a REST. Sám o sobě nemá schopnost vytvářet reporty, ke které potřebuje pluginy jako JUnit, nebo TestNG. Vytvoření testovací prostředí a efektivních testů může zabrat více času než u některých jiných nástrojů jako UFT, RTF.

WebDriver

Selenium WebDriver je jeden z nejdůležitějších nástrojů. Od verze Selenium 2.0 je integrován s WebDriver API, které poskytuje jednodušší rozhraní. Architektura Selenium WebDriver je složena ze čtyř základních komponent:

- Selenium Language Bindings / Selenium Client Libraries,
- JSON Wire Protocol,
- Browser Drivers,
- Browsers.

Selenium Client Libraries poskytují knihovny pro podporu různých programovacích jazyků. JSON Wire protocol poskytuje mechanismus na přesměrování dat mezi serverem a klientem. Browser Drivers jsou ovladače specifické pro každý webový prohlížeč. Zajištují bezpečné připojení s webovým prohlížečem a nijak neokrývají vnitřní logiku fungování prohlížeče. Browsers jsou všechny webové prohlížeče, které podporuje Selenium WebDriver.

Automatizované testování

Automatizované testování používá speciální nástroje pro automatické provedení manuálně vytvořených testových scénářů bez nutnosti zásahu uživatele. Tyto nástroje umožňují přístup k datům, kontrolu provedení a porovnání aktuálních výsledků s očekávanými výsledky.

Automatizované testování webových aplikací pomocí Selenium WebDriver probíhá v následujících krocích:

1. Generování HTTP požadavku, který je zaslán do browser driveru pro každý Selenium příkaz.
2. Driver obdrží požadavky přes HTTP server.

3. HTTP server na základě instrukcí určí všechny kroky, které budou vykonány v prohlížeči.
4. Výsledný status je pak zaslán HTTP serveru, který ho zasílá zpět automatizovanému skriptu.

[28], [29]

1.5 Použití mock objektů

Mock objekt je vytvořen umělou implementací třídy, nebo rozhraní. Umělou implementací je myšleno vlastní definování výstupů metod na základě potřeby. Jednoduchým příkladem mock objektu může být třída zajišťující data z databáze. V reálné aplikaci jsou používána opravdová data z databáze, ale pro testování je možné použít mock objekt, který simuluje zdroj dat a zajistí, že testovací podmínky budou vždy stejné, tudíž pokud by metoda měla například vracet list zaměstnanců, vždy vrátí stejný předdefinovaný seznam nehledě na to, co je opravdu uloženo v databázi.

Mock testování je přistup k jednotkovým testům, který umožňuje testovat modul izolovaně od jeho závislostí na jiných modulech. Účelem testování pomocí mock objektů je izolace a zaměření se pouze na kód, který má být testován, a ne na chování ostatních závislostí, ty by totiž mohly ovlivnit výsledek testu.

Umožňuje vytvářet testovací scénáře bez vedlejších efektů, příkladem může být zasílání notifikací, nebo emailů. Tento typ testů urychluje vývoj a celý proces testování, jelikož při chybném testu zužuje možnosti příčin chyby na maximálně jeden modul, na rozdíl od použití reálných objektů, kde by chyba mohla nastat i v jakémkoliv jiném závislém modulu. Také dále umožňuje testovat kód již od začátku, jelikož není třeba čekat na implementaci závislostí. Dalším důvodem pro testování pomocí mock objektů je vyhnutí se částem kódu, které jsou relativně pomalé, což může být například volání API třetí strany apod.

Osvědčené metody:

- Vytvářet mock objekty pouze na typy, které vývojář vlastní.
- Nevytvářet mock objekt pro získání hodnoty.
- Používání negativních testů, vhodné pro testování error handling („vypořádání se s chybou“).

[21], [45]

Mockito

Mockito je jedním z nejpopulárnějších mock frameworků, který může být spojen s JUnit. Umožňuje psát jednoduché a čisté testy. V současné době za ním stojí největší komunita, co se týče mock frameworků. Mock objekty se dají vytvořit jednoduše pomocí anotace @Mock.

[22]

1.6 Testování černé, bílé a šedé skříňky

Specifikem pro testování černé skříňky je, že tester nemá/nepotřebuje znalost o tom, jakou má software vnitřní implementaci. Černá skříňka je metaforou pro software, do kterého není vidět a je zavřený. Pro vytváření testových scénářů je nutnost znát pouze GUI (Graphical User Interface). Přístup k testům zahrnuje zkušební techniky a metody hádání chyb. Testeři se zaměřují pouze na vstupy a výstupy, to velmi zjednodušuje vytváření testovacích scénářů a doba přípravy je velmi krátká. Nepovažuje se za testování algoritmů. Testery v tomto případě mohou být testeři, vývojáři i koncoví uživatelé. Pokud tester nemá vědomí interní implementace poté nijak ovlivněn a přistupuje k testování neutrálně bez perspektivy, kterou mají vývojáři kódu, což pomáhá vést k vytvoření testů a odhalení chyb, na kterou by přímo vývojáři nemuseli přijít. Velkou nevýhodou je hledání poslepu, kdy testy neukazují přímo na část kódu, která vede k problémům. Tato metoda je možná efektivně použít na malé kusy softwaru. Při testování komplexních softwarů je tato metoda velmi neefektivní a časově náročná.

Specifikem pro testování bílé skříňky je, že tester má přístup ke všem interním datovým strukturám a algoritmům. Pro vytváření testovacích scénářů je znalost celého kódu nezbytná. Tester nejen vidí, ale může i manipulovat s kódem jako součást testovacího procesu. Bílá skříňka je v tomto případě naznačována jako průhledná a otevřená, kde je vevnitř vše vidět a může být i změněno. Vhodné pro testování algoritmů. Testování mohou provádět pouze testeři a vývojáři. Provádění těchto testů při vývoji může vést k odhalení místo, které mohou v budoucnu vést k chybám. S přístupem ke kódu tester může nejen testovat, ale i optimalizovat kód pro dosažení lepší výkonosti. Oproti metodě černé skříňky odhalení skrytých chyb je mnohem méně náročné. Nevýhodou je nutnost vysoké znalosti systému a kompetenci programování. Rychlost testování se odvíjí podle délky zdrojového kódu, v případě velmi dlouhého kódu testování zabere velmi dlouhou dobu.

Testování šedé skříňky je kombinací testování bíle a černé skříňky. Šedá skříňka je v tomto případě naznačována jako průhledná a zavřená, kde je vevnitř vše vidět, ale nemůže být změněno. Pro vytváření testovacích scénářů je nezbytná alespoň částečná znalost kódu. Oproti testování černé skříňky tester, díky své znalost může navrhnout mnohem lepší testovací scénáře. Nepovažuje se za vhodnou metodu pro testování algoritmů.

[4], [11], [12]

2 WEBOVÉ APLIKACE

Tato kapitola je zaměřena na problematiku webových aplikací, zejména na používané technologie a architektury.

2.1 Technologie

V této podkapitole jsou rozebrány vybrané technologie, které jsou používány při vývoji webových aplikací.

2.1.1 Java

Java je vysokoúrovňový objektově orientovaný programovací jazyk. Je to jeden z nejvíce používaných jazyků současné doby. Jedním z hlavních důvodů vysoké rozšířenosti je její nezávislost na platformě, kde stačí, aby na daném systému byl nainstalován JVM (Java Virtual Machine), který ten dokáže spustit zkompilovaný bytecode. Dalšími důvody mohou být její bezpečnost, škálovatelnost, správa paměti a multi-threading. Využití při vývoji je například pro desktopové, webové, mobilní, cloud a IoT aplikace.

[19], [46]

2.1.2 Spring

Spring je framework postavený na jazyku Java, který nabízí obsáhlou podporu pro vývoj Java aplikací. Zajišťuje infrastrukturu celé aplikace a předává hlavní zaměření programátora na vývoj. Propaguje dobré programátorské praktiky, tím že umožňuje POJO programovací model. Pomáhá vytvořit kvalitní a výkonný kód, jenž je jednoduše testovatelný a znovupoužitelný. Je postavený na dobře navržené MVC architektuře.

Dependency injection

Volně přeloženo jako „vkládání závislostí“, je velmi důležitý koncept Spring Frameworku. Je to jeden z mnoha řešení principu Inversion of Control (IoC).

IoC je zaměřen na odstranění závislostí mezi třídami a dosažení volných vazeb. Snaha je také dosáhnout toho, aby každá třída byla soustředěna pouze na svojí hlavní práci. Důležité je si uvědomit, že pokud mezi sebou objekty navzájem komunikují, aby provedli nějakou funkcionalitu, tak to ještě neznamená, že jsou jejich třídy na sebe tímto způsobem závislé. Závislost vzniká, pokud třída „A“ zajišťuje vytvoření a životnost objektu třídy „B“. Odvrácením

takového stavu se provede tak, že někdo jiný zajistí dodání objektu třídy „B“ do třídy „A“, což může zajistit např. tovární třída. Tento princip zajišťuje jednodušší testování, udržovatelnost a rozšiřitelnost.

DI vzor zahrnuje třídy typu client, service a injector. Client využívá službu, service zajištuje službu pro klienta a injector dodává objekt typu service klientovi. Metody vkládání objektu jsou contructor injection, který vkládá pomocí konstruktoru, setter injection, který používá set metodu a method injection, který vkládá přes metodu.

Autowiring je funkce Spring Frameworku, která umožňuje implicitní vkládání objektu, které automaticky použije constructor, nebo property injection. Touto metodou dosáhneme kratšího a jednoduššího kódu, jelikož stačí použít pouze anotaci @Autowired.

Beans

Bean je klíčový koncept Spring Frameworku. Jsou to objekty spravované pomocí IoC kontejneru. Nejčastěji jsou definovány pomocí anotací, nebo také přes XML soubor. Nejvíce používané Spring Bean anotace:

- @Component – Obecná anotace, která automaticky vytváří bean objekt dané třídy. Ve výchozím stavu se jméno shoduje s názvem třídy, pouze první písmeno je malé.
- @Repository – Označuje třídy v DAO vrstvě, které zajištují CRUD operace. Povoluje „automatic persistence exception translation“, což je automatický překlad persistence vyjímek.
- @Service – Anotace pro třídy v service vrstvě, ve kterých je implementována byznys logika.
- @Controller – Označuje třídy (controllers) v Spring MVC. Pro RESTful webové služby se používá anotace @RestController.
- @Configuration – Říká, že daná třída obsahuje jednu, nebo více metod označených anotací @Bean.

[32], [33], [43]

2.1.3 Spring Boot

Spring Boot je nadstavba Spring Frameworku, jedná se o zjednodušenou platformu pro vývoj produkčně schopných aplikací. Umožňuje rychlejší vývoj díky minimalizování nutných nastavení pro chod aplikace a také díky tomu, že nevyžaduje žádnou složitou konfiguraci přes

XML soubory. Automaticky konfiguruje Spring a knihovny třetích stran. Nabízí „starter“, ten by se dal podobou přirovnat např. k instalačnímu procesu, pomocí něhož je možné jednoduše začít nový projekt a přidat potřebné dependencies.

Základní anotace:

- @EnableAutoConfiguration – Automaticky nastavuje aplikace na základě přidaných dependencies.
- @ComponentScan – Při startu aplikace zajistí naskenování všech beans a deklarací balíčků.
- @SpringBootConfiguration – Alternativa anotace @Configuration, oproti které umožňuje automatické vyhledání konfigurace, může být užitečné např. u testů. Měla by být v aplikaci pouze jednou
- @SpringBootApplication – Označuje třídu, která je vstupním bodem aplikace a která by měla obsahovat metodu main. Zahrnuje všechny tři výše zmíněné anotace. Nejčastěji je používaná pouze tato anotace a samostatně.

Application properties

Soubor application.properties je používán na uchovávání properties („vlastností“) pro běh aplikace v různých prostředích. To je řešeno pomocí profilů, každý profil může mít svoje unikátní hodnoty. V souboru jsou například uloženy properties na připojení do databáze. Tak že pokud by se aplikace spouštěla na dvou místech a používala dvě různé databáze, stačí mít pouze jeden unikátní soubor se všemi hodnotami, pak už stačí jen správně nastavit profil při spuštění.

Pom

Project object model je konfigurační XML soubor, který používá Apache Maven. Tento nástroj slouží pro automatizaci buildů, správu, řížení aplikace a také výrazně ulehčuje import knihoven do projektu. V pom jsou uloženy základní informace o aplikaci jako je název aplikace, popis, verze, groupId, artifactId, Spring Boot verze, Java verze, dále jsou tam všechny dependencies a build. Nejdůležitější při vývoji jsou pro programátora především dependencies, které potřebuje pro používání funkcí různých knihoven. Pakliže potřebuje novou knihovnu, kterou nepřidal přes starter, stačí si jednoduše vyhledat Maven dependency v Maven repozitáři, najít požadovanou verzi a zkopírovat dependency do pom souboru.

V následující ukázce je ukázka jedné dependency v pom souboru.

```
<dependencies>
    …
    <dependency>
        <groupId>org.springframework.security</groupId>
        <artifactId>spring-security-core</artifactId>
        <version>5.2.5.RELEASE</version>
    </dependency>
    …
</dependencies>
```

Ukázka zdrojového kódu 1: POM dependency

[20], [32], [37], [41]

2.1.4 JavaScript

JavaScript je velmi rozšířený a populární programovací jazyk, který se využívá zejména při vývoji webových aplikací, avšak je používán i pro programování desktopových nebo serverových aplikací. Vyznačuje se dynamickým typováním.

Dříve byl JavaScript u vývoji webových aplikací využíván zejména pro doplnění frontendu aplikace neboli stránky, se kterou uživatel interaguje. Dopomáhal zejména tím, že zařizoval asynchronní zprávy se serverem, ze kterého získal dynamický kontext stránky, který poté zobrazil na stránce. Hlavní výhodou této techniky bylo eliminování nutnosti obnovovat celou stránku, protože se načetla pouze část, která byla požadována. Dále sloužil také pro nejrůznější efekty a animace. V dnešní době již čistý JavaScript není tak běžný a používají se zejména jeho nadstavby. Příkladem jazykové nadstavby je TypeScript od společnosti Microsoft. Rozšiřuje JavaScript o statické typování a možnost objektově programovat. TypeScript je kompatibilní s JavaScriptem, jelikož se do něho sám kompiluje a tím umožňuje používat i jakákoliv jeho další rozšíření. Příkladem frameworků mohou být React a Angular. Angular je velmi populární open-source framework spravovaný společností Google.

[2], [17], [44]

2.1.5 React

React je knihovna JavaScriptu od Společnosti Facebook. Slouží k vytváření interaktivních uživatelských rozhraní. Je založen na tzv. single-page architektuře, ta se vyznačuje tím, že přepisuje stávající stránku a pouze mění a přidává nový obsah na základě požadavků uživatele.

Virtual DOM

DOM (Document Object Model) je rozhraní, které slouží k dynamickému přístupu a změně kontentu, struktury a stylu dokumentu. HTML DOM je objektový model uspořádaný jako stromová struktura a programové rozhraní pro HTML, definuje:

- HTML elementy jako objekty.
- Atributy všech HTML elementů.
- Přístupové metody ke všem HTML elementům.
- Události pro všechny HTML elementy.

React přišel s konceptem zvaným Virtual DOM, který je virtuální reprezentací skutečného DOM a se kterým se při změnách stavu synchronizuje. Tento proces je nazván reconciliation, ke kterému používá The Diffing Algorithm. Při něm dochází k porovnání DOM s tím virtuálním. Efektivně nalezne pouze objekty, které byly změněny a promítne změny ve skutečném DOM.

JSX

JSX (JavaScript XML) rozšiřuje syntaxi pro JavaScript. Umožňuje a zjednodušuje psaní HTML v Reactu. Pro vytvoření elementu již není potřeba použít *createElement* nebo *appendChild* metody. JSX automaticky konvertuje HTML tagy na React elementy. Rozdílné použití pro vytvoření odstavce s a bez JSX:

- S JSX – const withJSX = Odstavec. ;
- Bez JSX – const withoutJSX = React.createElement('p', {}, 'Odstavec.');

Důležité je si uvědomit, že „ Odstavec. “, sice působí jako čisté HTML, ale ve skutečnosti je to zjednodušený zápis, který se převede na JavaScript kód.

Dále díky JSX je možné jednoduše vkládat výrazy pomocí složených závorek. Ten může nabývat jakémukoliv validnímu JavaScript výrazu např. {promenna} , 1 + 1 = {1+1} .

Components

Komponenty (components) jsou nezávislé a znovupoužitelné stavební bloky, které jsou poskládané do stromové struktury a tvoří celou React aplikaci. Komponenty existují buď funkcionální, nebo třídní.

Funkcionální komponenty jsou odlehčené a bezstavové, jejich hlavním úkolem je přijmout data a zobrazit je v nějaké formě. Obsahují pouze funkci, která může přijímat props jako argument a nejčastěji vrací JSX kód. Nemohou použít React lifecycle metody.

Třídní komponenty jsou stavové a zapouzdřeny ve třídách, jejich hlavním úkolem udržovat stav a implementovat logiku. Musí dědit od React.Component a musí obsahovat metodu *render*, která nejčastěji vrací JSX kód. Na rozdíl od funkcionálních mohou použít React lifecycle metody.

State

State je objekt, který obsahuje každá třídní komponenta a je v ní uložen stav dané komponenty. Ke stavu se přistupuje pomocí *this.state*. Oproti tomu pro změnu stavu je nezbytné použít metodu *this.setState*, která zajistí, že se komponenta dozví o změně, zavolá *render* a ostatní lifecycle metody.

Props

Props je neměnitelný objekt, který obsahuje každá komponenta. Slouží k předávání hodnot mezi komponentami. Používá k tomu stejnou syntaxi jako HTML atribut „<Komponenta argument='hodnota' />“. Je možné předat více argumentů různých datových typů, které se pak všechny zapouzdří do jednoho objektu props. Přistupuje se k němu pomocí *this.props*.

Component lifecycle

Každá komponenta má svůj lifecycle (životní cyklus), ve kterém je možno monitorovat a manipulovat s komponentou. V prvním cyklu zvaném Mounting se vkládají elementy do DOM, React postupně volá metody *constructor*, *getDerivedStateFromProps*, *render* a *componentDidMount*. Druhý cyklus Updating nastává v případě, že se komponentně změnil state nebo props, jsou zavolány metody *getDerivedStateFromProps*, *shouldComponentUpdate*, *render*, *getSnapshotBeforeUpdate* a *componentDidUpdate*. Poslední cyklus Unmounting nastává v případě, kdy je komponenta odebraná z DOM. Volá se pouze metoda *componentWillUnmount*.

Často používané lifecycle metody:

- constructor(props) – Nepovinná metoda, slouží pro inicializaci komponenty, nastavení stavů a jiných hodnot. V konstruktoru platí vyjímka, že state se nenastavuje přes metodu *this.setState*, ale přiřadí se přímo do this.state.
- render() – Jediná povinná metoda, nejčastěji vrací React elementy vytvořené přes JSX, ale může vrácet i Array, String, numbers, Boolean, null. Měla by pokaždé vrátit stejný výsledek a nikdy by neměla měnit state komponenty.
- componentDidMount() – Nepovinná metoda, slouží zejména pro načtení dat ze serveru, je zde možné zavolat metodu *this.setState*, při změně stavu se znovu provede render.
- componentDidUpdate(prevProps, prevState, snapshot) – Nepovinná metoda, může být použita pro načtení dat ze serveru, lze zde zavolat metodu *this.setState*. Důležité je tento kód obalit podmínkou, která zamezí průchodu v případě, kdy není nutné nic načítat nebo měnit. To pomůže předejít zbytečným dotazům na server nebo nekonečným cyklům. Neprovede se v případě, když metoda shouldComponentUpdate() vrací false.
- componentWillUnmount() – nepovinná metoda, provádí se těsně před tím, než je komponenta odebrána a smazána. Slouží pro nezbytný úklid. Neměla by volat metodu *this.setState*, jelikož se renderování této komponenty už nikdy neuskuteční.

Zřídka používané lifecycle metody:

- static getDerivedStateFromProps(props, state) – Nepovinná metoda, používá se velmi ojediněle v případě, pokud je state závislý na změnách props v čase, vrací buď změněný state, nebo null pro žádnou změnu.
- shouldComponentUpdate(nextProps, nextState) – Nepovinná metoda, implementuje se pouze za účelem testování, jelikož není vhodné v žádném jiném případě zamezovat renderování, protože to může vést k chybám. Vrací buď true, nebo false.
- getSnapshotBeforeUpdate(prevProps, prevState) – Nepovinná metoda, slouží pro zachování nějaké informace těsně před tím, než výstup renderování promítne do DOM. Návratová hodnota je předávána metodě *componentDidUpdate* jako parametr.¨

Lifecycle metody pro zachycení vyjímky:

- static getDerivedStateFromError(error) – Nepovinná metoda, je zavolána v případě, kdy komponenta potomka vyhodí error. Parametrem je předána daná chyba. Návratová hodnota by měla vracet hodnotu pro změnu state.

- componentDidCatch(error, info) – Podobně jako výše, nepovinná metoda, která se volá při chybě komponenty potomka. V parametrech obdrží error a informaci, která komponenta ho vyhodila. Slouží například pro logování.

[24], [27], [49]

2.1.6 MySQL

MySQL je jeden z nejpopulárnějších relačních databázových systémů. Je vyvíjen a distribuován společností Oracle Corporation pod licencí open source. Slouží ke správě a přístupu k datům uložených v databázi. Interakce je prováděna pomocí SQL dotazů. Čímž uživatele oprošťuje o svojí systémovou logiku jako například jak a kam si data ukládá.

Databáze je tvořená schématy, které obsahují datové struktury, pro uživatele tabulky, ve kterých jsou uchovávána data. Jaká datová struktura je opravdu použita pro uchování dat tabulky záleží na databázovém enginu. MyISAM engine je velmi rychlý a používá datovou strukturu heap, avšak nepodporuje transakce, ACID (atomicita, konzistence, izolace a trvalost) a cizí klíče. Nejvíce se hodí pro read-only, nebo read-mostly systémy jako jsou datové sklady a webové aplikace. InnoDB engine používá datovou strukturu B-tree (B-strom), podporuje transakce, ACID i cizí klíče. Je nejvíce vhodný pro velké databáze s relačními daty.

Struktura tabulky je tvořena sloupci, které mají definované vlastnosti jako datový typ, velikost, unikátnost, nulovost, defaultní hodnotu, zda je primární nebo cizí klíč a další. Řádky tabulky představují jeden záznam dat. Tabulky mezi sebou vytváří vztahy/relace, které propojují záznamy jedné tabulky se záznamy z druhé tabulky. Vztahy jsou pak děleny podle různých vlastností:

- stupeň vztahu,
- kardinalita,
- parcialita.

Stupeň vztahu vzniká na základě počtu tabulek zapojených do relace. V případě, že tabulka je napojena sama na sebe vzniká tzv. unární vztah. Nejčastějším stupněm je binární vztah, kdy vzniká relace mezi dvěma tabulkami. Velmi zřídka se objevují n-ární relace, které vytváří relaci mezi třemi, nebo více tabulkami.

Kardinalita určuje počet možných relací je možno vytvořit pro danou entitu. Relace 1:1 je nejjednodušší typ, ve kterém každý jednotlivý záznam z tabulky *A* tvoří relaci na právě jeden

unikátní záznam z tabulky *B*. Relace 1:N udává vztah, kdy každý jednotlivý záznam z tabulky *A* tvoří relaci na N (jeden a více) záznamů z tabulky *B*. Relace M:N je speciální případ v relačních databázích, kdy jeden záznam tabulky *A,* tvoří relaci na N záznamů tabulky *B* a opačně. Toto je speciální případ, protože vytvoření vztahu M:N v relačních databázích lze dosáhnout pouze pomocí dvou 1:N vztahů a vytvořením tzv. propojovací tabulky.

Parcialita určuje povinnost, nebo nepovinnost vztahu. Příkladem povinného vztahu na obou stranách je vztah 1:N. Nepovinnost vztahu na jedné či druhé straně je pak přidána pomocí nuly. Nepovinný vztah na obou stranách na stejném příkladu by měl podobu 0..1:0..N.

Structured Query Language zkráceně SQL je standardizovaný jazyk pro manipulaci a přístup k databázi. Příkazy jazyka se dělí na čtyři základní skupiny:

- Data definition language (DDL) – Příkazy, které slouží pro definici dat.
    - CREATE – Slouží pro vytvoření např.: databáze/schématu, tabulky, indexu, pohledu atd.
    - ALTER – Slouží pro modifikaci např.: databáze/schématu, tabulky, indexu, pohledu atd.
    - DROP – Slouží pro smazání např.: databáze/schématu, tabulky, indexu, pohledu atd.
- Data manipulation language (DML) – Příkazy, které slouží pro manipulaci s daty.
    - SELECT – Slouží pro vybrání dat z databáze.
    - INSERT INTO – Slouží k vložení nových záznamů do databáze.
    - UPDATE – Slouží k editaci záznamů v databázi.
    - DELETE – Slouží pro smazání záznamů v databázi.
- Data control language (DCL) – Příkazy, které slouží pro správu uživatelských práv a rolí.
    - GRANT – Příkaz pro přidělení uživatelských práv, nebo rolí.
    - REVOKE – Příkaz pro odebrání uživatelských práv, nebo rolí.
- Transactional control language (TCL) – Příkazy, které slouží pro správu databázových transakcí.
    - BEGIN – Označuje začátek transakce.

- COMMIT – Potvrzuje transakci.
- ROLLBACK – Odvolává transakci.

Dalšími důležitými klíčovými slovy jsou např.:

- WHERE – Vytváří restrikci.
- JOIN – Kombinuje záznamy z dvou, nebo více tabulek na základě jejich relace.
- AND, OR, NOT – Logické operátory.
- ORDER BY – Seřazení záznamů na základě parametru/parametrů.
- GROUP BY – Seskupení záznamů na základě parametru/parametrů.

S příchodem ORM (Object-Relational Mapping) frameworků jako je například Hibernate ubývá implementace SQL dotazů ve vrstvě aplikace, jelikož tabulky jsou tvořeny podle vrstvy modelu aplikace a dotazy jsou možné vytvářet pomocí názvů metod. Tato problematika je více popsána v kapitolách 4.3.3 Vrstva modelu a 4.3.4 DAO.

[23], [50]

2.1.7 HTML

Hyper Text Markup Language zkráceně (HTML) je značkovací jazyk pro tvorbu webových stránek. Jeho účel je popsat strukturu stránky za pomocí tagů/elementů. Jednotlivé elementy říkají prohlížeči, jak danou stránku zobrazit. Všechny HTML soubory musejí začínat s deklarací <!DOCTYPE html>. Dále následuje samotný HTML dokument, který začíná tagem <html>, pak následuje tělo celého dokumentu a nakonec tag </html>.

Jako první v tomto těle nejčastěji následuje element <head> (ukončený </head>), který slouží jako kontejner pro metadata. Typicky vně definuje název dokumentu, sadu znaků, linky, přes které například napojuje například skripty nebo CSS soubory a další.

Dále následuje tag <body> (ukončený </head>), který definuje tělo dokumentu. Zapouzdřuje vše, co bude zobrazováno jako výsledná HTML stránka. Základními elementy jsou například:

- <h1> – Definuje nadpis. Číslo za h určuje důležitost nadpisu, kde 1 je nejdůležitější (největší velikost písma), v rozsahu od 1 do 6. Příklad použití <h1>Nadpis</h1>.
- – Definuje odstavec. Příklad použití Odstavec. .

- – Definuje sekci, do které může být vkládán další kontent. Slouží primárně jako kontejner.

Kontent elementu je vždy vkládán vně začínající a končícího tagu, který má před názvem „/“. Dalším způsobem rozšíření jsou HTML atributy, které poskytují dodatečné informace k prvku. Jsou vždy specifikovány vně začínajícího tagu a většinou se zapisují ve tvaru atribut=“hodnota“, výjimkou mohou být boolean atributy, které při specifikaci názvu automaticky mají přiřazenou hodnotu *true*. Příklad <p id=“1“ hidden>Skrytý odstavec . V příkladu byl odstavci přiděleno *id* s hodnotou jedna a atribut *hidden*, který se automaticky nastavil na true a nebude zobrazen na výsledné stránce. *Id* identifikuje jeden unikátní element. Dalším důležitým atributem je například *class*, kterým je přiřazena třída libovolnému počtu elementů. Třídy jsou především vhodné s použitím CSS.

HTML rozlišuje dva základní druhy tagů:

- párové,
- nepárové.

Párové jsou všechny tagy, které se skládají z ukončovacího a startovacího elementu. Příkladem jsou <h1>, , , které byly již výše představeny. Nepárové tagy jsou takové, které nemají ukončovací element a jsou takové, které neobsahují žádný kontent. Příkladem může být například <img>, který je používán pro vložení obrázku na stránku pomocí odkazu. Tag <img> musí obsahovat atribut *src*, který specifikuje cestu k obrázku a *alt*, který specifikuje alternativní text v případě, že obrázek nemůže být zobrazen.

S příchodem s JavaScriptových technologií a jejich frameworků, jako je React ubývá implementace čistého HTML kódu. React například obsahuje pouze jeden HTML soubor, který obsahuje pouze základní strukturu jako <html>, <head> a <body>, ale výsledné tělo je vygenerováno pouze převodem kódu Reactu do JavaScriptu a do HTML.

[48]

2.1.8 CSS

Cascading Style Sheets (CSS) je jazyk používaný pro stylování webových stránek. Popisuje, jak se HTML prvky budou zobrazovat uživateli. CSS vyřešil problém a nedostatek HTML, který nebyl nikdy navržen, tak aby formátoval webovou stránku, pouze její kontent pomocí tagů jako <h1>, . S příchodem HTML 3.2 byly přidány tagy jako <font> a atributy jako color, které dokázaly stylovat vzhled. Jenže správa těchto stránek byla velmi náročná a

neumožňovala znovupoužitelnost, což se ukázalo jako velmi nepraktické u větších projektů. Tento problém vyřešil World Wide Web Consortium (W3C), který vytvořil CSS. Tím bylo odstraněno stylování z HTML stránek. Ušetření mnoha práce a času spočívá ve vytvoření externího CSS souboru s koncovkou „.css“, který umožňuje definici vzhledu prvků v jednom souboru. Napojením více HTML stránek na jeden CSS soubor umožňuje znovupoužitelnost nadefinovaných stylů pro všechny tyto stránky a mnohem jednodušší správu. Vzhled celého webu tak může být upravován změnou pouze jednoho pouze jednoho souboru.

Syntaxe stylu/pravidla spočívá v deklaraci selektoru a deklaračního bloku. Selektor ukazuje na HTML prvek, který bude ovlivněn a deklarační blok obsahuje samotné stylování. To zahrnuje CSS property name (název vlastnosti) a hodnotu. Při vícenásobné deklaraci jsou tyto styly odděleny středníkem. Ohraničení bloku je vytvořeno pomocí složených závorek.

```
p {
    color: red;
    font-family: "Times New Roman";
}
```

Ukázka zdrojového kódu 2: CSS blok

Selektory je možné rozdělit do pěti kategorií:

- Simple selectors – Ukazuje na prvky na základě jména, id, třídy.
- Combinator selectors – Ukazuje na prvky na základě specifického vztahu mezi nimi.
- Pseudo-class selectors – Ukazuje na prvky na základě jejich specifického stavu.
- Pseudo-elements selectors – Ukazuje na prvky a styluje pouze jejich specifickou část.
- Attribute selectors – Ukazuje na prvky na základě jejich atributu, nebo hodnoty atributu.

Simple selektor na základě id je zapsán se znakem # před identifikátorem prvku. Id by mělo být v HTML stránce unikátní, a proto by tento styl měl upravovat vzhled pouze jednoho elementu. Class selector oproti tomu slouží pro stylování více elementů se stejnou třídou, zapisuje se pomocí tečky před názvem třídy. Do simple selektorů také patří stylování přímo HTML elementů, které je zapsáno pouze pomocí názvu elementu bez „<>“. Pro definici stylu nad HTML elementem s danou třídou je použit zápis ve tvaru *element.class*. Seskupením více

selektorů je jejich oddělením čárkou v definici (selektor, selektor, selektor {…}). Speciálním selektorem je „*“, který ukazuje na všechny HTML elementy.

Pro zjednodušení práce stylování webových stránek byly vytvořeny CSS frameworky, ve kterých jsou předdefinované styly elementů pro různá použití. Příkladem CSS frameworků jsou např.: Bootstrap, Foundation, Font Awesome a další. Pro použití je stačí pouze napojit do HTML stránky a popsat elementy podle dokumentace. Obvykle přidáním specifické třídy k prvku.

[47]

2.2 Architektury

V této podkapitole jsou popsány některé vybrané softwarové architektury, které se používají při vývoji webových aplikací.

2.2.1 REST a SOAP

REST

REST (Representational State Transfer) je softwarový architektonický styl pro distribuované prostředí. Základními stavebními kameny REST jsou:

- architektura klient-server,
- bezstavovost,
- jednotné rozhraní,
- využití vyrovnávací paměti,
- vrstvený systém.

Architektura klient-server je jedna z nejvíce používaných síťových architektur, která popisuje nerovný vztah mezi serverem a klientem. Server naslouchá, plní a odpovídá požadavky. Klient zasílá požadavky a přijímá odpovědi. Důležité je také rozdělení klientské a serverové části, kde server má jedno jednotné rozhraní, jedno rozhraní nemusí přímo znamenat, že server je jen jedno zařízení, a může ho používat vícero klientů, a to i napříč různými platformami a různými typy aplikací např. webové, desktopové, nebo mobilní aplikace.

Bezstavovost je další důležitá vlastnost v REST architektuře. Jak bylo zmíněno výše server nemůže být přímo chápán jako jedno zařízení, ale může jich být rovnou několik a tvořit tzv.

serverový cluster. Vzhledem k tomu, že tyto zařízení jsou mezi sebou nezávislé, není vhodné nebo dokonce možné, aby mezi klientem a serverem vznikal nějaký vztah či session. Proto každý klientský požadavek musí být naprosto nezávislý. Server tedy zpracovává pouze požadavek, ze kterého zjistí všechny potřebné informace. Ověření klienta může být například pomocí autorizačních tokenů.

Jednotné rozhraní definují čtyři omezení:

- Identifikace zdrojů v požadavku – Jednotlivé zdroje jsou identifikovatelné v požadavku a nemají žádný vztah k reprezentaci, která je vrácena klientovi. Používají se k tomu např. URI.
- Manipulace se zdroji přes jejich reprezentace – Klient obdrží pouze reprezentaci zdroje, se kterou následně pracuje, ale měl by být schopný jí měnit i mazat, pokud má dostatečná oprávnění.
- Self-descriptive zprávy – Zpráva musí obsahovat dostatek informací, aby jí server mohl zpracovat.
- HATEOAS – Klient nemusí vědět, jak přímo komunikovat se serverem, ale stačí mu pouze porozumění hypermedií, které mu jsou poskytnuty v odpovědi a on se pak pomocí nich může dále navigovat.

Aplikování vyrovnávací paměti umožňuje klientovi ukládat příchozí data ze serveru. Použití této techniky může odlehčit práci serveru a zvýšit propustnost, jelikož nebude muset zpracovávat tolik požadavků. Aby si mohl klient uložit data, musí být označená serverem jako cacheable. Důležitým aspektem dobrého použití je správně určit, která data ukládat. Při špatném zacházení může vzniknout situace, kdy klient pracuje se zastaralými daty z vyrovnávací paměti, která se liší od dat, jenž by poskytl server.

Koncept vrstveného systému zajišťuje rozdělení architektury systému do hierarchických vrstev. To umožňuje větší nezávislost, škálovatelnost a napomáhá vyvažování zátěže. Nevýhodou může být větší režie a odezva.

Vztah HTTP a REST

Nejčastěji se klient se serverem spojuje přes HTTP. Jejich vzájemná komunikace probíhá pomocí HTTP zpráv. Ty jsou buď request (požadavky), nebo response (odpovědi). Struktura zprávy je složena z tzv. start line (status line, pokud se jedná o response zprávu), kde se uvádí protokol a verze s request metodou, nebo response kódem. Dále headers (hlavičky), kde jsou

uvedeny doplňující informace o zprávě a nakonec body (tělo), kde se nachází data zprávy. Některé zprávy tělo mít nemusí, nebo ho dokonce vůbec nemají.

Request metody HTTP:

- GET – Je používána na vytvoření požadavku získání dat ze specifikovaného zdroje.
- HEAD – Velmi podobná GET metodě, akorát bez response body.
- POST – Používá se v případě zasílání dat serveru pro zpracování a vytvoření zdroje.
- PUT – Podobně jako POST zasílá data serveru pro zpracování s tím rozdílem, že místo vytvoření upraví stávající zdroj. Více stejných PUT požadavků bude mít stejný výsledek a provede se pokaždé na stejný zdroj, více stejných POST požadavků zapříčiní opakované vytvoření stejného zdroje.
- PATCH – Podobně jako PUT slouží pro modifikaci stávajícího zdroje s tím rozdílem, že v metodě PATCH jsou zaslány pouze provedené změny, v případě PUT je zaslána celá reprezentace zdroje.
- DELETE – Požadavek na smazání zdroje.
- OPTIONS – Slouží pro získání informací o komunikačních možnostech s cílovým zdrojem.

Na správné používání request metod přímo navazuje vytvoření přehledného URI. Při tomto procesu je dobré dodržovat nastavené praktiky a zásady. V první řadě nejdůležitější praktikou je udržovat jednotnost, ta zajistí mnohem jednodušší vývoj a použití, i kdyby URI nedodržovala žádnou z níže popsaných zásad, jednotnost alespoň zajistí přehlednost. Další praktika spočívá v nechávání typu operace pouze na typu HTTP request metody, v URI by pak explicitně nemělo být, o jakou metodu se jedná. Pro udržení jednotnosti rozhraní je dobré používat vždy množná čísla, tak že pokud je požadavek například získat všechny produkty (/products), pokud chceme pouze jeden, určí nám ho parametr (/products/1), tuto adresu by mohl mít například i PUT a DELETE. Parametrů se může v URI nacházet více. Pokud se v označení nachází mezera, nejčastěji se nahrazuje pomocí pomlčky. V adrese by se měly používat pouze malá písmena.

HTTP response je odpověď na HTTP request, vyznačuje se pomocí stavového kódu, což je tříciferné číslo, které přesně definuje výsledek požadavku. Stavové kódy se dají rozdělit do pěti skupin 1 až 5 podle jejich první cifry:

- 1 – Informační odpovědi, např. kód 100 (Continue) oznamuje, že byly obdrženy request hlavičky a klient by měl pokračovat v poslání těla zprávy.

- 2 – Úspěšné odpovědi, např. kód 200 (OK), informuje, že požadavek byl úspěšně zpracován. Tělo zprávy je ovlivněno na základě HTTP request metodě.
- 3 – Přesměrování, např. kód 301 (Moved Permanently), předává informaci o tom, že URI zdroje byl změněn a zasílá informace o novém.
- 4 – Klientské chyby, např. kód 403 (Forbiden), oznamuje o tom, že požadavek selhal kvůli nedostatečnému oprávnění.
- 5 – Serverové chyby, např. kód 500 (Internal Server Error), obecná zpráva, která říká, že na serveru došlo k neočekávané chybě.

[13]

SOAP

Simple Object Acess Protocol zkráceně SOAP, je síťový komunikační protokol na aplikační vrstvě, založený na XML. SOAP podobně jako REST může tvořit distribuované rozhraní, pomocí kterého mohou klienti zasílat svoje požadavky a server jim na ně odpovídat. Nejčastější způsob komunikace probíhá přes HTTP.

SOAP zpráva je XML dokument, který obsahuje elementy:

- Envelope,
- Header,
- Body,
- Fault.

Envelope (obálka) je povinný tag, který ohraničuje začátek a konec zprávy. Uvnitř jsou zapouzdřeny ostatní element. Volitelně může obsahovat elementy Header a Fault, naopak musí mít jeden element Body.

Header (hlavička) je volitelný element, obsahuje doplňující atributy používané ke zpracování zprávy. Uvnitř dokumentu se musí nacházet vždy na začátku pod tagem Envelope.

Body (tělo) je povinný element, který obsahuje data strukturovaná do aplikací definovaného XML. Ve struktuře se nachází pod elementem Header, pokud je definovaný.

Fault je volitelný tag zapouzdřený v těle, sloužící jako chybový mechanismus, který poskytne informace v případě chyby. Pokud vznikne chyba při zpracování požadavku, odpovědí na SOAP zprávu bude Fault element, který se vloží do těla zprávy.

Web Services Description Language zkráceně WSDL je dokument napsaný v XML používán pro popis SOAP webové služby.

- <definitions> – Kořenový element ohraničující začátek a konec dokumentu. Definuje také například název webové služby.
- <types> – Definují nové datové typy používané mezi klientem a serverem. Tyto definované datové typy mohou být použity u více webových služeb. Není třeba definovat, pokud jsou používány pouze předdefinované datové typy jako interger a string.
- <message> – Každá webová služba má tyto elementy dva. První popisuje vstupní parametry a druhý výstupní parametry. Datový typ je určen buď předdefinovaným datovým typem, nebo pomocí datového typu definovaného v elementu <types>.
- <portType> – Definuje poslopnost zpráv, která náleží dané službě.
- <binding> – Svázán s elementem <portType>, poskytuje dodatečné informace, jaký protokol se použije pro *portType* operace.
- <port> – Udává jedinečný koncový bod služby.
- <service> – Zapouždřuje několik elementů <port> dohromady.

[40], [42], [52]

Porovnání REST a SOAP

REST je spíše architektonický styl než protokol, to znamená, že umožňuje mnohem větší flexibilitu, co se týče struktury, formátu zpráv a toho jak se klient nebo server rozšiřuje. SOAP na druhou stranu vyžaduje mnohem těsnější vazbu mezi klientem a serverem a pokud by jedna strana něco změnila, může dojít k chybám. SOAP se poprvé objevil v roce 1998, REST byl představen až v roce 2000.

Jejich hlavní rozdíly jsou zobrazeny v tabulce níže a dále jsou pak popsány jejich výhody.

Tabulka 1: Rozdíly mezi REST a SOAP

	REST	SOAP
Druh	Architektonický styl	Komunikační protokol
Stav	Bezstatový	Bezstavový, stavový
Formát	JSON, XML, HTML, text	XML
Komunikační protokoly	HTTP/HTTPS	HTTP/HTTPS, FTP, TCP, SMPT, XMPP
Bezpečnost	WS-Security, ACID, HTTPS, SSL	HTTPS, SSL
Rychlost	Vysoká	Nízká
Komunita	Velká	Malá

Výhody REST:

- Umožňuje pracovat s různou řadou datových formátů jako jsou JSON, který je nejčastěji používaný, dále XML, HTML a další. SOAP na druhou stranu je omezený pouze na XML.
- Nabízí lepší podporu webových prohlížečů díky tomu, že pracuje s datovým formátem JSON.
- Poskytuje vyšší výkon, zejména díky tomu, že ukládá do cache data, která se nemění nebo nejsou dynamická.
- Považuje se za jednodušší na použití.
- Je obecně rychlejší a používá méně bandwith („šířka pásma“).
- Je jednodušší začlenit do již fungujících webových aplikací, bez potřeby redaktorování síťové infrastruktury.

Výhody SOAP:

- Nabízí podporu pro robustnější bezpečnost díky WS-Security.
- Nabízí zabudovanou retry logiku, která v případě neúspěšné komunikace zajistí několik dalších pokusů o spojení před tím, než oznámí klientovi neúspěch. Tyto chyby také umožňuje logovat. Oproti tomu klient, který používá REST toto neumožňuje, pokud nastane chyba a chce se pokusit zaslat zprávu znovu musí tak učinit sám.
- Podporuje transakce, které dodržují pravidla ACID.
- Umožňuje dvojfázové ověřování skrz distribuované transakční prostředky.

REST je v dnešní době oproti SOAP mnohem více využívaný zejména díky jednoduchosti použití, ať už pro vývojáře, tak i pro konzumenty služby, dále díky svojí rychlosti a podpoře formátu JSON. Trendem u systému používajících SOAP je buď doplnění o REST rozhraní, nebo případně pak jeho úplné nahrazení. SOAP zůstává a převažuje především u velkých systému, které zařizují např. bankovní transakce, platby a rezervace letů atd. Obě řešení mají své klady a zápory, důležité je si vybrat správné řešení, který dává největší smysl pro poskytovatele služeb na základě všech požadavků.

[30], [31], [55]

2.2.2 MVC

Model-view-controller zkráceně MVC je architektonický vzor, který se používá při vývoji aplikací. Hlavní ideou je oddělit výslednou aplikaci do tří vrstev, kde se každá vrstva stará pouze o svoji práci a pořádně. Tím se výsledná aplikace stává více nezávislá, přehledná a rozšiřitelná. Nejčastěji je MVC použit pouze jako základní myšlenka, konkrétní implementace se mohou lišit a rozvíjet na základě použitého jazyka či konkrétního projektu, příkladem může být přidání další vrstvy.

Model reprezentuje dynamickou datovou strukturu, která nese data aplikace. Každou třídu modelu je si možné představit jako POJO třídu (Plain old Java object), což je prostá veřejná třída, který musí mít defaultní konstruktor, může mít i konstruktor s parametry a obsahuje privátní atributy, se kterými se manipuluje přes get a set metody.

View je zodpovědný za uživatelské rozhraní a veškerou práci s ním, představuje vše, co uvidí uživatel, který bude aplikaci používat. V tomto vzoru je především zodpovědný za zobrazení dat z modelu a umožnění manipulace s nimi.

Controller funguje jako rozhraní mezi model a view komponentami. Zřizuje všechnu byznys logiku, zpracovává příchozí požadavky, manipuluje data pomocí modelu a podílí se na vytváření výsledného view.

[39]

Spring MVC

Spring MVC je Java framework, který se používá při vytváření webových aplikací. Už podle názvu výchází ze základního MVC vzoru, který obohacuje o funkce, které přidává Spring. Nabízí elegantní řešení za pomocí třídy DispatcherServlet, dále již jenom jako DS, která zpracovává příchozí požadavky a mapuje je. Oproti klasickému MVC je zde přidána nová vrstva tzv. front controller.

Komponenty view a model se vůči klasickému MVC nijak moc neliší, avšak controller se už nyní zařizuje zejména na aplikaci byznys logiky. Front controller je odpovědný za mapování požadavků a správu toku, který by se dal popsat v pěti krocích:

1. Na front controller přijde požadavek.
2. DS zmapuje požadavek a přepošle ho na controller.
3. Controller zpracuje požadavek a zasílá zpět ModelAndView objekt, který obsahuje data a název daného view.
4. DS zašle název na rozhraní ViewResolver, který podle něj najde požadovaný view a zašle ho zpátky.
5. Nakonec DS předá model objekt do view, který na základě něj vytvoří výsledek.

[35], [51]

2.2.3 DAO

Data Access Object zkráceně DAO. DAO designový vzor je používán pro oddělení data persistence logiky do separátní vrstvy, pomocí které je servisní vrstva oproštěna o logiku low-level operací používaných pro přístup do databáze. Je skládán z následujících komponent:

- model,
- DAO rozhraní,
- třídu implementující DAO rozhraní.

Výhodou vytvoření nové vrstvy je oddělení závislosti servisní vrstvy na operacích přístupu k databázi, či dalším datovým zdrojům. V případě změny databáze není třeba provádět žádné změny v servisní vrstvě, stačí pouze změnit DAO implementaci. Odděluje také svoji závislost vůči view vrstvě. Další výhodou může být zjednodušení testování například při psaní unit testů, umožňuje jednoduše vytvořit DAO jako mock objekt.

[8], [9], [38]

3 ANALÝZA A NÁVRH ŘEŠENÍ

Tato kapitola se zabývá analýzou a návrhem systému. Nejprve je provedena analýza podobných řešení a dále je popis funkčních a nefunkčních požadavků, storyboardu, rich picture, activity diagramu, diagramu tříd a databázového modelu. Nakonec je popsán návrh aplikace z pohledu implementace rolí, oprávnění a funkcionality.

3.1 Podobná řešení

Tato podkapitola se zabývá analýzou podobných systémů, jako je vytvářená aplikace. Zaměřuje se zejména na poskytované funkce. Dále zde budou analyzovaná řešení porovnána s navrhovaným systémem.

3.1.1 CodingBat

CodingBat je webová aplikace, která nabízí testování zdrojového kódu na vytvořených úlohách v jazycích Java a Python. Úlohy jsou děleny do kategorií podle řešené problematiky a obtížnosti např. String-1, String-2, Array-1, Array-2, Map-1. V každé kategorii je připraveno několik úloh. Každá úloha se skládá z implementace jedné metody. Přesné chování metody je uvedeno v zadaní, ve kterém je dále naznačeno, jaký výstup je očekáván při daném vstupu.

Textový editor se chová podobně jako běžné vývojové prostředí, které zabarvuje klíčová slova, pouze neobsahuje nápovědu. Některé úlohy nabízí možnost nějaké formy nápovědy:

- zobrazení celého řešení,
- zobrazení hlavního klíče k řešení,
- zobrazení popisu API.

Po odeslání kódu se buď ukáže chyba, nebo tabulka s výsledkem řešení. Tabulka řešení zobrazuje očekávaný výstup na základě vstupu, s aktuální hodnotou výstupu a zda daný test prošel, nebo ne. Zobrazeno je většinou několik testových případů s kolonkou other test („další testy“), ve které jsou další testové scénáře, které zabraňují tomu, aby uživatel vyřešil úlohu tak, že by vytvořil podmínky na základě vstupů, do kterých by dal výstupy zobrazené v tabulce. Toto zajistí, aby úloha byla vyřešena na 100 % pomocí správného algoritmu. Uživatel si také může zobrazit graf, který mu ukazuje progres na dané úloze. Přihlášenému uživateli se ukládá historie řešených úloh, kde si může zobrazit tyto grafy na úlohách, které řešil v minulosti.

Aplikace dále nabízí nápovědu, jak řešit dané úlohy s několika ukázkovými příklady a obecnou nápovědu pro kategorie daného jazyka.

[7]

3.1.2 Codewars

Codewars je webová aplikace, která nabízí testování zdrojového kódu na vytvořených úlohách v mnoha jazycích např. C#, C++, C, Python, PHP, Kotlin a další. Nepřihlášenému uživateli je umožněn přístup pouze k jedné jednoduché úloze z každého jazyka. Po registraci a přihlášení je odemknut veškerý obsah.

Přihlášený uživatel si může vybrat specifikovat svoje oblíbené jazyky a systém mu na základě toho předává automaticky úlohy, které označí jako „next challenge“. Úlohy je také možno vyhledávat a plnit samostatně, každá úloha obsahuje seznam jazyků, pro které je navržena, dále je pak označena značkou obtížnosti a tagy, které specifikují typ úlohy.

Úlohy se podobně jako u CodingBat skládají z implementace především jedné metody, ale je možnost implementovat i celou třídu. Je zde předloženo zadání s ukázkovým řešením vstupů a výstupů. K řešení je přidána ukázková testovací třída, ve které je několik testovacích scénářů, tuto třído je možno libovolně upravovat a použít nanečisto na implementovanou metodu. Po opravdovém pokusu o odevzdání se implementace otestuje vůči předpřipraveným testům a vrátí se výsledek, který se v grafické obdobě podobá výsledku testů ve vývojovém prostředí. Ukazuje prošlé testy a u neprošlých zobrazuje aktuální a očekávanou hodnotu. Uživateli jsou pak v profilu zobrazeny nejrůznější statistiky o jeho aktivitách na stránce a výsledcích úloh. V záložce řešení má možnost si zobrazit splněné úlohy s implementací a případně je refaktorovat, nebo nesplněné úlohy, kde je uložena poslední odevzdaná implementace s možností dokončení úlohy.

[6]

3.1.3 Porovnání řešení

V níže zobrazené tabulce jsou zhodnoceny a porovnány některé vlastnosti/kritéria daných řešení a navrhovaného řešení.

Tabulka 2: Porovnání řešení

	CodingBat	Codingwar	Navrhovaný systém
Přihlášení/registrace	Ano	Ano	Ano
Testování implementace metody	Ano	Ano	Ano
Testování implementace třídy	Ne	Ano	Ano
Uložení řešení	Ne	Pouze poslední řešení	Všechna odevzdaná řešení
Graf progresu úlohy	Ano i anonymní graf	Ne	Ano i anonymní graf
Statistika celkového progresu	Ne	Ano	Ne
Přístup k úlohám	Bez omezení	Pouze po přihlášení	Na základě odemčení
Testování nanečisto	Ne	Ano	Ne
Zobrazení přesných výsledků testovacích scénářů	Pouze vybraných	Ano	Ano i ne
Možnost splnění úlohy na základě informací z výsledků testů	Ne	Ne	Ano i ne

[6], [7]

3.2 Funkční a nefunkční požadavky

V této podkapitole jsou popsány hlavní funkční a nefunkční požadavky na systém na základě analýzy zadaní a konzultací se zadavatelem.

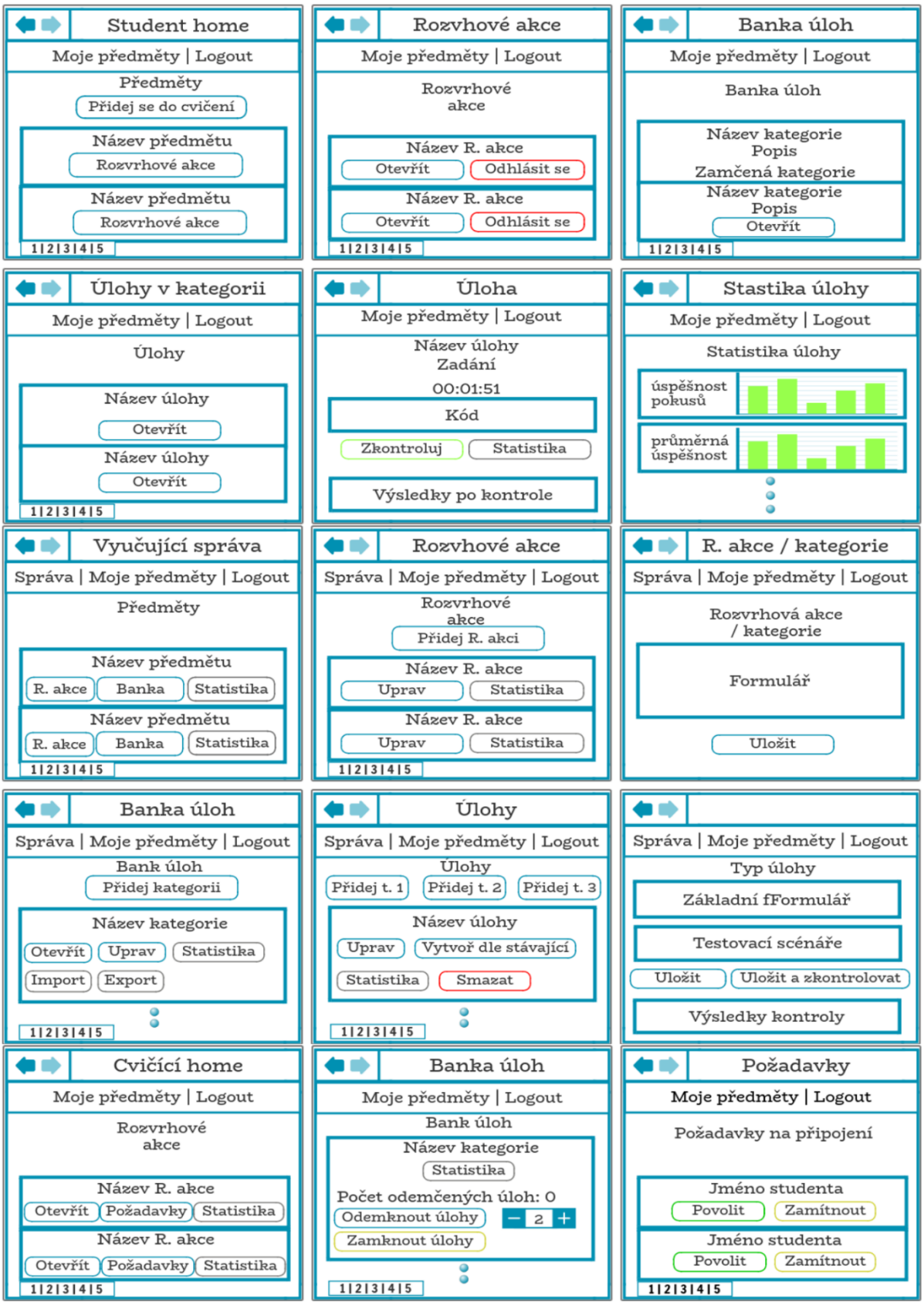
Funkční požadavky

- Systém musí umožňovat automatickou kontrolu zdrojových kódů pro jazyk Java SE.
- Systém musí umožňovat vytváření předmětů.
- Systém musí umožňovat vytvoření cvičení pro daný předmět.
- Systém musí umožňovat vytvoření kategorií pro daný předmět.
- Systém musí umožňovat vytváření úloh pro danou kategorii.
- Systém musí umožňovat vytváření testových scénářů pro danou úlohu.
- Systém musí umožňovat import a export úloh v kategorii.
- Systém musí umožňovat práci s primitivními datovými typy.
- Systém musí umožňovat práci s referenčními datovými typy.
- Systém musí umožňovat testování metody, třídy a třídy s rozhraním.
- Systém musí umožňovat zapsání studenta k cvičení.
- Systém musí umožňovat odemykání úloh studentům ve vybrané kategorii.
- Systém musí při odemykání úloh náhodně zvolit úlohy pro každého studenta zvlášť.
- Systém musí umožňovat správu uživatelů.
- Systém musí umožňovat import studentů na základě dokumentů ze systému stag.

Nefunkční požadavky

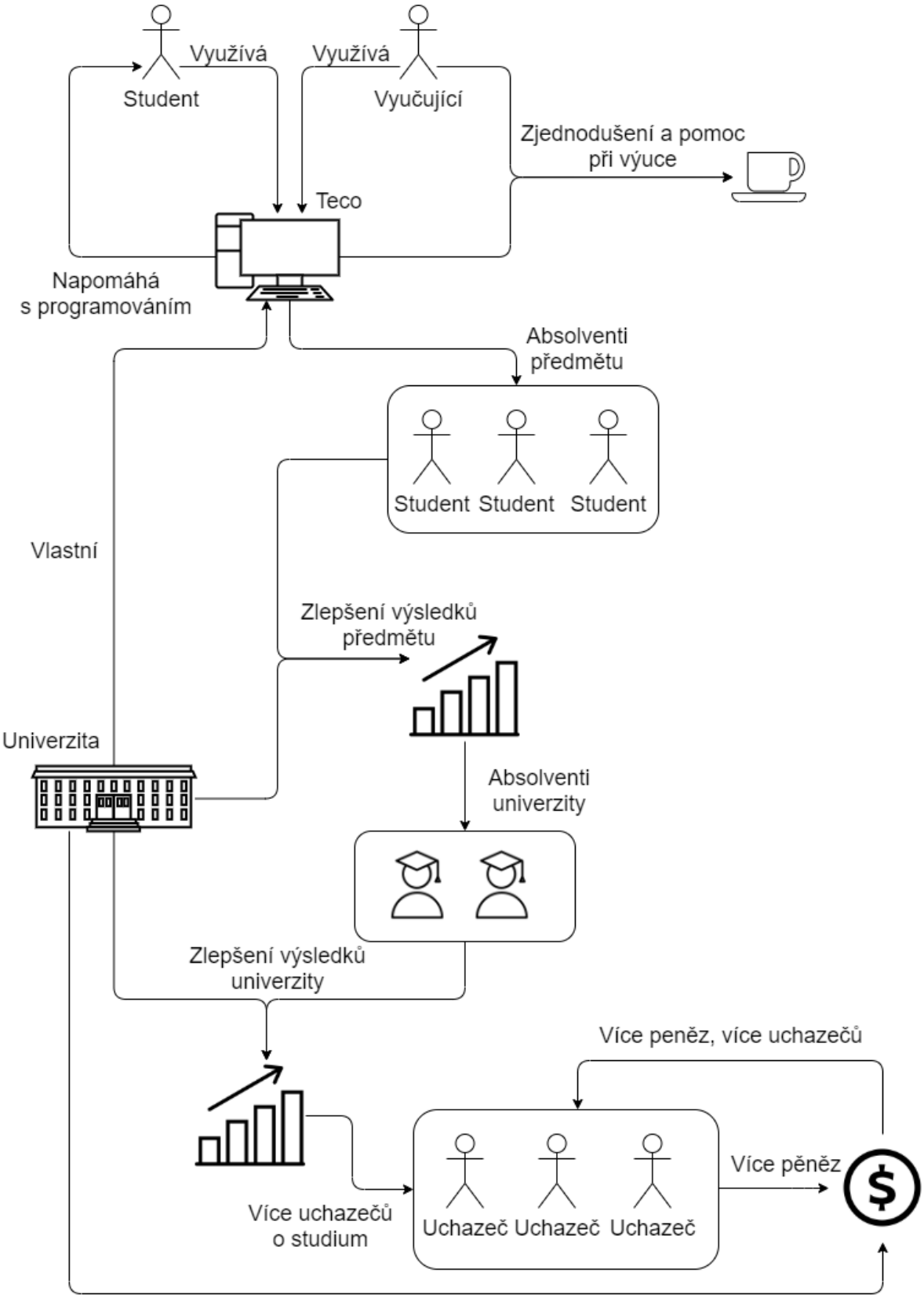
- Systém musí poskytovat výstižný přehled statistik o odevzdaných úlohách.
- Systém musí poskytovat možnost rozšíření do dalších předmětů.
- Systém musí zamezit zpracování kódu nebezpečného kódu.
- Systém musí zamezit zpracování zacykleného kódu.

3.3 Storyboard



Obrázek 1: Storyboard

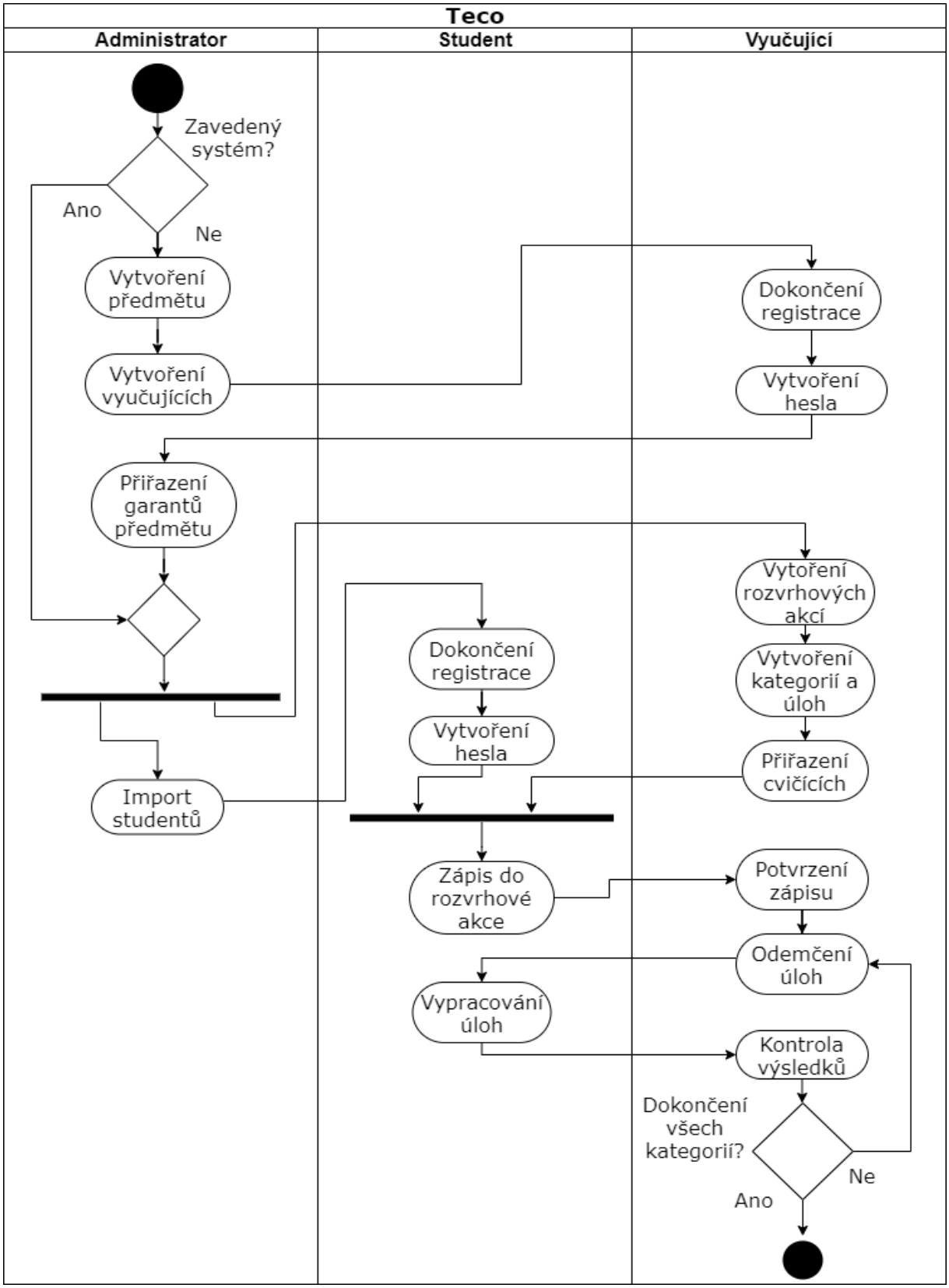
3.4 Rich picture



Obrázek 2: Rich picture

3.5 Activity diagram

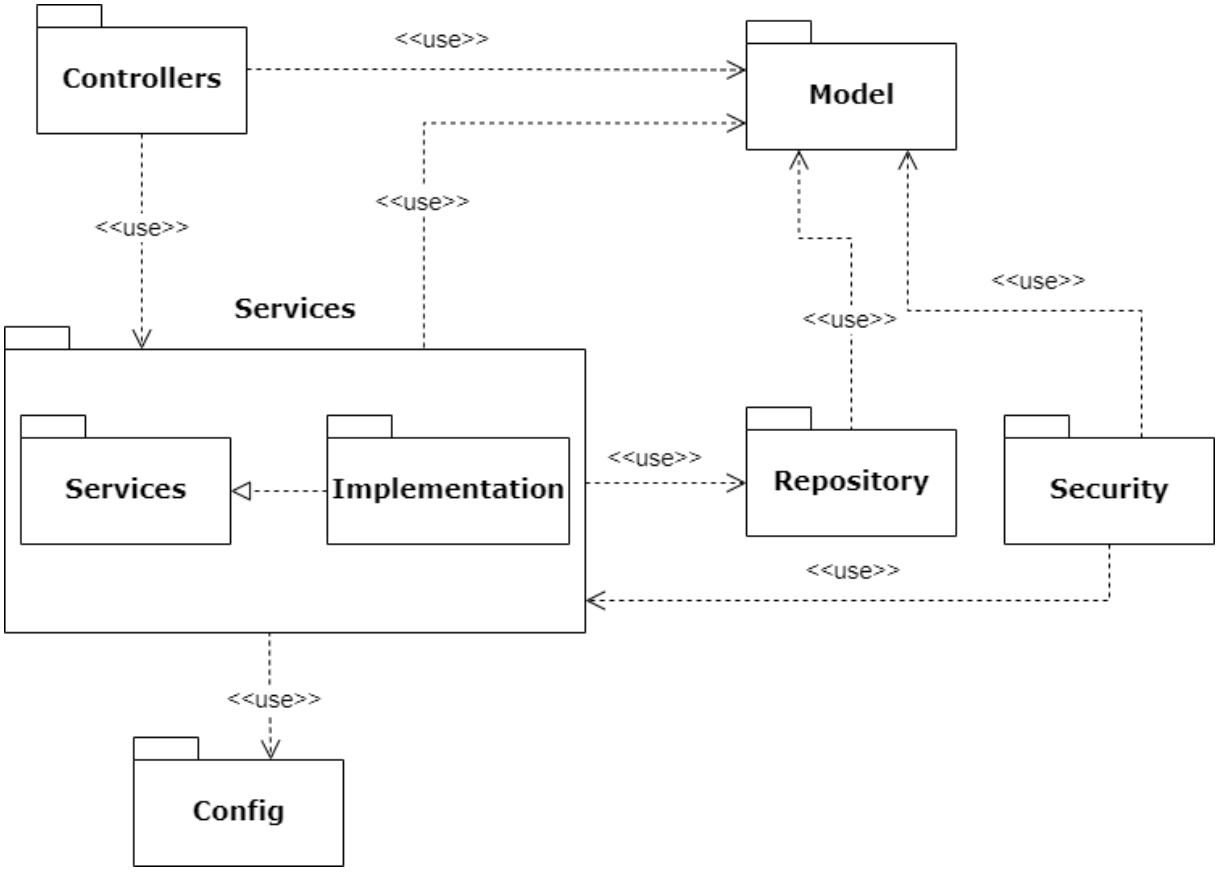
Activity diagram zobrazuje obecný chod systému.



Obrázek 3: Activity diagram

3.6 Package diagram

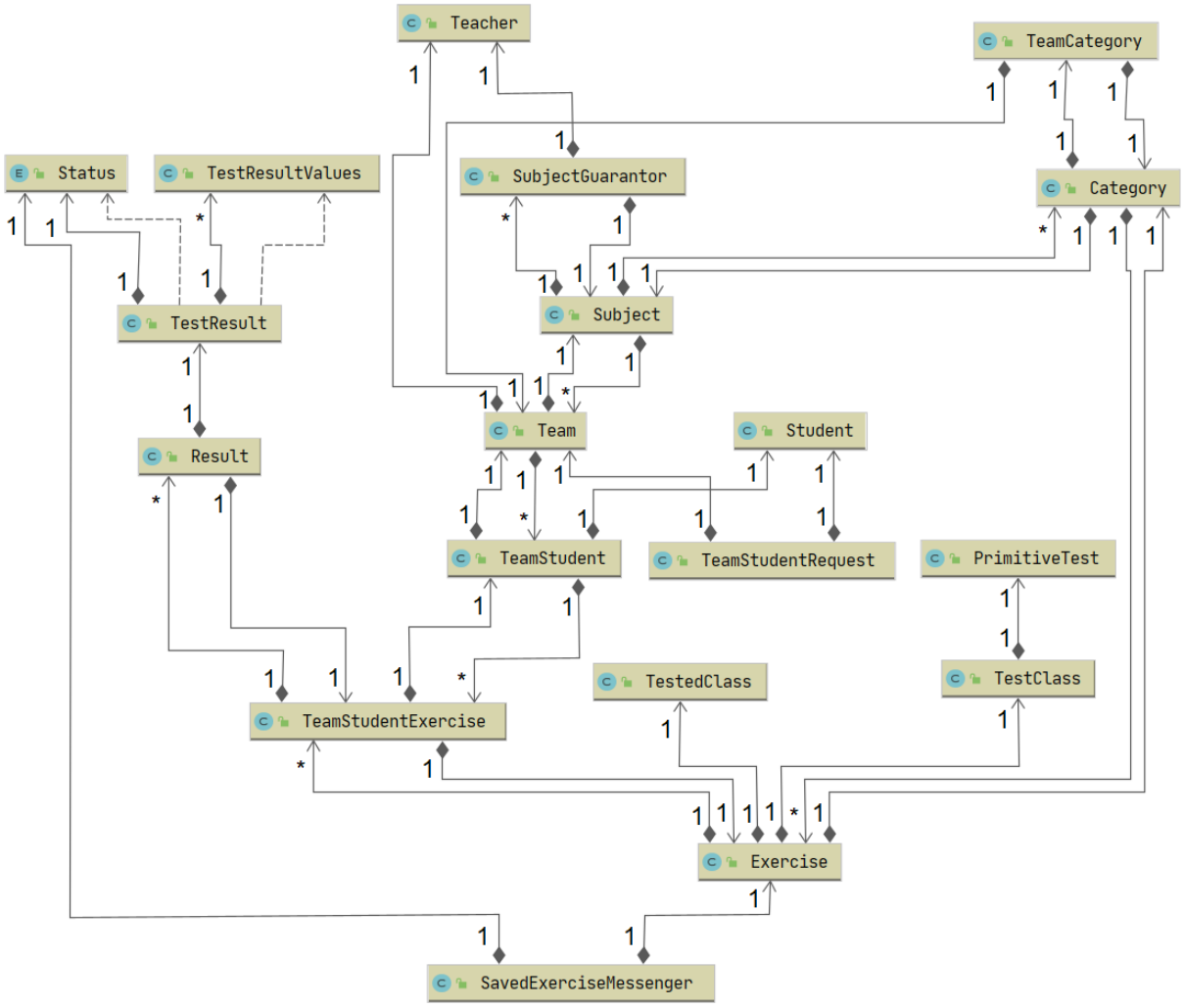
Package diagram zobrazuje vztahy mezi jednotlivými balíčky v backendové aplikaci. Podrobnější implementace tříd v balíčcích je rozebrána v kapitole 4 Implementace webové aplikace. Digram balíčku model zobrazuje také kapitola 3.7 3.7Třídní diagram, kde jsou zobrazeny vztahy mezi třídami a kapitola 3.8 Databázový model, kde model databáze vychází z implementace tříd v balíčku model.



Obrázek 4: Package diagram

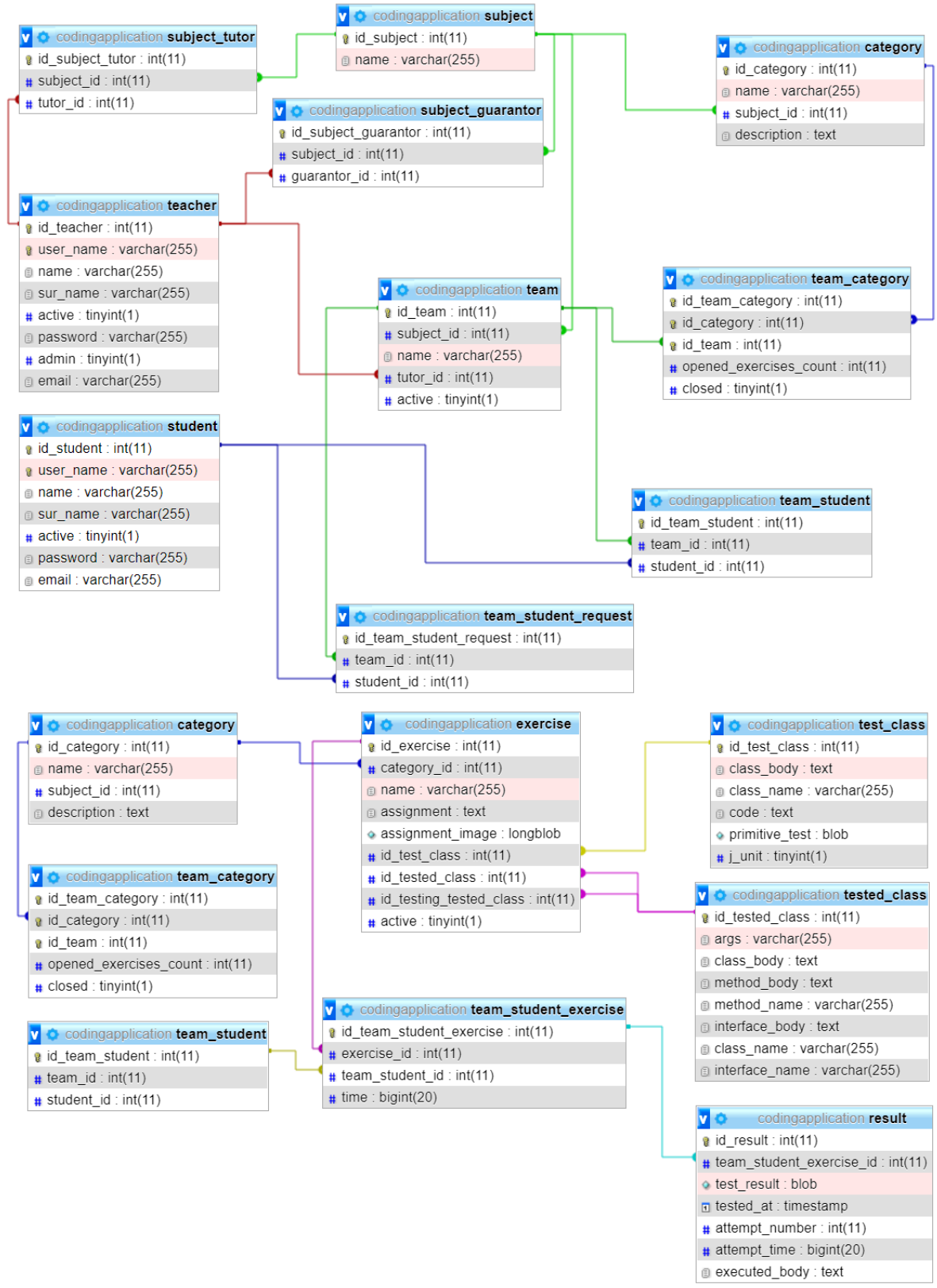
3.7 Třídní diagram

Třídní diagram zobrazuje vztahy mezi třídami v modelu.



Obrázek 5: Třídní diagram

3.8 Databázový model



Obrázek 6: Databázový model

3.9 Návrh aplikace

Tato kapitola je zaměřena na navržení aplikace z pohledu vytvoření rolí a oprávnění. Dále jsou zde popsány základní funckionality s potřebnými oprávněními.

3.9.1 Role a oprávnění

V systému jsou zavedeny role:

- administrátor,
- vyučující,
- student.

Administrátor má přiřazené administrátorské oprávnění. Vyučující může mít omezené administrátorské oprávnění v případě, že mu ho administrátor přidělí, to slouží pro jednodušší správu systému. Oprávnění garanta a cvičícího jsou vyučujícímu přidělovány ke konkrétnímu předmětu. Student má automaticky oprávnění studenta.

3.9.2 Funkcionalita

V této kapitole jsou popsány základní funkcionality, které může uživatel provést s danými oprávněními.

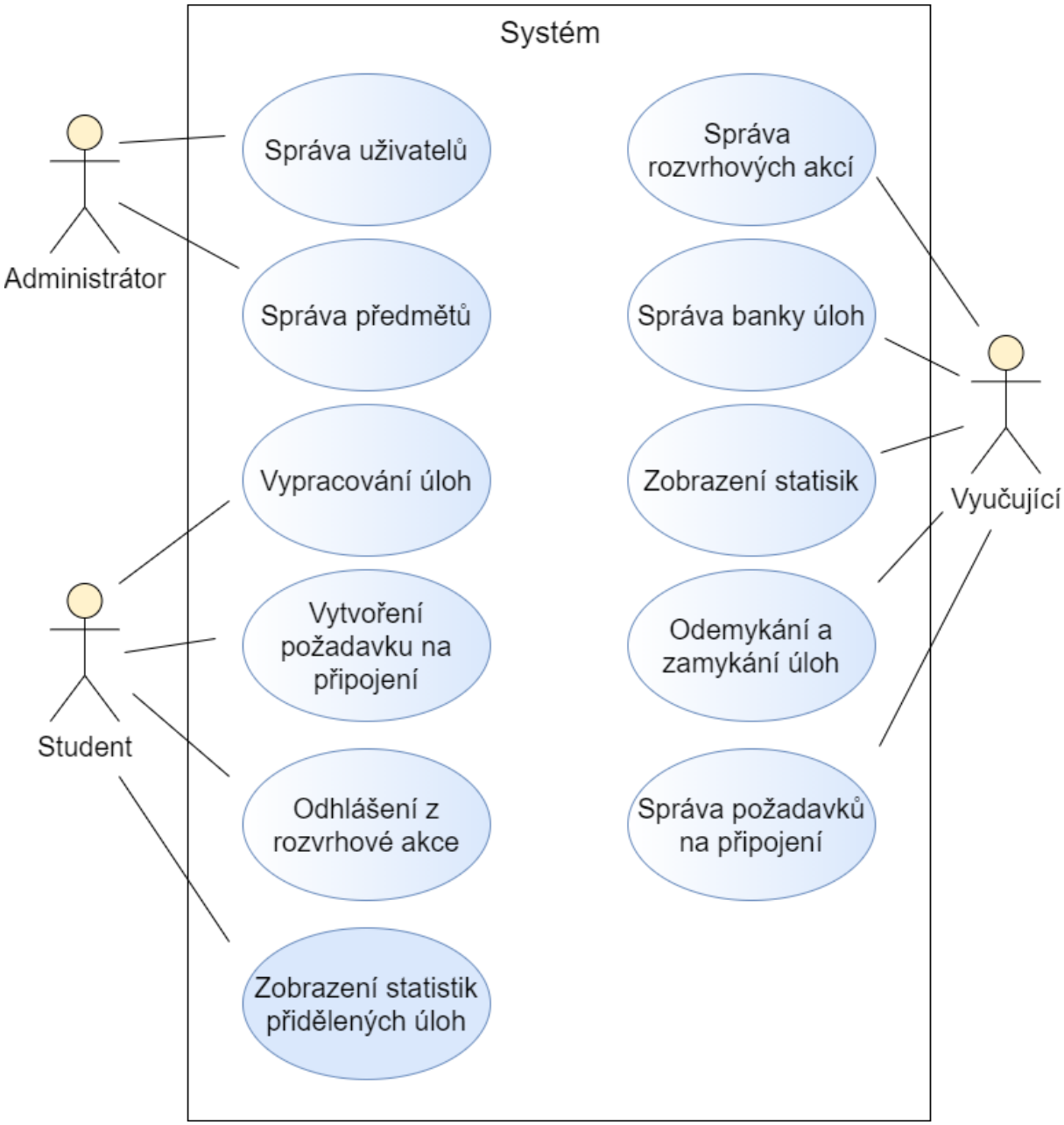
Administrátorské oprávnění

Administrátorské oprávnění umožňuje:

- Přidat administrátora – Vytvoření nového administrátorského účtu.
- Spravovat svůj účet – Změna základních údajů.
- Spravovat vyučující – Přidávání a úprava údajů vyučujícího. Možnost zaslání emailu pro nové heslo a nastavit vyučujícímu omezené administrátorské oprávnění.
- Spravovat studenty – Přidávat jednotlivě, nebo importovat ze souboru. Úprava údajů studenta. Možnost zaslání emailu pro nové heslo.
- Spravovat předměty – Přidávání a mazání předmětu. Úprava názvu předmětu a přiřazení garantů předmětu.

3.9.3 Use case diagram

Zjednodušený use case diagram představuje interakci hlavních rolí se systémem.



Obrázek 7: Use case diagram

Omezené administrátorské oprávnění vyučujícího

Stejné oprávnění jako výše uvedené administrátorské oprávnění kromě možnosti přidat nového administrátora a přidělení vyučujícímu omezené administrátorské oprávnění.

Oprávnění garanta

Oprávnění garanta umožňuje vyučujícímu správu svých předmětů, což zahrnuje:

- Správa rozvrhových akcí – Přidávání rozvrhových akcí. Úprava názvu rozvrhové akce a přiřazení cvičícího. Možnost zobrazení statistik pro rozvrhovou akci.
- Správa banky úloh – Vytváření a úprava kategorií v rámci předmětu. Import a export úloh kategorie. Vytváření, úprava a mazaní úloh. Možnost zobrazení statistik kategorií a úloh.
- Možnost zobrazení výsledků a statistik předmětu a jeho studentů.

Oprávnění cvičícího

Oprávnění cvičícího umožňuje vyučujícímu správu svých rozvrhových akci, což zahrnuje:

- Odemykání a zamykání úloh v rámci kategorií pro studenty.
- Správa požadavků na připojení do rozvrhové akce.
- Možnost zobrazení výsledků statistik rozvrhové akce a jednotlivých kategorií.

Oprávnění studenta

Oprávnění cvičícího umožňuje studentovi:

- Vypracování odemčených úloh.
- Zobrazení statistik na přidělené úlohy a zobrazení svých odevzdaných pokusů.
- Vytvoření požadavku na zápis do rozvrhové akce předmětu.
- Odhlášení se z rozvrhové akce.

4 IMPLEMENTACE WEBOVÉ APLIKACE

Tato kapitola je zaměřena na implementaci webové aplikace. Nejprve jsou zmíněny použité technologie a software. Dále je popsána architektura a struktura aplikace, kde jsou poté detailněji rozebrány vrstvy backendu a frontendu.

4.1 Použité technologie a software pro vypracování

V této podkapitole jsou vypsány použité technologie a software, pomocí nichž byla vypracována výsledná aplikace.

4.1.1 Technologie

V této podkapitole jsou vypsány použité technologie pro vytvoření webové aplikace.

- Java – Jeden z nejrozšířenějších programovacích jazyků. Pro vývoj backendu byla použita nadstavba Spring Boot, která nabízí zjednodušenou platformu pro tvorbu aplikací.
- JavaScript – Další velmi rozšířený a populární programovací jazyk. Využita byla jeho knihovna React, která je vhodná pro vytváření frontendu webových aplikací.
- CSS – Jazyk použitý pro nastavení vzhledu aplikace. Použité připravené knihovny jako Bootstrap.
- MySQL – Relační databázový systém pro ukládání dat. Vhodné řešení díky své velké rozšířenosti a dokumentaci. Dostačující rozsahu aplikace.

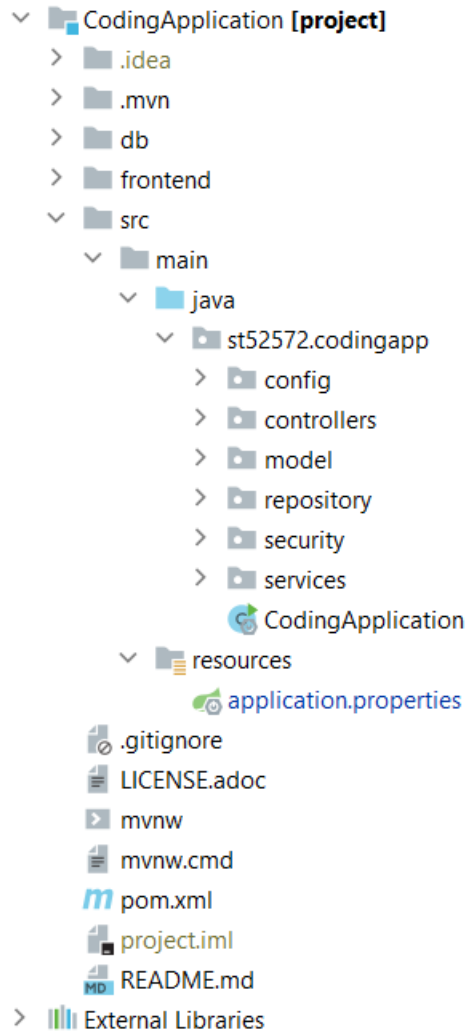
4.1.2 Software

V této podkapitole je vypsán použitý software při vytváření systému.

- Enterprise Architect – Nástroj pro analýzu, návrh a modelování systémů.
- IntelliJ IDEA – Vývojové prostředí.
- phpMyAdmin – Nástroj pro správu MySQL databáze.
- XAMPP – Nástroj vytvoření lokálního webového serveru a databáze.

4.2 Architektura a struktura

Architektura aplikace je složena z několika návrhových vzorů. Hlavní rozdělení aplikace přichází separací na frontend a backend. Frontend je zodpovědný za uživatelskou aplikaci, vytváření požadavků, které zasílá na backend a jejich přijetí a zpracování/zobrazení. Backend tyto požadavky zpracovává a zasílá zpět výsledky. Na následujícím obrázku je vyobrazena struktura aplikace ve vývojovém prostředí.



Obrázek 8: Struktura aplikace

Pro vytvoření finální architektury byla použita základní myšlenka MVC, kde je view (frontend) oddělen od zbytku aplikace, model reprezentuje datovou strukturu a controller zpracovává požadavky. V tomto případě se controller rozkládá ještě na další části:

- Controller – V tomto případě jednotlivé kontrolery vytváří rozhraní, které používá frontend pro komunikaci s backendem.

- Service – Tvoří rozhraní a třídy pro implementaci byznys logiky. Výhodou je další logické rozdělení kódu a jeho znovupoužitelnost.
- Repository – Tvoří rozhraní, které odděluje data persistence logiku. Zajišťuje CRUD operace s databází. Zde bylo využitu návrhového vzoru DAO.

4.3 Backend

Tato podkapitola detailně rozebírá backend, což je serverová část aplikace, která zpracovává, odpovídá na požadavky, komunikuje s databází a zajišťuje autorizovaný přístup.

4.3.1 REST API

REST rozhraní tvoří všechny controller třídy aplikace. Zajišťují přístupové body, pomocí kterých uživatelé mohou provádět různé akce. Kontrolery jsou roztříděny do různých tříd podle použití. Jsou označeny anotacemi @RestController a @RequestMapping. @RestController označuje třídu jako controller a automaticky serializuje návratový objekt do HttpResponse. @RequestMapping(„/mapování“) zajišťuje jednotné mapování pro všechny metody v kontroleru. Aplikace obsahuje:

- *AdminController* – Zajištuje rozhraní pro práci s entitou *Admin*. Využívá službu *AdminService.*
- *CategoryController* – Zprostředkovává rozhraní pro práci s entitou *Category*, k tomu využívá službu *CategoryService.*
- *CodeTesterController* – Zajišťuje rozhraní pro ověřování kódů uživateli. Využívá službu *CodeTestService.*
- *ExerciseController* – Zprostředkovává rozhraní pro práci s entitou *Exercise*. Využívá službu *ExerciseService.*
- *ResultController* – Zajišťuje rozhraní pro práci s entitou *Result*, k tomu využívá službu *ResultService.*
- *StudentController* – Zprostředkovává rozhraní pro práci s entitou *Student.* Využívá službu *StudentService.*
- *SubjectController* – Zajišťuje rozhraní pro práci s entitou *Subject.* Využívá službu *SubjectService.*

- *TeacherController* – Zprostředkovává rozhraní pro práci s entitou *Teacher.* Využívá službu *TeacherService.*
- *TeamController* – Zajišťuje rozhraní pro práci s entitou *Team.* Využívá službu *TeamService.*
- *TeamStudentExerciseController* – Rozhraní pro práci s entitou *TeamStudentExercise.* Využívá službu *TeamStudentExerciseService.*
- *UserController* – Zajišťuje rozhraní pro práci se všemi uživateli např.: login. Využívá službu *LoginService.*

4.3.2 Servisní vrstva

Servisní vrstva zajišťuje implementaci byznys logiky aplikace. Je tvořena z rozhraní a tříd, které je implementují. Třídy obsahují anotaci @Service(„název“), která označí třídu jako service a pomocí názvu umožní její autowiring. Rozhraní jsou pojmenovány jako „Název+Service“ a implementující třídy „Název+ServiceImpl“. Aplikace obsahuje:

- *AdminService* – Zajišťuje práci s entitou *Admin*.
- *CategoryService* – Zprostředkovává práci s entitou *Category* jako uložení kategorie, získání jedné, nebo seznamu kategorií na základě různých parametrů (kategorie předmětu, kategorie teamu).
- *CodeTestService* – Zajišťuje zpracování, kompilaci a testování kódů odevzdaných uživateli, více popsáno v kapitole 5 Testování zdrojových kódů.
- *ExerciseService* – Obstarává práci s entitou *Exercise* jako generování úloh, deaktivaci úlohy, smazání úlohy, uložení úlohy, export a import úloh v JSON formátu, získání jedné, nebo seznamu úloh na základě parametrů (studentovi úlohy v kategorii, úlohy předmětu v dané kategorii).
- *LoginService* – Zajišťuje přihlášení, což zahrnuje autentizaci pomocí rozhraní *AuthenticationManeger* na základě jména a hesla. Po úspěšné autentizaci je vytvořen JTW (JSON Web Token), který je zaslán zpět uživateli, který bude sloužit pro autentizace dalších požadavků.
- *NewPasswordTokenService* – Zprostředkovává službu vytvoření nového hesla pomocí zaslání unikátního odkazu do emailu, pomocí něhož si uživatel může vytvořit nové heslo.

- *ResultService* – Zajišťuje práci s entitou *Result*. Ukládá výsledky a vrací seznamy výsledků na základě různých parametrů.
- *StudentService* – Obstarává práci s entitou *Student*. Ukládá studenta, importuje studenty, získává jednoho, nebo seznam studentů na základě parametrů.
- *SubjectService* – Zajišťuje práci s entitou *Subject*. Ukládá předmět, maže předmět, získává jeden, nebo seznam předmětů na základě parametrů (podle studenta, cvičícího, garanta)
- *TeacherService* – Zprostředkovává práci s entitou *Teacher*. Ukládá vyučujícího, získává jednoho, nebo seznam vyučujících na základě parametrů.
- *TeamService* – Zajišťuje práci s entitou *Team*. Ukládá cvičení, spravuje požadavky na připojení ke cvičení, získává jedno, nebo seznam cvičení na základě parametrů (cvičení předmětu, cvičení vyučujícího, cvičení studenta)
- *TeamStudentExerciseService* – Zajišťuje práci s entitou *TeamStudentExercise.*
- *UserDetailService* – Rozhraní ze Spring Framework security, které obsahuje metodu *loadUserByUsername*. Metoda načte uživatele z databáze a vrací objekt třídy, která implementuje rozhraní *UserDetails*, které obsahuje metody:
    - *getAuthorities* – Vrací kolekci oprávnění, v tomto případě se jedná o kolekcí rolí uživatele.
    - *getPassword* – Vrací heslo.
    - *getUsername* – Vrací uživatelské jméno.
    - *isAccountNonExpired* – Vrací, zda fungování účtu nevypršelo.
    - *isAccountNonLocked* – Vrací, zda účet není zablokován.
    - *isCredentialsNonExpired* – Vrací, zda ověření účtu nevypršelo.
    - *isEnabled* – Vrací, zda účet je povolen.

4.3.3 Vrstva modelu

Vrstva modelu představuje datovou vrstvu aplikace. To zahrnuje vytvoření entitních a dalších potřebných datových tříd. Každá entitní třída také představuje tabulku v databázi, čím velmi usnadňuje vývoj, jelikož pomocí frameworku Hiberante, který je implementací Java Persistence API, definuje, jak se budou uchovávat data v aplikaci. Hibernate je jeden z nejpopulárnějších

ORM frameworků, který převádí Java objekty na databázové tabulky a umožňuje práci s databází bez použití SQL.

Definování entit se provádí pomocí anotace @Entity. Další důležitou anotací je @Table, která umožňuje nastavení tabulky v databázi, například nastavením názvu tabulky nebo schématu. Každá entitní třída také musí implementovat rozhraní *Serializable*.

Sloupce tabulky jsou definovány pomocí atributů třídy. Každá entita by měla obsahovat identifikátor (primární klíč v databázi), který se specifikuje pomocí anotace @Id. Automatické generování identifikátoru je nastaveno anotací @GeneratedValue. Sloupce tabulky jsou možné definovat nepovinnou anotací @Column, ve které je možné nastavit jméno, délku, definici sloupce a další constraints (omezení) jako nullable, unique, updatable, insertable. Výchozí jméno sloupce je nastaveno podle názvu atributu, kde se místo standartního lower camel case v jazyce Java používá snake case, kde jsou slova oddělena podtržítkem.

Mapování relací/vztahů mezi tabulkami/entitami se vytváří pomocí anotací:

- @OneToOne – Vztah 1:1.
- @OneToMany – Vztah 1:N.
- @ManyToOne – Vztah N:1.
- @ManyToMany – Vztah M:N.

Další specifikací relace se provádí pomocí @JoinColumn a @JoinTable. Mapování může pak být buď unidirectional, nebo bidirectional. Bidirectional oproti unidirectional mapování umožňuje přístup k entitám v obou směrech.

Implementace vztahu 1:1 s cizím klíčem a s bidirectional mapováním se provede označením atributu v třídě *A* anotací @OneToOne, kde specifikuje parametr *mappedBy*, kterému se nastaví jako hodnota název atributu v třídě *B*. V třídě *B* se atribut anotuje pomocí @OneToOne a @JoinColumn, ten bude sloužit v databázi jako cizí klíč.

Implementace vztahu 1:N s bidirectional mapováním se provede obdobně jako u vztahu 1:1. Rozdílem pro třídu *A* je použití anotace atributu @OneToMany. Dále také datový typ tohoto atributu musí být nějaká kolekce (*Set*, *List*). U třídy *B* je změna pouze v anotaci atributu na @ManyToOne.

Implementace vztahu M:N s bidirectional mapováním může být buď dosaženo použitím dvou 1:N vazeb, nebo pomocí anotací @ManyToMany. V třídě A je pak kolekce anotovaná pomocí @ManyToMany s nastavením parametru *mappedBy*. V třídě B je atributem také kolekce, která

má anotace @ManyToMany a @JoinTable, který specifikuje vytvoření propojovací tabulky v databázi.

Ve vytvářené aplikaci je použito je u některých vztahů použito mapování 1:1. Zbylé mapování je provedeno pomocí vztahu 1:N i pro M:N vazby. Implementace tříd byla zjednodušena pomocí knihovny Lombok, která nabízí anotace např.:

- @Getter – Automaticky vytváří get metody.
- @Setter – Automaticky vytváří set metody.
- @Data – Zahrnuje anotace @Getter a @Setter, dále automaticky vytváří metody *toString*, *equals* a *hashCode* a také konstruktor, který jako parametry obsahuje všechny final (konstantní) a nenulové atributy třídy.
- @NoArgsConstructor – Automaticky vytváří defaultní konstruktor bez parametrů.
- @AllArgsConstructor – Automaticky vytváří konstruktor, který zahrnuje parametry pro všechny atributy třídy.

[3], [25]

4.3.4 DAO

DAO vrstva neboli vrstva repozitářů je tvořena rozhraními, které jsou označeny anotací @Repository. Tato rozhraní dědí od rozhraní JpaRepository<T, ID>. To obsahuje dva generické datové typy, první je třída, kterou repozitář spravuje (třída modelu) a druhý je datový typ identifikátoru dané entity.

Dotazy

Základní CRUD metody/dotazy jsou v rozraní zděděny po připojení JpaRepository. Velkou výhodou je, že tyto metody již není třeba implementovat v žádné další třídě, ale jsou rovnou připraveny k použití, stačí pouze autowiring daného rozhraní. Mezi takové metody patří například:

- find – Metody začínající slovem find slouží pro získání entit.
    - *findById* – Vrací objekt typu *Optional*<T>, která slouží jako kontejner, který může, nebo nemusí nabývat hodnoty. Bude obsahovat entitu v případě, že najde shodu na základě předaného identifikátoru.

    - *findOne* – Obdobně jako metoda *findById*, pouze je shoda hledána pomocí předané entity.
    - *findAll* – Vrací buď všechny, nebo pouze entity nalezené pomocí entit předaných argumentem.
- get – Obsahuje pouze metodu *getOne*, která získá entitu na základě identifikátoru. Na rozdíl od *findById* a *findOne* metod je její návratový typ přímo třída entity a v případě, že entita nebyla nalezena vyhodí výjimku.
- save – Metody začínající slovem save slouží pro ukládání entit.
    - *save* – Uloží jednu entitu.
    - *saveAll* – Uloží všechny entity předané jako argument.
- delete – Metody začínající slovem delete slouží pro mazání entit.
    - *deleteById* – Smaže entitu na základě předaného identifikátoru.
    - *delete* – Smaže entitu na základě předané entity.
    - *deleteAll* – Smaže buď všechny, nebo pouze entity předané jako argument.
- count – Metoda *count* vrací počet dostupných entit.
- exists – Metody začínající slovem *exists* vrací, zda daná entita existuje.
    - *existsById* – Vrací *true*/*false* na základě identifikátoru.
    - *exists* – Vrací *true*/*false* na základě předané entity.

Vlastní dotazy jsou vytvářeny pomocí přidání nových metod. Existují dva způsoby vytváření dotazů:

- Dotaz vytvořený z názvů metod.
- Dotaz vytvořený pomocí výrazu.

U výše zmíněných zděděných metod je možné si povšimnout vzoru podle, kterého jsou metody tvořeny nejdříve je určena operace, která se dále specifikuje (By, All, …). Takto podobně se vytvářejí i nové dotazy. S nápovědou vývojového prostředí je vytváření dotazů tímto způsobem velmi intuitivní.

```
@Repository
public interface StudentRepository extends JpaRepository<Student, Integer> {
    Optional<Student> findByUserName(String userName);
}
```

Ukázka zdrojového kódu 3: StudentRepository

Dotazy tvořeny pomocí výrazů se buď vytváří podle JPQL (Java Persistence Query Language), nebo SQL. Název metody v tomto případě nehraje žádnou roli na funkčnost. Výraz se metodě předá pomocí anotace @Query(*value* = „dotaz“). Ve výchozím stavu je používán JPQL, pro použití SQL stačí nastavit *nativeQuery* na hodnotu *true*.

[34], [56]

4.3.5 Konfigurace

V této podkapitole jsou popsány konfigurace provedené při vytváření aplikace.

Config

Třída *MailConfig* obsahuje bean typu *JavaMailSender* sloužící k zasílání emailů. Bean uvnitř obsahuje vytvoření instance třídy *JavaMailSenderImpl*, která implementuje rozhraní *JavaMailSender*. Této instanci jsou nastaveny atributy *host*, *port*, *username*, *password* a další properties jako transportní protokol. Tato služba je používaná při zasílání odkazu pro nastavení hesla k účtu.

Třída *SwaggerConfig* nastavuje Swagger, což je open source nástroj pro návrh, tvorbu, dokumentaci a testování RESTful API. [36]

Třída *BCryptPasswordEncoderConfig* definuje bean *BCryptPasswordEncoder*. Třída *BCryptPasswordEncoder* je implementací rozhraní *PasswordEncoder*, která používá silné BCrypt hashovací funkce. V aplikací slouží pro šifrování a ověření hesel.

Třída *WebConfig* implementuje rozhraní WebMvcConfigurer. Je zde přepsána metoda *addCorsMappings*, ve které je přidáno mapování CORS (Cross-origin resource sharing).

Security

Třída *WebSecurity*, která dědí od abstraktní třídy *WebSecurityConfigurerAdapter*, je implementována k zajištění služby přihlašování, bezpečnosti endpointů a ověřování požadavků. Je zde konfigurován:

- *HttpSecurity*,
- *AuthenticationManagerBuilder*,
- *AuthenticationManager* bean – Používán službou login pro ověření přihlášení.

HttpSecurity objektu je přidán filtr, což je objekt třídy *JWTAuthorizationFilter*, která dědí od *BasicAuthenticationFilter*. V implementaci této třídy je přepsána metoda *doFilterInternal*. V ní se provádí kontrola hlavičky požadavku, kde se hledá JWT, v případě že se najde, provede se jeho validace a objektu *SecurityContext* se nastaví autentizace získaná z tokenu, v případě že se nenajde, nenastavuje se nic. Nakonec se zavolá metoda *doFilter* nad objektem třídy *FilterChain*.

Pro *HttpSecurity* je dále nastaven výše zmíněný *ChainFilter*, který zakazuje/povoluje přístup k endpointům na základě autorizace požadavku. Strategie při vytváření filtru byla povolit požadavky na endpointy bez nutné autorizace např: login. Zbylé požadavky ověřit na základě role uživatele a zamítnout, či povolit přístup.

Pro *AuthenticationManagerBuilder* se nastavuje *UserDetailsService*, jeho implementace je více popsána v kapitole 4.3.2 Servisní vrstva, a používaný *PasswordEndcoder*

Application properties

V application.properties souboru je uloženo databázové připojení pomocí properties:

- spring.datasource.url,
- spring.datasource.username,
- spring.datasource.password.

Dále je zde spring.jpa.database, který nastavuje typ databázového serveru. Poslední položkou je spring.jpa.hibernate.ddl-auto, který nastavuje inicializaci databáze. Jako hodnotu přijímá enum:

- update – Mění schéma databáze, pokud je potřeba.
- create – Vytvoří nové schéma databáze a smaže předchozí data a strukturu.

- create-drop – Podobné jako create s tím, že smaže schéma na konci session.
- none – Zakuje DDL (Data Definition Language) úpravy.
- validate – Pouze validuje schéma a nedělá žádné změny.

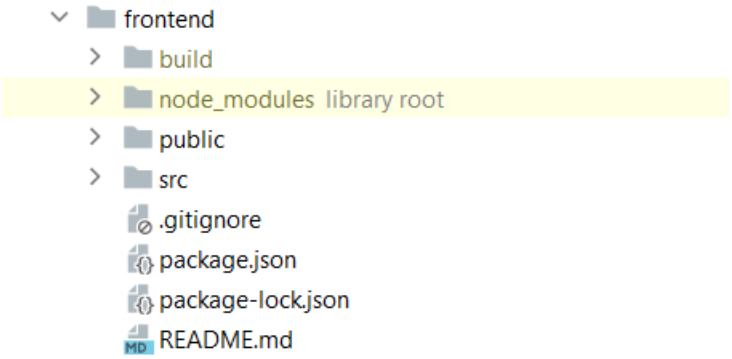
[10]

Pom

V pom jsou uloženy základní informace o aplikaci jako je název aplikace, popis, verze, groupId, artifactId, Spring Boot verze, Java verze, které byly nastaveny při vytváření projektu. Dále obsahuje všechny dependencies a build. Build byl ponechán výchozí po vytvoření projektu. Dependencies byly postupně přidávány při vývoji například knihovna Lombok.

4.4 Frontend

Tato podkapitola popisuje implementaci frontendu neboli uživatelské webové aplikace. Zaměřuje se na strukturu a popis nejdůležitějších souborů. Následující obrázek zobrazuje strukturu.



Obrázek 9: Frontend struktura

Ve složce build jsou obsaženy soubory, které jsou vytvořeny po buildu (sestavení) projektu. V této složce se nachází všechny potřebné soubory pro nasazení aplikace. Build se provádí převedením JSX do čistého JavaScriptu a minifikací. Ve složce node_modules jsou obsaženy všechny nainstalované dependency/moduly v projektu. Soubor package.json obsahuje informace o projektu (jméno, verzi) a také všechny potřebné moduly.

Složka src dále obsahuje:

- index.js
- App.js,
- složku components,
- složku service.

index.js

Index.js je JavaScriptový soubor, který koresponduje s index.html a slouží jako kořenový soubor pro React aplikaci. Soubor index.html se nachází ve složce public a je jediným html souborem celé aplikace. Obsahuje pouze základní html strukturu, kde se v jeho těle nachází tag * *, do kterého se ze souboru index.js načítá komponenta *App*, pomocí příkazu *ReactDOM.render(<App />, document.getElementById('root')).*

App.js

Tento soubor obsahuje *App* komponentu, která je hlavní komponentou celé React aplikace a slouží jako kontejner pro všechny ostatní. V této třídě je v aplikaci zejména řešeno směrování pomocí komponent z knihovny react-router-dom:

- Router – Hlavní obalující komponenta směrování.
- Link – Zajišťuje deklarativní navigaci aplikace.
- Switch – Vrací první komponentu Route, nebo Redirect, která odpovídá aktuální cestě.
- Route – Renderuje komponentu, obsahuje cestu podle, které switch vybírá shodu.

Components

Ve složce components jsou uloženy všechny vytvořené komponenty aplikace. Komponenty jsou obecné znovupoužitelné blok kódu, které se mezi sebou skládají a vytvářejí tak celek. Příkladem obecnosti může být například třída *AddEditStudent*, která zajišťuje jak přidávání, tak editaci studenta, jelikož jak při přidávání, tak při editaci je použit stejný formulář, rozdíly chování nastanou pouze při a po odeslání požadavku. Příkladem znuvupoužitelnosti může být komponenta *PaginationComponent*, která vytváří stránkování. Příkladem skládání komponent můžou být komponenty:

- *StudentList* – Komponenta, která zobrazuje seznam studentů a další prvky stránky.

- *StudentListComponent* – Komponenta, která vytvoří seznam studentů na základě předaného parametru.
- *PaginationComponent* – Komponenta, která vytvoří stránkování seznamu.

Service

Ve složce service se nachází funkční komponenta *UserProfile* a tři třídy:

- *AuthService*,
- *AxiosUtils*,
- *Server*

Třída *AuthService* zajišťuje ověřování požadavku, obsahuje metodu *getToken*, která vrací uložený JWT v local storage. Třída *AxiosUtils* obsahuje metody vytvářející požadavky na server pomocí klienta axios. Tyto metody jsou rozděleny podle způsobu autorizace a typu http požadavku. Autorizované požadavky, obsahují JWT, který uložen v hlavičce v parametru *Bearer*, pomocí něhož se provádí ověření na serverové straně. Vkládání probíhá přes výše zmíněnou metodu *getToken*. Neautorizované neobsahují žádný ověřovací parametr. Dále podle typu http požadavku:

- post,
- put,
- get,
- delete.

Každá z vytvořených metod přijímá parametr url, parametr body obsahují pouze metody post a put, které mají ve své http zprávě tělo.

Třída *Server* obsahuje konstantu *USER_API_BASE_URL*, ve které je uložena url adresa serveru. Pomáhá tak k jednotnému použití a vyhnutí se na pevno napsaných adres v kódu, což umožňuje snazší správu v případě změny.

Třída *UserProfile* slouží jako kontejner pro uložení uživatelského profilu. Obsahuje atributy *id*, *username*, *token*, *roles*, které ukládá do *localStrorage*. Tyto atributy jsou naplněny při přihlášení uživatele a odstraněny při odhlášení.

[1], [26]

5 TESTOVÁNÍ ZDROJOVÝCH KÓDŮ

Tato kapitola je zaměřená na řešení problematiky testování zdrojových kódů a její implementace. Vyvíjená aplikace poskytuje tři typy testovacích úloh – základní úlohy, úlohy s JUnit testem a úlohy třídy a rozhraní s JUnit testem. Každý typ bude detailně popsán v následujících kapitolách. Výhodou všech třech typů testování je použití runtime kompilace. Kód je kompilován dynamicky a je načítán z databáze, nebo z příchozích požadavků. Nic není zapisováno ani čteno ze souborů. Pomocí vláken je zajištěno paralelní zpracování některých úkonů pro rychlejší zpracování a optimalizaci, dále také aby kompilace a provedení testů netrvala déle maximální stanovenou dobu.

5.1 Základní úloha

Tento typ úloh nabízí jednoduché a rychlé řešení vytváření testovacích scénářů pro jednu metodu. Je určen zejména pro začátečnické a trochu pokročilejší úlohy. Nabízí plný výstup testu s výsledky všech testovacích scénářů. Omezení toho typu testů je práce pouze s hodnotovými datovými typy a datovým typem *String*. Na vstupu v parametru metody je však možné použít i pole, *ArrayList*, *Map* a *Set*.

Vytvoření úlohy

Nejdříve je nutné vyplnit povinné pole názvu a zadání úlohy. Dále je zde volitelný obrázek, který může sloužit jako doplnění zadání. Poté název testované metody, který se musí shodovat s názvem testované metody v kódu. Pak vyplnění datových typů parametrů, opět se musí shodovat s typy parametrů metody v kódu. Příklady zápisu parametrů:

- *String; String*,
- *int; int*,
- *String[]* – pro pole,
- *ArrayList,*
- *Map,*
- *Set.*

Jednotlivé datové typy se oddělují středníkem. Pole předpis metody a testování řešení jsou nepovinná. Předpis bude studentovi zobrazen při otevření úlohy. Testování řešení umožňuje ověřit vlastní kód vůči vytvořeným testovacím scénářům a ověření správnosti úlohy.

Pro vytvoření jednoho testovacího scénáře stačí přidat jeden testovací řádek, ve kterém je třeba vyplnit hodnoty na vstupu (oddělené středníkem) a hodnotu na výstupu do input oken. Podoba vstupu může být např:

- *String; String* – "hodnota 1";"hodnota 2".
- *int; int* – 1;2.
- *String[]* – (Object) new String[]{"hodnota1","hodnota2"}.
- *ArrayList* – (Object) new ArrayList<String>(Arrays.asList("hodnota1","hodnota2")).

Implementace

Implementace testování těchto úloh se nachází ve třídě *CodeTestServiceImpl* v metodě *runBasic*. Nejprve je zkompilována testovaná třída, k tomu je použita metoda *compile* z třídy *CodeCompiler*. Poté jsou zpracovány zadané parametry uživatelem a provede se kompilace testovací třídy, což v tomto případě není třída s JUnit testy. Je to připravená třída s metodami, kde každá metoda vytváří jeden testovací scénář a přijímá objekt testované třídy a její metodu, ta je pak spuštěna pomocí reflexe (metoda invoke) a z jejího výstupu je vytvořen výsledek. Tyto výsledky se ukládají do objektu třídy *TestResult*, což je vytvořená třída sloužící jako kontejner pro výsledky testů.

5.2 Úloha s JUnit testem

Tento typ úloh nabízí vytváření mnohem pokročilejších úloh, než tomu bylo u těch základních. Umožňuje vytváření testovacích scénářů pro jednu, více metod, či vytvoření celého těla třídy (atributy, metody, konstruktory). Výstup z testu je bez úpravy uživatelem pouze procentuální a zobrazuje úspěšnost/neúspěšnost každého testovacího scénáře bez popisu. Popis je možné libovolně dopsat pro každou testovací metodu.

Vytvoření úlohy

Úplně stejně jako u základní úlohy je zde povinný název, zadání a nepovinný obrázek a předpis. Dále je zde zobrazen název třídy, který je konstantní a je pouze informativní, pro korektní použití v JUnit testech. Poté je zde testovací řešení, ve kterém je opět možno ověřit vlastní kód vůči vytvořeným testovacím scénářům a ověření správnosti úlohy. V poslední řadě je vytvoření samotné testovací třídy. Pro vytvoření testovacího scénáře stačí přidat novou testovací metodu podle vzoru. Vytvoření popisu metody pro výstup je možno provést přes anotaci @DisplayName(„popis“).

Implementace

Implementace testování těchto úloh se nachází ve třídě *CodeTestServiceImpl* v metodě *runJUnit*. Nejprve je nutné zkompilovat prázdnou testovací třídu a prázdnou testovanou třídu právě v tomto pořadí, jelikož je nezbytné smazat vazby, které mezi sebou tyto třídy mají. Poté je teprve možné zkompilovat testovanou třídu. Kompilace jsou zajištěny pomocí knihovny *net.openhft.compiler.CompilerUtils*, kde je použita třída *ChachedCompiler* jako kompilátor. Z původní knihovny byla tato třída upravena pro lepší zacházení s výjimkami při kompilaci. Dále je zkompilován kód testovací třídy. Následně je z testovací třídy získáno pole deklarovaných metod, přes které je provedena iterace a každá metoda je předána jako argument metodě *run* ze třídy *JUnitCore*, která zajistí spuštění testovací metody a vrátí výsledek v objektu třídy *Result*. Z něho jsou získány výsledky testu a uloženy do objektu třídy *TestResult*.

5.3 Úloha třídy a rozhraní s JUnit testem

Oproti předchozí úloze s JUnit testem, jsou úlohy v této kategorii rozšířeny o implementaci celé třídy, ne pouze jejího těla a také použití rozhraní. Nabízí stejný výstup jako úlohy s JUnit testem, výstup je bez úpravy uživatelem pouze procentuální a zobrazuje úspěšnost/neúspěšnost každého testovacího scénáře bez popisu. Popis je možné libovolně dopsat pro každou testovací metodu.

Vytvoření úlohy

Stejně jako u základní úlohy a úlohy s JUnit testem je zde povinný název, zadání a nepovinný obrázek a předpis. Dále je zde zobrazen název třídy a název rozhraní, které jsou konstantní. Nově je tu kolonka rozhraní, ve kterém je nutné vytvořit interface. Testovací řešení je opět pro ověření správnosti. Implementace testovací třídy je stejná jako pro úlohy s JUnit testy, testovací scénáře jsou vytvořeny přidáním metody a popis je možno doplnit přes anotaci @DisplayName.

Implementace

Implementace testování těchto úloh je skoro totožná s implementací zmíněnou v předchozí kapitole pro úlohy s JUnit testem. Rozdíl je, že po kompilaci prázdných tříd je nutné ještě zkompilovat rozhraní a pak až testovanou třídu.

Následující ukázka kódu zachycuje nejdůležitější část metody *runJUnitWithInterface*, která zajišťuje běh testů pro úlohy typu třídy a rozhraní s JUnit testem. Nejprve se zkompiluje prázdná testovací třída a prázdná testovaná třída. Poté se zkompiluje rozhraní a testovaná třída, která už obsahuje implementované tělo. Pokud je zachycena výjimka, metoda vrací adekvátní výsledek. Dále se spouští metoda *getTestResult*, která zajišťuje spuštění a vyhodnocení testovacích scénářů, její provedení a výsledek je zajištěn pomocí rozhraní Future.

```
…
try {
    codeCompiler.compile("public class " + testClassName + "{}", testClassName);
    codeCompiler.compile("public class " + testedClassName + "{}", testedClassName);
    codeCompiler.compile(interfaceBody, interfaceName);
    codeCompiler.compile(code, testedClassName);
} catch (Exception e) {
    …
    return testResult;
}
Future<TestResult> futureTestResult = getTestResult(exercise);
try {
    return futureTestResult.get(timeout, TimeUnit.MILLISECONDS);
} catch …
return testResult;
```

Ukázka zdrojového kódu 4: runJUnitWithInterface

Následující ukázka kódu zachycuje nejdůležitější část metody *getTestResult*, která spouští a vyhodnocuje testovací scénáře. Pomocí rozhraní *ExecutorService* a jeho metody *submit* je spuštěn task, který se nachází vně lamda výrazu. V tom je nejdříve zkompilována testovací třída, z ní jsou pak načteny do pole metod všechny deklarované metody. Dále jsou pomocí třídy *JUnitCore* a její metody *run* spouštěny jednotlivé metody, které testovací třída obsahuje. Výsledky jsou pak zpracovány a předány zpět.

```
return executorService.submit(() -> {
    Class<?> testClass = codeCompiler.compile(testClassBody, testClassName);
    Method[] methods = testClass.getDeclaredMethods();
    JUnitCore junit = new JUnitCore();
    for (Method method : methods) {
        Result run = junit.run(Request.method(testClass, method.getName()));
        …
    }
    return testResult;
});
```

Ukázka zdrojového kódu 5: getTestResult

6 NASAZENÍ APLIKACE

V této kapitole je popsán postup nasazení aplikace na server. Jsou zde detailně popsány všechny potřebné instalace a kroky pro úspěšné spuštění aplikace.

6.1 Příprava ke spuštění

Pro spuštění celého systému je nutno nainstalovat JDK 11, MySQL, Apache HTTP Server a Apache Tomcat. Pro Apache Tomcat je nutné přiřazení JDK 11, pokud se tak nestane automaticky při instalaci, je nutné ve složce /bin/catalina.bat nakonfigurovat pomocí *set JAVA_HOME = <cesta k JDK>*. Dále se pomocí nástroje pro správu databáze naimportuje SQL skript, který vytvoří schéma databáze, strukturu a vloží potřebná data pro spuštění.

6.2 Spuštění backendu

Nejprve je nutné přepsat několik hodnot v souboru *application.properties*. Hodnoty v tomto souboru nejsou uvozovány uvozovkami ani apostrofy. Jako první property *frontend.baseUrl*, kde se musí nastavit veřejná url adresa aplikace zakončená znakem „/“. Dále se musí nastavit *datasource.username* a *datasource.password* pro specifikaci přístupu do databáze.

Poté už je možné provést build aplikace pomocí nástroje Apache Maven. Pro sestavení výsledného war souboru slouží příkaz mvn package, který se spustí v adresáři projektu (pokud není cesta k mvn nastavena v systémové proměnné PATH, je nutné uvést v příkazu celou cestu).

War soubor pak jen stačí vložit do složky webapps, která se nachází ve složce, do které byl nainstalován Apache Tomcat. Jeho spuštění se pak provede pomocí spuštění souboru startup.bat, který se nachází ve složce bin (případně alternativně příkazem v konzoli *catalina.bat run*). Backend by měl být spuštěn v cestě localhost:8080/teco.

6.3 Spuštění frontendu

Pro spuštění frontendu bylo také nejprvé nutné provést build aplikace pomocí balíčkovacího nástroje NPM, který byl součástí instalace Node.js. Pro sestavení produkčních souborů byl použit příkaz npm run build (pokud není cesta k mvn nastavena v systémové proměnné PATH, je nutné uvést v příkazu celou cestu). Do složky build by dále přidán soubor .htaccess, který zařídí správné routování. Celý obsah této složky pak stačí vložit do Apache HTTP Serveru do složky /var/www/teco (případně /htdocs/teco). Spuštění serveru se provede pomocí httpd.exe.

ZÁVĚR

V rámci této diplomové práce byl vytvořen funkční systém, který umožňuje automatickou kontrolu odevzdaných zdrojových kódů. Hlavní cíle práce, kterými byly navržení, implementace a nasazení webové aplikace do výuky, byly uskutečněny.

Výsledná aplikace splňuje všechny vytyčené požadavky ze zadání. Umožňuje vytváření testovacích scénářů pro jazyk Java SE s podporou pro práci s primitivními i referenčními datovými typy, správu předmětů a cvičení, správu uživatelů a poskytnutí zpětné vazby vyučujícím a studentům k odevzdaným kódům.

Oproti původně plánovanému využití ověřovacího systému, který momentálně používá např. aplikace IS/STAG, byl implementován separátní systém, který ověřuje uživatele na základě dat uložených v databázi aplikace.

Nasazení systému úspěšně proběhlo na nově zřízený univerzitní server, na kterém je spuštěn frontend, backend i databáze aplikace. Systém je připraven sloužit jako doplňující nástroj k předmětu Základy programování, případně i k dalším předmětům s podobným zaměřením, které jsou vyučovány v jazyce Java.

Práce by se dále mohla rozšířit o další možnosti testování, jako vytvoření nové skupiny úloh, které by testovaly více než jednu třídu, případně i celé projekty. Další možností by mohlo být rozšíření škály programovacích jazyků, ve kterých by úlohy byly řešeny. Tím by aplikace mohla být přínosem i v předmětech, které nejsou v jazyce Java vyučovány.

POUŽITÁ LITERATURA

[1] A quick guide to help you understand and create ReactJS apps. FreeCodeCamp Programming Tutorials [online]. Sridhar, 2018 [cit. 2021-5-12]. Dostupné z: https://www.freecodecamp.org/news/quick-guide-to-understanding-and-creating-reactjs-apps-8457ee8f7123/

[2] Angular [online]. c2010-2021 [cit. 2021-5-12]. Dostupné z: https://angular.io/

[3] Baeldung [online]. [cit. 2021-5-12]. Dostupné z: https://www.baeldung.com/

[4] Black Box Testing vs. White Box Testing vs. Grey Box Testing. Javatpoint [online]. c2011-2018 [cit. 2021-5-12]. Dostupné z: https://www.javatpoint.com/black-box-testing-vs-white-box-testing-vs-grey-box-testing

[5] Code Driven Testing. Tutorialspoint [online]. c2021 [cit. 2021-5-12]. Dostupné z: https://www.tutorialspoint.com/software_testing_dictionary/code_driven_testing.htm

[6] Codewars [online]. c2021 [cit. 2021-5-12]. Dostupné z: https://www.codewars.com/

[7] CodingBat [online]. Parlante, 2017 [cit. 2021-5-12]. Dostupné z: https://codingbat.com/

[8] DAO Design Pattern. JournalDev [online]. [cit. 2021-5-12]. Dostupné z: https://www.journaldev.com/16813/dao-design-pattern

[9] Data Access Object (DAO) design pattern in Java. Javarevisited [online]. Paul [cit. 2021-5-12]. Dostupné z: https://javarevisited.blogspot.com/2013/01/data-access-object-dao-design-pattern-java-tutorial-example.html#axzz6rvFbRwKQ

[10] Database initialization. Spring [online]. [cit. 2021-5-12]. Dostupné z: https://docs.spring.io/spring-boot/docs/1.1.0.M1/reference/html/howto-database-initialization.html

[11] Difference among Black Box, White Box and Grey Box Testing. Medium [online]. Copper, 2018 [cit. 2021-5-12]. Dostupné z: https://medium.com/@mccccc/difference-among-white-box-black-box-and-grey-box-testing-35482292473f

[12] Difference Between Black-Box, White-Box, and Grey-Box Testing: Programming & DevOps news, tutorials & tools. DZone [online]. Smith, 2020 [cit. 2021-5-12]. Dostupné z: https://dzone.com/articles/difference-between-black-box-white-box-and-grey-bo

[13] GREŠÁK, Viktor. REST na platformě Java [online]. Pardubice, 2016 [cit. 2021-5-12]. Dostupné z: https://dk.upce.cz/handle/10195/64875. Bakalářská práce. Univerzita Pardubice.

[14] Integration testing. Javatpoint [online]. c2011–2018 [cit. 2021-5-12]. Dostupné z: https://www.javatpoint.com/integration-testing

[15] Integration Testing. SOFTWARE TESTING Fundamentals [online]. STF, 2020 [cit. 2021-5-12]. Dostupné z: https://softwaretestingfundamentals.com/integration-testing/#Method

[16] Integration Testing: What is, Types, Top Down & Bottom Up Example. Meet Guru99 [online]. c2021 [cit. 2021-5-12]. Dostupné z: https://www.guru99.com/integration-testing.html

[17] JavaScript.com [online]. c2016-2021 [cit. 2021-5-12]. Dostupné z: https://www.javascript.com/

[18] JUnit 5 [online]. c2021 [cit. 2021-5-12]. Dostupné z: https://junit.org/junit5/

[19] Learn Java. Javatpoint [online]. c2011-2018 [cit. 2021-5-12]. Dostupné z: https://www.javatpoint.com/java-tutorial

[20] Maven [online]. c2002-2021 [cit. 2021-5-12]. Dostupné z: https://maven.apache.org/

[21] Mock Testing. Devopedia [online]. 2019 [cit. 2021-5-12]. Dostupné z: https://devopedia.org/mock-testing

[22] Mockito framework site [online]. [cit. 2021-5-12]. Dostupné z: https://site.mockito.org/

[23] MySQL: Developer Zone [online]. c2021 [cit. 2021-5-12]. Dostupné z: https://dev.mysql.com/

[24] PORCELLO E, BANKS A. Learning React: Modern Patterns for Developing React Apps. O'Reilly Media; 2nd Edition, 2020, 310 pp. ISBN: 978-1492051725.

[25] Project Lombok [online]. c2009-2021 [cit. 2021-5-12]. Dostupné z: https://projectlombok.org/

[26] React Router: Declarative Routing for React.js [online]. c2021 [cit. 2021-5-12]. Dostupné z: https://reactrouter.com/

[27] React: A JavaScript library for building user interfaces [online]. c2021 [cit. 2021-5-12]. Dostupné z: https://reactjs.org/

[28] Selenium Tutorial. Javatpoint [online]. c2011-2018 [cit. 2021-5-12]. Dostupné z: https://www.javatpoint.com/selenium-tutorial

[29] SeleniumHQ Browser Automation [online]. c2021 [cit. 2021-5-12]. Dostupné z: https://www.selenium.dev/

[30] SOAP vs REST APIs: Which Is Right For You? SoapUI [online]. c2021 [cit. 2021-5-12]. Dostupné z: https://www.soapui.org/learn/api/soap-vs-rest-api/

[31] SOAP vs. REST: The Differences and Benefits Between the Two Widely-Used Web Service Communication Protocols. Stackify [online]. ALTVATER, 2017 [cit. 2021-5-12]. Dostupné z: https://stackify.com/soap-vs-rest/

[32] Spring [online]. c2021 [cit. 2021-5-12]. Dostupné z: https://spring.io/

[33] Spring Beans in Depth. Medium [online]. Santis, 2020 [cit. 2021-5-12]. Dostupné z: https://medium.com/javarevisited/spring-beans-in-depth-a6d8b31db8a1

[34] Spring Data JPA @Query. Baeldung [online]. [cit. 2021-5-12]. Dostupné z: https://www.baeldung.com/spring-data-jpa-query

[35] Spring MVC Tutorial. Javatpoint [online]. c2011-2018 [cit. 2021-5-12]. Dostupné z: https://www.javatpoint.com/spring-mvc-tutorial

[36] Swagger: API Documentation & Design Tools for Teams [online]. c2021 [cit. 2021-5-12]. Dostupné z: https://swagger.io/

[37] TURNQUIST L. G.Learning Spring Boot 2.0 - Second Edition: Simplify the development of lightning fast applications based on microservices and reactive programming. Packt Publishing, 2017, 370 pp. ISBN978-1786463784.

[38] Tutorialspoint: Data Access Object Pattern [online]. c2021 [cit. 2021-5-12]. Dostupné z: https://www.tutorialspoint.com/design_pattern/data_access_object_pattern.htm

[39] Tutorialspoint: MVC Framework Tutorial [online]. c2021 [cit. 2021-5-12]. Dostupné z: https://www.tutorialspoint.com/mvc_framework/

[40] Tutorialspoint: SOAP Tutorial [online]. c2021 [cit. 2021-5-12]. Dostupné z: https://www.tutorialspoint.com/soap/

[41] Tutorialspoint: Spring Boot Tutorial [online]. c2021 [cit. 2021-5-12]. Dostupné z: https://www.tutorialspoint.com/spring_boot/

[42] Tutorialspoint: WSDL Tutorial [online]. c2021 [cit. 2021-5-12]. Dostupné z: https://www.tutorialspoint.com/wsdl/

[43] TutorialsTeacher: Learn Web Technologies [online]. c2020 [cit. 2021-5-12]. Dostupné z: https://www.tutorialsteacher.com/

[44] TypeScript: Typed JavaScript at Any Scale [online]. c2012-2021 [cit. 2021-5-12]. Dostupné z: https://www.typescriptlang.org/

[45] Unit tests with Mockito - Tutorial. Eclipse, Android and Java training and support [online]. Vogel, 2021 [cit. 2021-5-12]. Dostupné z: https://www.vogella.com/tutorials/Mockito/article.html

[46] Usage statistics of server-side programming languages for websites. W3Techs [online]. c2009-2021 [cit. 2021-5-12]. Dostupné z: https://w3techs.com/technologies/overview/programming_language

[47] W3Schools Online Web Tutorials: CSS Tutorial [online]. c1999-2021 [cit. 2021-5-12]. Dostupné z: https://www.w3schools.com/css/

[48] W3Schools Online Web Tutorials: HTML Tutorial [online]. c1999-2021 [cit. 2021-5-12]. Dostupné z: https://www.w3schools.com/html/

[49] W3Schools Online Web Tutorials: React Tutorial [online]. c1999-2021 [cit. 2021-5-12]. Dostupné z: https://www.w3schools.com/react/

[50] W3Schools Online Web Tutorials: SQL Tutorial [online]. c1999-2021 [cit. 2021-5-12]. Dostupné z: https://www.w3schools.com/sql/

[51] Web MVC framework. Spring [online]. [cit. 2021-5-12]. Dostupné z: https://docs.spring.io/spring-framework/docs/3.2.x/spring-framework-reference/html/mvc.html

[52] Web Service Definition Language (WSDL). World Wide Web Consortium (W3C) [online]. 2001 [cit. 2021-5-12]. Dostupné z: https://www.w3.org/TR/wsdl.html

[53] What is code testing and why is it important? Zeomag [online]. Prachi Manchanda, 2017 [cit. 2021-5-12]. Dostupné z: https://www.zeolearn.com/magazine/what-is-code-testing-and-why-is-it-important

[54] What Is Integration Testing (Tutorial With Integration Testing Example). Software Testing Help [online]. 2021 [cit. 2021-5-12]. Dostupné z: https://www.softwaretestinghelp.com/what-is-integration-testing/

[55] What is SOAP API: Formats, Protocols, and Architecture. AltexSoft [online]. 2019 [cit. 2021-5-12]. Dostupné z: https://www.altexsoft.com/blog/engineering/what-is-soap-formats-protocols-message-structure-and-how-soap-is-different-from-rest/

[56] Working with Spring Data Repositories. Spring [online]. [cit. 2021-5-12]. Dostupné z: https://docs.spring.io/spring-data/jpa/docs/1.5.0.RELEASE/reference/html/repositories.html

PŘÍLOHY

PŘÍLOHA A – INSTALAČNÍ CD

Instalační CD obsahuje zdrojové soubory aplikace a SQL soubor pro vytvoření databáze.