Automatizované testování znalostí

Bc. Patrik Řepa

Diplomová práce
2019

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
akademický rok: 2018/2019

ZADÁNÍ DIPLOMOVÉ PRÁCE
(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Bc. Patrik Řepa**
Osobní číslo:	**A17247**
Studijní program:	**N3902 Inženýrská informatika**
Studijní obor:	**Počítačové a komunikační systémy**
Forma studia:	**prezenční**
Téma práce:	**Automatizované testování znalostí**
Téma anglicky:	**Automated Skill Testing**

Zásady pro vypracování:

1. **Vytvořte literární rešerši na dané téma.**
2. **Popište použité technologie a proveďte návrh aplikace pro automatizované testování znalostí.**
3. **Navhněte vhodnou architekturu aplikace, umožňující testovat více programovacích jazyků.**
4. **Realizujte navrženou aplikaci v prostředí .NET.**
5. **Navrhněte a vytvořte testovací úlohy pro vytvořenou aplikaci a to s ohledem na možné využití při výuce předmětů zaměřených na vývoj software.**
6. **Testováním ověřte funkčnost aplikace a proveďte celkové zhodnocení navrženého systému.**

Rozsah diplomové práce:

Rozsah příloh:

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

Seznam odborné literatury:

1. **GALLOWAY, Jon, Brad WILSON, K. Scott ALLEN a David MATSON. Professional ASP.NET MVC 5. Indianapolis, IN: Wrox, a Wiley brand, [2014]. Wrox professional guides. ISBN 978-1-118-79475-3.**
2. **PEHLIVANIAN, Ara a Don NGUYEN. JavaScript Okamžitě: Ovládněte JavaScript za víkend. Computer Press, 2014. ISBN 978-80-251-4163-2.**
3. **EVJEN, Bill, Christian NAGEL, Jay GLYNN, Morgan SKINNER a Karli WATSON. C# 2008 Programujeme profesionálně. COMPUTER PRESS, 2009. ISBN 978-80-251-2401-7.**
4. **FENTON, Steve. Pro typescript. 2nd ed. New York, NY: Springer Science Business Media, 2017. ISBN 978-148-4232-484.**
5. **MACDONALD, Matthew, Adam FREEMAN a Mario SZPUSZTA. ASP.NET 4 a C# 2010: tvorba dynamických stránek profesionálně. Brno: Zoner Press, 2011. Encyklopedie Zoner Press. ISBN 978-80-7413-131-8.**

Vedoucí diplomové práce:	**Ing. Milan Navrátil, Ph.D.** Ústav elektroniky a měření
Konzultant:	**Ing. František Bolf** Ústav počítačových a komunikačních systémů
Datum zadání diplomové práce:	**30. listopadu 2018**
Termín odevzdání diplomové práce:	**17. května 2019**

Ve Zlíně dne 14. prosince 2018

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

doc. Mgr. Milan Adámek, Ph.D.
děkan

Ing. Miroslav Matýsek, Ph.D.
ředitel ústavu

Prohlašuji, že

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

Prohlašuji,

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

Ve Zlíně, dne 21. 5. 2019 Patrik Řepa, v. r.

ABSTRAKT

Tato diplomová práce se zabývá návrhem a realizací webové aplikace pro testování programovacích jazyků. Teoretická část diplomové práce obsahuje způsoby návrhu architektury softwaru, použité technologie pro vývoj webové aplikace a popis editorů pro editaci zdrojového kódu ve webovém prostředí. Praktická část obsahuje návrh architektury softwaru a popis implementace vytvořené webové aplikace.

Klíčová slova: programovací jazyk, webová aplikace, SOLID, Dependency Injection, .NET Framework, JavaScript, SQL, Monaco editor

ABSTRACT

This diploma thesis deals with the design and implementation of web application for testing programming languages. The theoretical part contains ways of designing software architecture, used technologies for web application development and description of editors for editing source code in web environment. The practical part contains design of software architecture and description of created web application implementation.

Keywords: Programming Language, Web Application, SOLID, Dependency Injection, .NET Framework, JavaScript, SQL, Monaco Editor

Na tomto místě bych rád poděkoval za odborné rady a pomoc při realizaci diplomové práce panu Ing. Františku Bolfovi. Dále mé rodině a přátelům za důvěru, kterou do mne vkládají.

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

OBSAH

ÚVOD

V dnešní době existuje mnoho programovacích jazyků, určených pro vývoj webových aplikací, avšak jen část se z nich stane použitelná pro praxi. Vývojář webových aplikací by měl tyto programovací jazyky ovládat. Vytvořená webová aplikace by měla umožňovat testování znalostí těchto programovacích jazyků. Může soužit pro testování studentů, kteří se učí C#, JavaScript nebo SQL, nebo může být použita při přijímacím pohovoru do firmy, která se zabývá vývojem aplikací v těchto jazycích.

Teoretická část diplomové práce obsahuje základní popis programovacích jazyků a jejich dělení podle míry abstrakce nebo podle programovacího paradigmatu. Jsou zde popsány programovací jazyky, které jsou použity pro vývoj samotné webové aplikace a také programovací jazyky, které budou předmětem testů. Další část teoretické části se zabývá správným návrhem architektury softwaru, ať už se jedná například o SOLID principy, nebo využití techniky zvané dependency injection. Jsou popsány použité technologie, na kterých bude aplikace postavena. V poslední části jsou popsány nejčastěji používané editory zdrojových kódu pro webové aplikace.

Praktická část popisuje implementaci samotného softwaru. Od návrhu architektury softwaru, přes návrh databáze až po implementaci jednotlivých funkcionalit webové aplikace. Druhá půle praktické části se zabývá způsobem dynamické kompilace programovacích jazyků, které jsou předmětem testů. Nacházejí se zde použité technologie pro kompilaci a ukázky jejich použití ve vytvořené webové aplikaci.

I. TEORETICKÁ ČÁST

1 PROGRAMOVACÍ JAZYK

Programovací jazyk je médium, které nám slouží pro komunikaci s počítačovým systémem. V dnešní době existuje mnoho programovací jazyků. Každý programovací jazyk má své specifika a tím pádem také své výhody a nevýhody. Obvykle lze programovací jazyky zařadit do několika skupin. Nicméně každý jazyk může podporovat více paradigmat a proto se jednotlivé skupiny prolínají. Každým rokem se implementují nové programovací jazyky, avšak jen malá část se stane populární na tolik, aby je mohl programátor využít ve své kariéře [1].

Dělení programovacích jazyků dle abstrakce:

- Vyšší programovací jazyky
    - Imperativní
        - Strukturované
        - Objektově orientované
    - Deklarativní
        - Funkcionální
        - Logické
- Nižší programovací jazyky

Dělení programovacích jazyků dle způsobu překladu a spuštění:

- Kompilované programovací jazyky
- Interpretované programovací jazyky

1.1 Nižší programovací jazyky

Tento typ jazyka poskytuje malou, nebo dokonce žádnou abstrakci od toho, jak funguje procesor počítače. Programy napsané v tomto jazyce jsou těžko čitelné a udržovatelné. Nevýhoda těchto jazyků je ta, že jsou závislé na platformě. Patří sem například strojový kód a assembler [2].

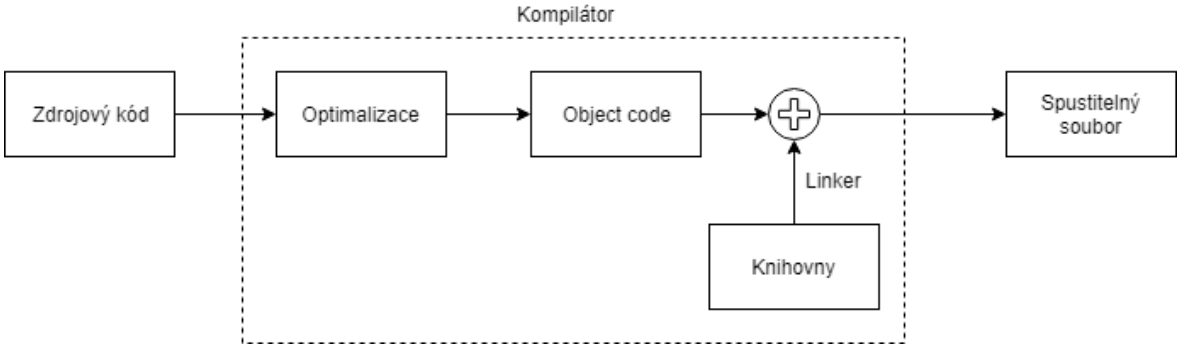
1.2 Vyšší programovací jazyky

Jedná se o jazyky, které jsou nezávislé na platformě a jejich hlavní výhoda je ta, že programy napsané v těchto jazycích lze snadno číst, upravovat a rozšiřovat. Tyto jazyky mají funkce a knihovny tříd, díky kterým lze jednodušeji, a hlavně rychleji psát program. Výsledný jazyk je následně kompilován, případně interpretován do strojového kódu [3].

1.2.1 Kompilované programovací jazyky

Programátor píše program v jazyce, kterému rozumí. Avšak tomuto jazyku nedokáže porozumět procesor, proto je zde kompilátor. Kompilátor přeloží zdrojový kód do strojového kódu, který již dokáže procesor zpracovat. Tento proces se může spustit několikrát, aby se dosáhlo optimalizovaného kódu pro procesor.

Kompilátor zkompiluje zdrojový kód, optimalizuje ho a vznikne object kód. Tento kód ještě není spustitelný, musí se připojit dynamické knihovny. Tyto knihovny jsou připojeny pomocí linkeru. Finální produkt je spustitelný soubor. Do téhle skupiny programovacích jazyků patří C, C++, Pascal nebo také Smalltalk [4].



Obr. 1. Schéma kompilátoru.

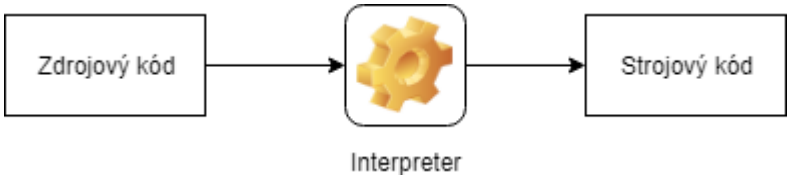
1.2.2 Interpretované programovací jazyky

Interpretovaný jazyk se vyznačuje tím, že není přeložen před spuštěním programu, na rozdíl od kompilovaného jazyka. Jednoduchá instrukce interpretovaného jazyka může být spuštěna bez kompilace na strojový kód, avšak složitější části kódu je potřeba zkompilovat za běhu programu.

Tento typ jazyků má velkou výhodu, nejsou závislé na platformě nebo operačním systému, a jelikož jsou kompilovány za běhu, je kód optimalizován pro konkrétní platformu, na které

program běží. Další výhodou těchto jazyků je ten, že interpreter dokáže generovat užitečné chybové zprávy, které programátorovi usnadní práci při hledání chyb.

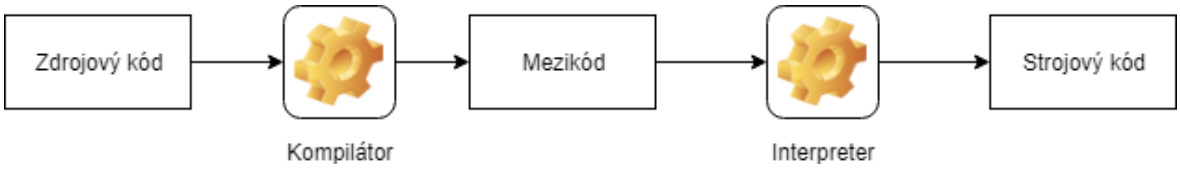
Nevýhoda těchto jazyků je rychlost, jelikož se musí provádět kompilace za běhu programu. V dnešní době však jsou interpretery tak optimalizované, aby výkonnostní ztráty oproti kompilovaným jazykům minimalizovali. Do této skupiny jazyků patří například PHP, JavaScript, Python nebo také Perl [4].



Obr. 2. Interpretace zdrojového kódu.

1.2.3 JIT Kompilace

JIT neboli Just in Time, je typ kompilátoru, který přeloží zdrojový kód do tzv. mezikódu. Tento mezikód je následně interpretován, výsledný program je tedy rychlejší, jelikož je rychlejší interpretovat již zkompilovaný a optimalizovaný kód než zdrojový kód. Tyto kompilátory mají přístup k informacím o běhu programu a jsou schopny monitorovat a optimalizovat spuštěný program. Do této skupiny jazyků patří například Java nebo C# [4].



Obr. 3. JIT kompilace.

1.2.4 Programovací jazyk C#

C# byl vyvinut firmou Microsoft speciálně pro technologii .NET Framework. Jedná se o moderní, jednoduchý, typově bezpečný a objektově orientovaný programovací jazyk, odvozený od jazyka C, C++ a Java.

Vlastnosti jazyka C#:

- Plná podpora tříd, objektově orientované programování, včetně dědičnosti rozhraní a implementace, virtuálních funkcí a přetížení operátorů.

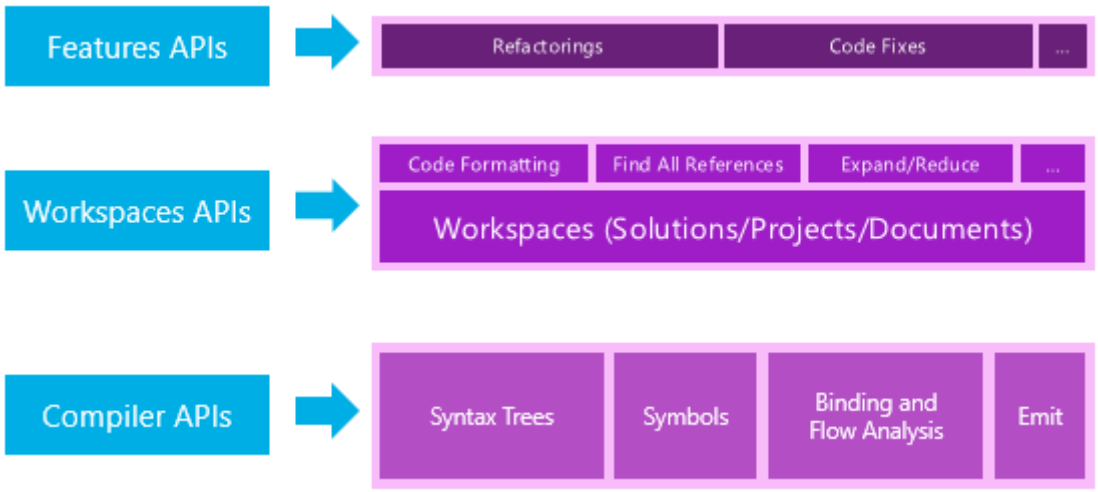
- Vhodně definovaná sada základních typů.
- Automatické čištění dynamicky přidělené paměti (GC – Garbage Collector).
- Podpora automatického generování dokumentace ve formátu XML.
- Označení tříd nebo metod vlastními atributy.
- Plný přístup ke knihovně základních tříd technologie .NET.
- V případě potřeby jsou dostupné ukazatele a přímý přístup do paměti.
- Podpora vlastností a událostí ve stylu jazyka Visual Basic.
- Změnou nastavení překladače můžeme vytvořit spustitelný soubor nebo knihovnu komponent, kterou může volat jiný kód.

Jelikož je programovací jazyk C# dostatečně výkonný, stále se nehodí pro vývoj časově kritického nebo extrémně zátěžového kódu. Pro toto odvětví je vhodné zvolit například C++ [5].

ROSLYN

ROSLYN je nový kompilátor pro technologie .NET Framework, který překládá zdrojový kód do MSIL. Starý kompilátor, naprogramovaný v C++, již bylo složité udržovat pro novější verze jazyka C#. Dále již není kompilátor pouze černá skříňka, jak tomu bylo doposud, ale poskytuje programátorům API. Nový kompilátor je napsaný v jazyce C# pro programovací jazyk C# a kompilátor pro VB.NET je napsán v programovacím jazyce VB.NET. Novinkou také je, že ROSLYN je open source, tudíž každý si může stáhnout zdrojové kódy.

Dalším důvodem, proč byl kompilátor přepsán, byla lepší spolupráci IDE a programátora. Chceme rychleji prohledávat solution, důkladnou statickou analýzu kódu, pokročilé nástroje pro refactoring a další možnosti které programátorovi ušetří, zrychlí a zkvalitní práci [6].



Obr. 4. Struktura ROSLYN [6].

Na obrázku (Obr. 4.) můžeme vidět strukturu kompilátoru ROSLYN. V nejnižší vrstvě se nachází Compiler API, dále API pro práci se solution nezávislou na Visual Studiu a v nejvyšší vrstvě se nachází funkce Visual Studia jako je například refactoring, návrhy oprav, atd. [6].

1.2.5 JavaScript

JavaScript je interpretovaný programovací jazyk, vyvinut v roce 1995 od společnosti Netscape. Postupem času se z něho stal jeden z nejpoužívanějších jazyků vůbec. Má jedinečné postavení mezi programovacími jazyky, jelikož jako jediný je podporován téměř všemi internetovými prohlížeči. Jedná se tedy o skriptovací jazyk pro webové stránky. Přestože vznikl jako jednoduchý nástroj pro validaci formulářů a drobné manipulovaní s obsahem webové stránky, vyvinul se do stavu, kdy můžeme pomocí JS tvořit bohaté klientské aplikace [7].

JavaScript je interpretovaný, objektově orientovaný jazyk a jak již bylo řečeno, je nejlépe známý jako skriptovací jazyk pro webové stránky. Je založený na prototypech, a podporuje více programovacích paradigmat, objektově orientované, imperativní a funkcionální. Základní syntaxe jazyka JavaScript je podobná jako syntaxe jazyka Java nebo C++, aby se snížil počet konceptů pro naučení jazyka JS. Jedná se o dynamicky typovaný jazyk, tudíž programátor nemusí uvádět typy a interpreter si typy spravuje sám. V JavaScriptu existuje celkově 7 typů [8].

- Number.
- String.
- Boolen.
- Null.
- Undefined.
- Object.
- Function.

Každý internetový prohlížeč má v sobě engine, který intepretuje JavaScriptový kód. Google Chome používá V8 engine, Microsoft Edge používá Chakra engine, avšak nejnovější Microsoft Edge je založen na Chromiu, tudíž používá také V8 engine. Mozilla Firefox má Spidermonkey a Safari má JavaScriptCode. Engine V8 používá také Node.js, což je nástroj pro psaní webových serverů v jazyce JavaScript [9].

1.2.6 SQL jazyk

SQL neboli Structured Query Language je standartní programovací jazyk pro správu relační databáze a pro práci s daty. Slouží pro dotazování, vkládání, mazání a úpravě dat. Většina relačních databází jazyk SQL podporuje, což je výhoda pro správce databází, protože je potřeba podporovat databáze na různých platformách [10].

T-SQL

T-SQL neboli Transact SQL je programové rozšíření od firmy Microsoft, které přidává funkce do SQL (Structured Query Language), včetně řízení transakcí, výjimek, zpracování dat a deklaraci proměnných. Všechny aplikace, které komunikují s MS SQL serverem, zasílají na server dotaz, který obsahuje, co se má na MS SQL serveru provést. V dotazu musí být jasně uvedený identifikátor objektu, se kterým dotaz pracuje. Může se jednat o tabulky, uložení procedury nebo také pohledy.

Nejčastějším T-SQL příkazem je uložená procedura. Jedná se o SQL příkaz, který je zkompilovaný a uložený na databázovém serveru.

Dalším příkladem pro T-SQL jsou uživatelsky definované funkce. Jedná se o funkce napsané v jazyce SQL, uložené na databázovém serveru, které můžou přijímat parametry a vracet požadované hodnoty [10].

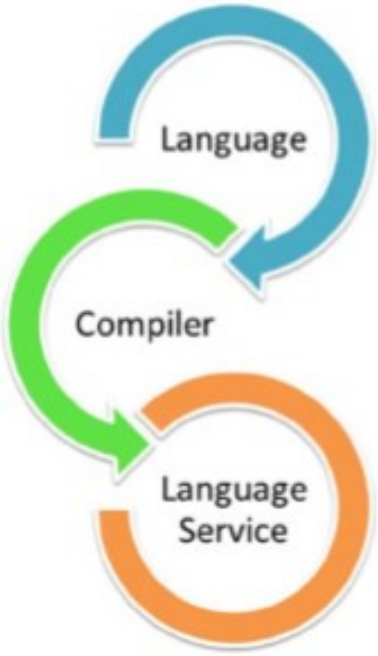
1.2.7 TypeScript

TypeScript (zkráceně TS) je moderní programovací jazyk vytvořený firmou Microsoft a je to open source projekt. Byl vytvořen, aby odstranil problémy JavaScriptu. TypeScript přináší typovou kontrolu, lepší návrhové nástroje, kontrolu kompilace a dynamické načítání modulů.

TypeScript je tedy nadmnožina jazyka JavaScript a TypeScriptový kód je zkompilován do jazyka JavaScript, a tudíž poběží na každém webovém prohlížeči. TypeScript lze rozdělit do tří kategorií na základě jejich vztahu k JS (Obr. 5.). První dvě se týkají verzí jazykové specifikace ECMAScript, což je oficiální specifikace pro jazyk JS. Poslední položka „Additional Features“ přidává funkcionality, které se neplánují stát součásti standardu ECMAScript, jedná se například o generické třídy a typovou kontrolu [11].

Obr. 5. Struktura TypeScriptu [11].

TypeScript se skládá ze tří odlišných, vzájemně se doplňujících částí, které jsou znázorněny na obrázku (Obr. 6.).



Obr. 6. Komponenty

TypeScriptu [11].

První část (Language) obsahuje novou syntaxi jazyka, klíčová slova a anotaci typů. Kompilátor poté převádí zdrojový kód TS do JS. V případě chybně zapsaného zdrojového kódu generuje chybové hlášky. Poslední komponenta poskytuje služby pro vývojářské nástroje, jako například automatické doplňování výrazů nebo možnosti refactoringu, atd. [11].

2 ARCHITEKTURA SOFTWARU

Architektura softwaru je jedna z nejdůležitějších částí při návrhu nového systému. Díky dobře navržené architektuře se dá vytvořit robustní a spolehlivá aplikace. Programátor při návrhu a programování aplikace musí uvažovat o věcech, jako jsou například principy objektově orientovaného programování, třídy, rozhraní, inversion of control, refactoring nebo jednotkové testy. Pokud se programátor bude držet správného návrhu, je schopen psát kvalitní, robustní a znovupoužitelný kód [12].

2.1 SOLID principy

SOLID je soupis pěti principů pro tvorbu software, kterými by se měl programátor řídit. Autor těchto principů je Robert C. Martin.

2.1.1 Single-Responsibility-Princip

Třída by měla mít jen jednu zodpovědnost a dělat jí správně. Neměli by se vytvářet univerzální třídy, které obsahují několik odlišných funkcionalit. Jedna třída se stará například o připojení k databázi, další třída se stará o formátování dat, další třída se stará o komunikaci s emailovým serverem atd. Rozdělení funkcionalit do separátních tříd přidává na přehlednosti. Název třídy by měl vyjadřovat, co třída dělá. Pokud to není splnitelné, pravděpodobně se porušuje tento zákon [13].

2.1.2 Open Closed Principle

Ze všech objektově orientovaných principů je tento nejdůležitější. Říká, že naprogramovaný modul by měl být rozšiřitelný, ale stávající funkcionalita by neměla být měněna. Neměla by nastat situace, kdy programátor neúmyslně změní funkcionalitu jiného modulu. Jak již bylo řečeno, je to nejdůležitější princip, avšak najdou se situace, kdy programátor musí editovat již existující a funkční kód [13].

2.1.3 Liskov Substitution Principle

Tento princip říká, že odvozená třída nesmí nikdy vyžadovat více a poskytovat méně než její bázová třída. Odvozená třída by měla v každém případě nahradit třídu bázovou [13].

2.1.4 Interface Segregation Principle

Hlavní myšlenka tohoto principu je tvorba více specifických rozhraní než vytvoření jednoho univerzálního. Změna v jednom rozhraní, které by popisovalo všechny potřebné metody, by vedla k úpravě všech klientů používajících tohle rozhraní. Další nevýhoda takového rozhraní je jeho nepřehlednost. Je tedy lepší vytvořit více separovaných rozhraní pro konkrétní klienty [13].

2.1.5 Dependency Inversion Principle

Závislost by měla být na abstraktních třídách nebo rozhraních a nikoli na konkrétní implementaci. Je to z toho důvodu, že rozhraní se upravují mnohem častěji než implementace. V případě změny implementace by bylo nutné upravit aplikaci všude, kde by byla závislost na této upravené implementaci [13].

2.2 Dependency Injection

Hlavní myšlenkou DI, neboli vkládání závislostí, je dosažení volného spojení mezi objekty. Tudíž to, aby si objekty nevytvářely instance objektů, které potřebují k běhu, ale aby tyto instance dostávaly zvenčí. Dependency Injection nám také pomáhá při vhodném návrhu architektury softwaru [14].

Existují tři základní metody pro vkládání závislostí. Constructor Injection, Setter Injection a Interface Injection.

2.2.1 Constructor Injection

Jedná se o vkládání závislostí pomocí konstruktoru. Pokud objekt potřebuje ke své činnosti instanci jiného objektu, jednoduše ji injektujeme přes konstruktor. Jedná se, pravděpodobně o nejčastější metodu pro vkládání závislostí [15].

```
public class ClientRepository
{
    private readonly IDbconnection _dbconnection;

    public ClientRepository(IDbconnection dbconnection)
    {
        _dbconnection = dbconnection;
    }
}
```

Obr. 7. Ukázka vkládání závislosti pomocí konstruktoru.

2.2.2 Setter Injection

Další metodou pro vkládání závislostí je Setter Injection. Jedná se o metodu, kdy závislosti není vložená pomocí konstruktoru, ale pomocí setteru [15]. Ukázka viz níže.

```
public class ClientRepository
{
    private IDbconnection _dbconnection;

    public ClientRepository()
    {
        //constructor
    }

    public void SetDbConnection(IDbconnection dbconnection)
    {
        _dbconnection = dbconnection;
    }
}
```

Obr. 8. Ukázka vkládání závislosti pomocí setteru.

2.2.3 Interface Injection

Nejedná se o tak častou metodu vkládání závislostí jako v předchozích dvou případech. Princip této metody je podobný jako u Setter Injection, jen je získání závislosti vázané na rozhraní [15].

```
public interface IInjectDependencies
{
    void InjectDBconnection(IDbconnection dbconnection);
}

public class ClientRepository : IInjectDependencies
{
    private IDbconnection _dbconnection;

    public ClientRepository()
    {
        //constructor
    }

    public void InjectDBconnection(IDbconnection dbconnection)
    {
        _dbconnection = dbconnection;
    }
}
```

Obr. 9. Ukázka vkládání závislosti pomocí interface injection.

3 FRAMEWORK

Framework je skupina knihoven, které dávají ucelený přístup k řešení určitého problému. Ať už se jedná o vývoj backendu, frontendu nebo testování. Důležité je, že vývojář nemusí psát základní prvky stále dokola a tím pádem se vývoj urychlí. Frameworky řeší problémy jako například bezpečnost, komunikaci po síti, autorizaci nebo obsluhu zařízení.

3.1 .NET Framework

.NET Framework je runtime prostředí pro Windows, které spravuje aplikace napsané pro .NET Framework. Skládá se ze dvou hlavních komponent. CLR (Common language runtime), který poskytuje správu paměti, a rozsáhlou knihovnu tříd (.NET Framework class library), která umožňuje programátorům využívat robustní a spolehlivý kód pro všechny možné druhy aplikací [16].

Služby, které .NET Framework poskytuje pro běh aplikací:

- Memory management. Programátor již nemusí řešit přidělování a uvolňování paměti.
- Společný typový systém. V tradičních programovacích jazycích jsou typy definovány překladačem. V .NET Framework jsou základní typy definovány systémem .NET Framework.
- Rozsáhlou knihovnu tříd. Namísto psaní nízko-úrovňového kódu využívají programátoři připravenou rozsáhlou knihovnu tříd.
- Specifické technologie. .NET Framework poskytuje knihovny pro specifické oblasti aplikací. Například ASP.NET pro tvorbu webových aplikací, ADO.NET pro přístup k datům, WCF (Windows Common Foundation) pro servisní aplikace a WPF (Windows Presentation Foundation) pro nativní aplikace pro Windows.
- Jazykovou interoperabilitu. Aplikace napsané pro .NET Framework jsou přeloženy do tzv. CLI (Common Intermediate Language), který je kompilován za běhu runtime modulem.
- Kompatibilitu verzí. Stávající program bude kompatibilní s novější verzí .NET Frameworku bez větších úprav.
- Spouštění aplikací pro specifickou verzi .NET Framework. .NET Framework povoluje mít více verzí na jednom počítači. Aplikace tudíž poběží ve verzi běhového prostředí, pro kterou byla vytvořena [16].

3.2 ASP.NET

ASP.NET je technologie vyvinutá firmou Microsoft, která slouží pro vytváření webových aplikací běžících na straně serveru. Je založena na Microsoft .NET Frameworku, který je seskupením úzce souvisejících technologií, které přináší revoluci při vytváření webových aplikací: od přístupu do databáze až po distribuované aplikace. Technologie ASP.NET je jednou z nejdůležitějších částí .NET Frameworku, která umožňuje vyvíjet plnohodnotné, komplexní a výkonné webové aplikace [17].

Dá se říct, že technologie ASP.NET je nejkomplexnější platformou pro vývoj webu, jaká kdy byla sestavena. ASP.NET bylo vytvořeno, aby nahradilo svého předchůdce ASP, což byla jen sada nástrojů, která sloužila pro vkládání dynamického obsahu na webové stránky [17].

3.2.1 ASP.NET MVC

ASP.NET MVC je Framework, který využívá architektonický vzor MVC pro vytváření webových stránek. Hlavní myšlenkou je oddělení jednotlivých vrstev aplikace, prezenční vrstvy, aplikační a databázové vrstvy. Od roku 2007, kdy byl architektonický vzor představen, se stal jedním z nejpoužívanějších pro tvorbu webových aplikací [18].

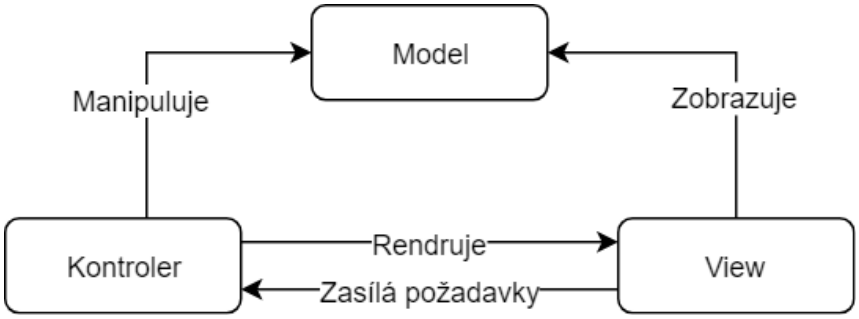
MVC Architektura

Jak již bylo řečeno, tento návrhový vzor odděluje aplikační, logickou a prezentační vrstvu od sebe, avšak jednotlivé vrstvy mezi sebou komunikují. To můžeme pozorovat na obrázku číslo 10.

Model – jedná se o sadu tříd, která popisuje data a metody, které můžou tato data modifikovat a provádět s nimi manipulaci.

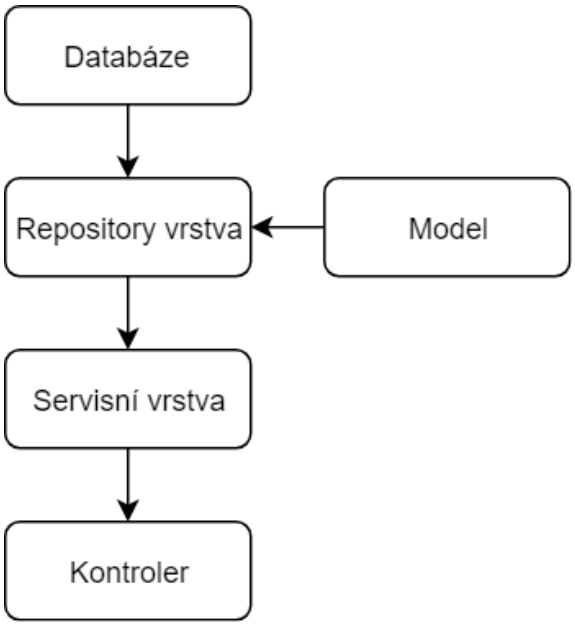
View – definuje, jak se bude zobrazovat uživatelské rozhraní.

Controller – sada tříd a jejich metod, které reagují na požadavky a provádí komunikaci mezi modelem a view v závislosti na konkrétním požadavku [18].



Obr. 10. Schéma MVC architektury.

Mezi hlavní výhody této architektury patří již zmíněné oddělení jednotlivých vrstev a lepší způsob testování aplikace. Další výhoda této architektury je další dělení aplikace do více vrstev. Mezi controllerem a modelem můžou být přidány další vrstvy, jako například servisní vrstva nebo repository vrstva. Servisní vrstva se stará o práci s daty, tím pádem model slouží pouze pro popis dat, ale manipulaci s nimi bude provádět servisní vrstva. Repository vrstva se bude starat pouze o komunikaci s úložištěm. Princip vícevrstvé aplikace je znázorněna na obrázku (Obr. 11.).



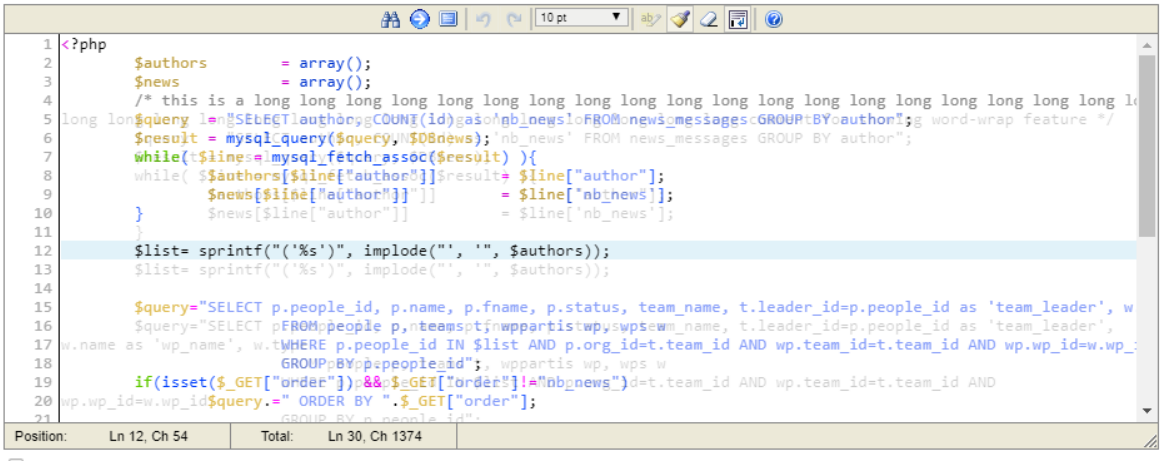
Obr. 11. Ukázka vícevrstvé architektury.

4 ONLINE EDITORY ZDROJOVÝCH KÓDŮ

V dnešní době se snaží webové aplikace nahradit klasické aplikace na osobních počítačích. To také platí pro vývojové nástroje. Existuje mnoho editorů zdrojových kódů, které jsou schopné konkurovat v základních funkcích klasickým IDE.

4.1 EditArea

Jedná se o online editor, který je napsaný v programovacím jazyku JavaScript. EditArea umožňuje formátování, vyhledávání, nahrazování slov ve zdrojovém souboru a zvýrazňování klíčových slov.



Obr. 12. EditArea ve webovém prohlížeči.

Možnosti editoru:

- Snadná integrace do webové stránky.
- Formátování textu včetně zvýraznění syntaxe pro daný jazyk.
- Podpora mnoha programovacích jazyků.
- Hledání a nahrazení slov v editovaném souboru.
- Číslování řádků.
- Možnost více instancí na jedné webové stránce.
- Možnost integrace vlastního rozšíření.
- Dynamická zpráva obsahu.
- Možnost otevřít editor přes celou obrazovku.

Nevýhody:

- Jedná se o starý editor, poslední verze vydána v roce 2010.
- Pomalý chod editoru, pokud je zdrojový obsáhlý.
- Zvýraznění syntaxe pouze pro jeden zvolený jazyk.

```
<html>
<head>
        <title>EditArea Test</title>
<script type="text/javascript" src="/editarea/edit_area/edit_area_full.js">
</script>
        <script language="javascript" type="text/javascript">
        editAreaLoader.init({id : "textarea_1" ,syntax: "css",start_highlight: true});
        </script>
</head>
<body>
<form method="post">
    <textarea id="textarea_1" name="content" cols="80" rows="15">
</textarea>

</form>
</body>
</html>
```

Obr. 13. Implementace editoru EditArea do webové stránky.

4.2 CodeFlask

Jedná se o další editor pro webové stránky napsaný v JavaScriptu. Tento editor je určený pro jednoduché úpravy, jelikož nenabízí tolik možností. Má pouze jedno schéma a formátování pro malou část jazyků (JS, CSS, HTML, XML a jazyky z rodiny C jazyků). Pro podporu více jazyků je nutné implementovat knihovnu Prism.js.

```
1  new Array(5)
2    .fill('Option ')
3    .map((e, i) => e + (10 + i).toString(36)
4    .toUpperCase()
5    )
6    .join('\n')
7
8
```

Obr. 14. Ukázka editoru CodeFlask.

Výhoda editoru je jednoduchá integrace do webové aplikace. Ukázka zdrojového kódu je níže.

```
<html>
<head>
    <script language="javascript" type="text/javascript">
        import CodeFlask from 'codeflask';
        const flask = new CodeFlask('codeFlask', {
            language: 'js'
        });

    </script>
</head>
<body>
<form method="post">
    <textarea id="codeFlask" name="content" >
</textarea>

</form>
</body>
</html>
```

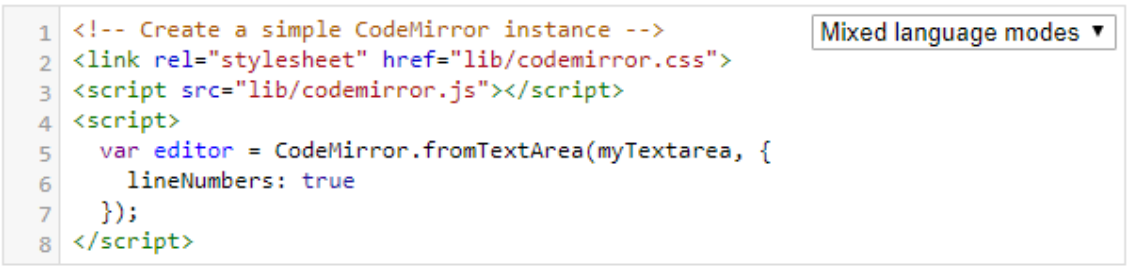
Obr. 15. Implementace editoru CodeFlask do webové stránky.

4.3 CodeMirror

CodeMirror je další editor, napsaný v JavaScriptu, pro webové aplikace. Je to robustní editor, který přichází s podporou více než 100 programovacích jazyků, bohatým API a různými schématy. Také můžou být přidávány rozšíření pro další funkcionality.

Možnosti editoru:

- Podpora více než 100 jazyků.
- Automatické doplňování.
- Formátování kódu.
- Konfigurovatelné klávesové zkratky.
- Párování závorek.
- Podpora zvýraznění syntaxe pro více jazyků v jednom okně editoru.
- Nastavitelné schéma.
- Rozsáhlé API.



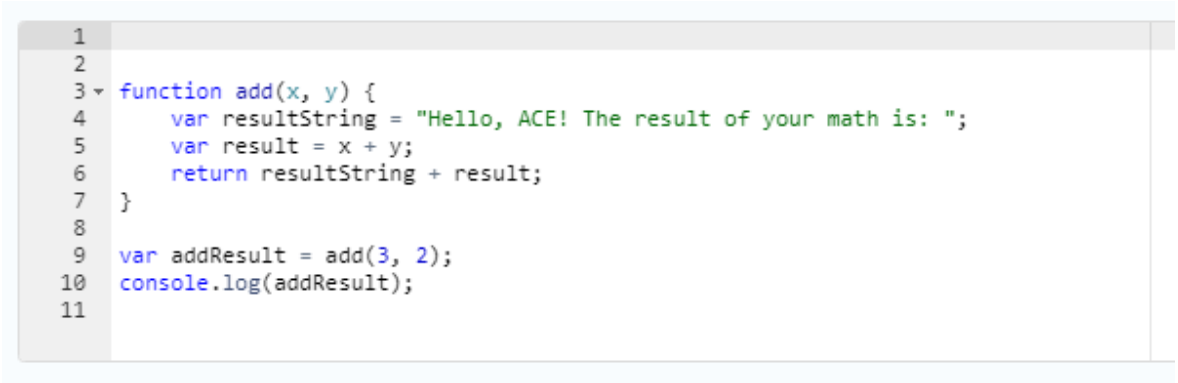
Obr. 16. Ukázka editoru Code Mirror.

4.4 ACE Editor

ACE Editor je další výkonný editor pro webové aplikace. Stejně jako předchozí, je napsán v programovacím jazyce JavaScript. Editor je udržován jako primární editor pro Cloud9 IDE a je nástupcem projektu Mozilla Skywriter.

Možnosti editoru:

- Podpora více než 110 programovacích jazyků.
- Velké množství schémat.
- Automatické formátování.
- Možnost zpracovávat rozsáhlé zdrojové kódy.
- Konfigurovatelné klávesové zkratky.
- Zvýrazňování párových závorek.
- Přetažení textu myší.
- Zobrazení skrytých znaků.
- Automatické doplňování a napovídání.



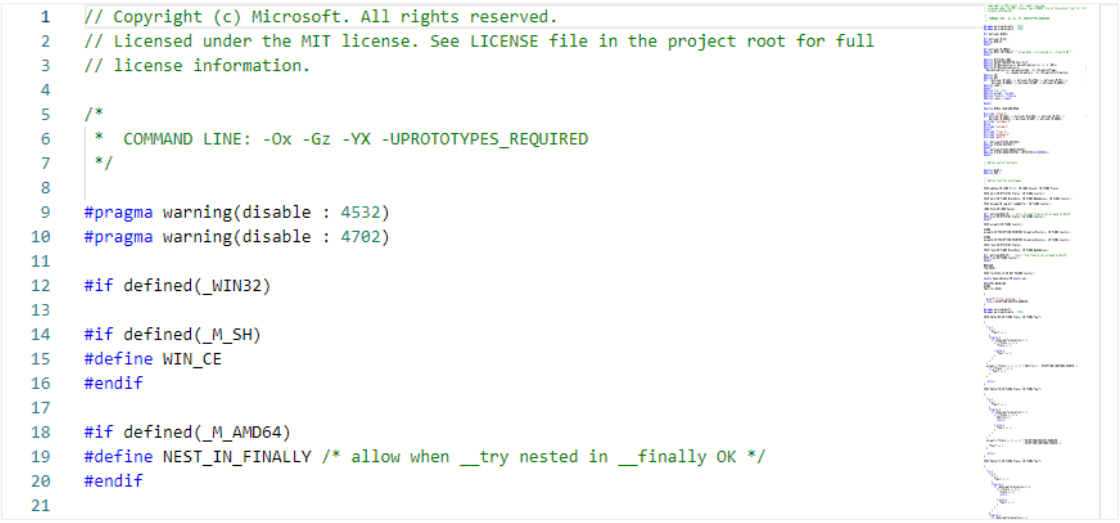
Obr. 17. Ukázka editoru ACE Editor.

4.5 Monaco Editor

Monaco Editor je robustní editor napsaný v TypeScriptu od firmy Microsoft. Jedná se o editor, na kterém je postaven desktopový program Microsoft Visual Code.

Možnosti editoru:

- Podpora více než 50 jazyků.
- Rozsáhlé API editoru.
- IntelliSence pro JavaScript, HTML, XML, TypeScript, LESS, JSON a CSS.
- Obrovské množství nastavení.
- Porovnávání dvou zdrojových kódů.
- Složitější implementace do webové stránky.



Obr. 18. Ukázka Monaco Editoru.

II. PRAKTICKÁ ČÁST

5 ÚVOD DO PRAKTICKÉ ČÁSTI

Tato část diplomové práce se zabývá kompletním návrhem a celkovou implementací webové aplikace pro automatizované testování programovacích jazyků (C#, JavaScript a MS-SQL). Nejdříve je popsána funkcionalita webové aplikace a také designový návrh. Následně je popsána architektura softwaru a použité technologie a nástroje. Závěrečná část obsahuje detailní popis zdrojových kódu aplikace, ověření funkčnosti a výsledný design aplikace.

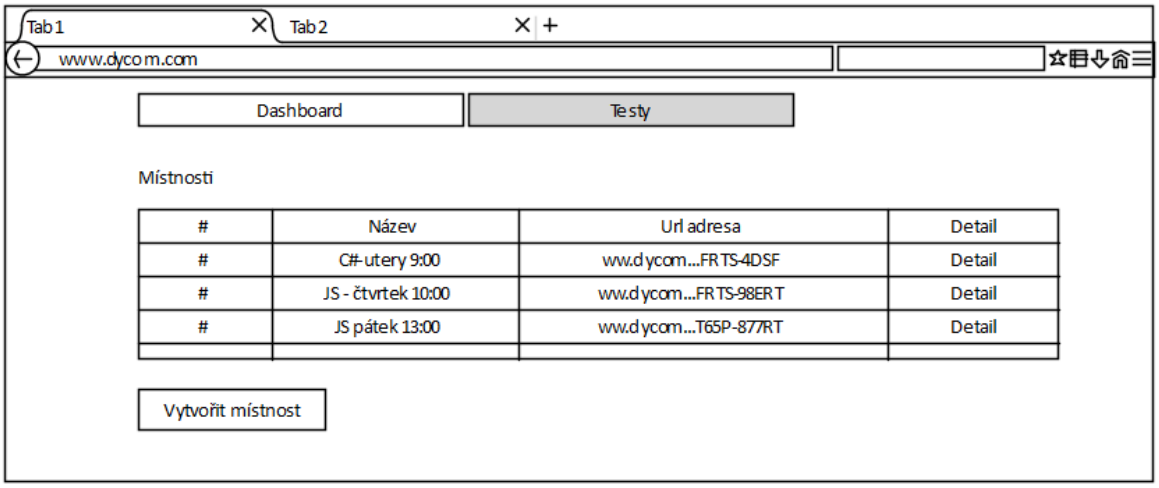
6 NÁVRH APLIKACE

Hlavní účel této vytvořené aplikace bude testování programovacích jazyků C#, JavaScript a MS-SQL.

Bude se jednat o webovou aplikaci, která bude vyhodnocovat správnost úloh, které uživatel vyřešil. Uživateli bude zaslána URL adresa testovací místnosti, do které se zaregistruje se svým jménem, příjmením a datem narození. Po registraci se zobrazí webová stránka se zadáním úlohy, kterou je potřeba vyřešit. Mezi jednotlivými úlohami lze přepínat. Pomocí tlačítka na webové stránce si může uživatel ověřit správnost řešení. Uživatel je časově omezen pro vyřešení úkolů a to v závislosti na nastavení testovací místnosti. Po vypršení časového intervalu již není možné úkoly upravovat a dojde k přesměrování na stránku, kde je zobrazena procentuální úspěšnost pro každý úkol.

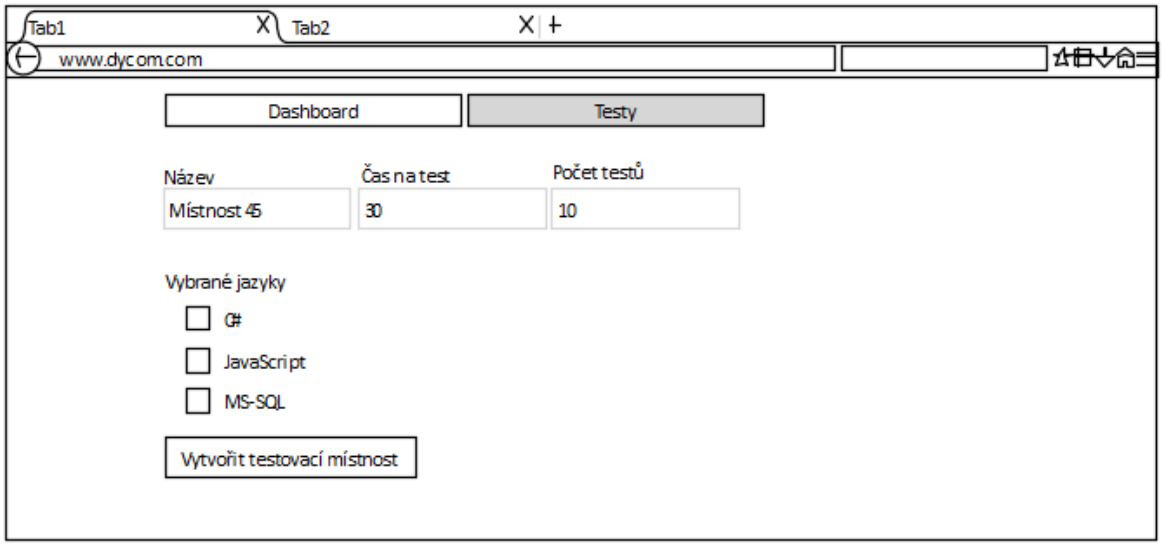
Administrátorská část webové aplikace bude obsahovat seznam testovacích místností, do kterých se můžou uživatelé registrovat. Dále bude implementována funkcionalita pro přidávání a úpravu jednotlivých testů a jejich testovacích scénářů. Administrátor také bude mít přehled o uživatelích v konkrétní místnosti a jejich procentuální úspěšnosti z jednotlivých testů.

6.1 Design aplikace

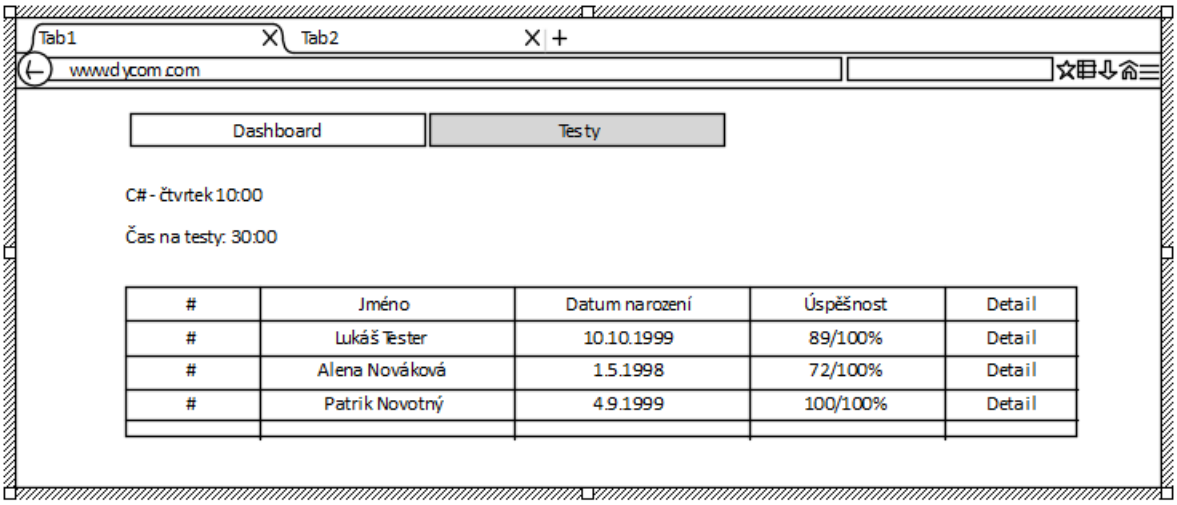


Obr. 19. Seznam místností v administrátorském prostředí.

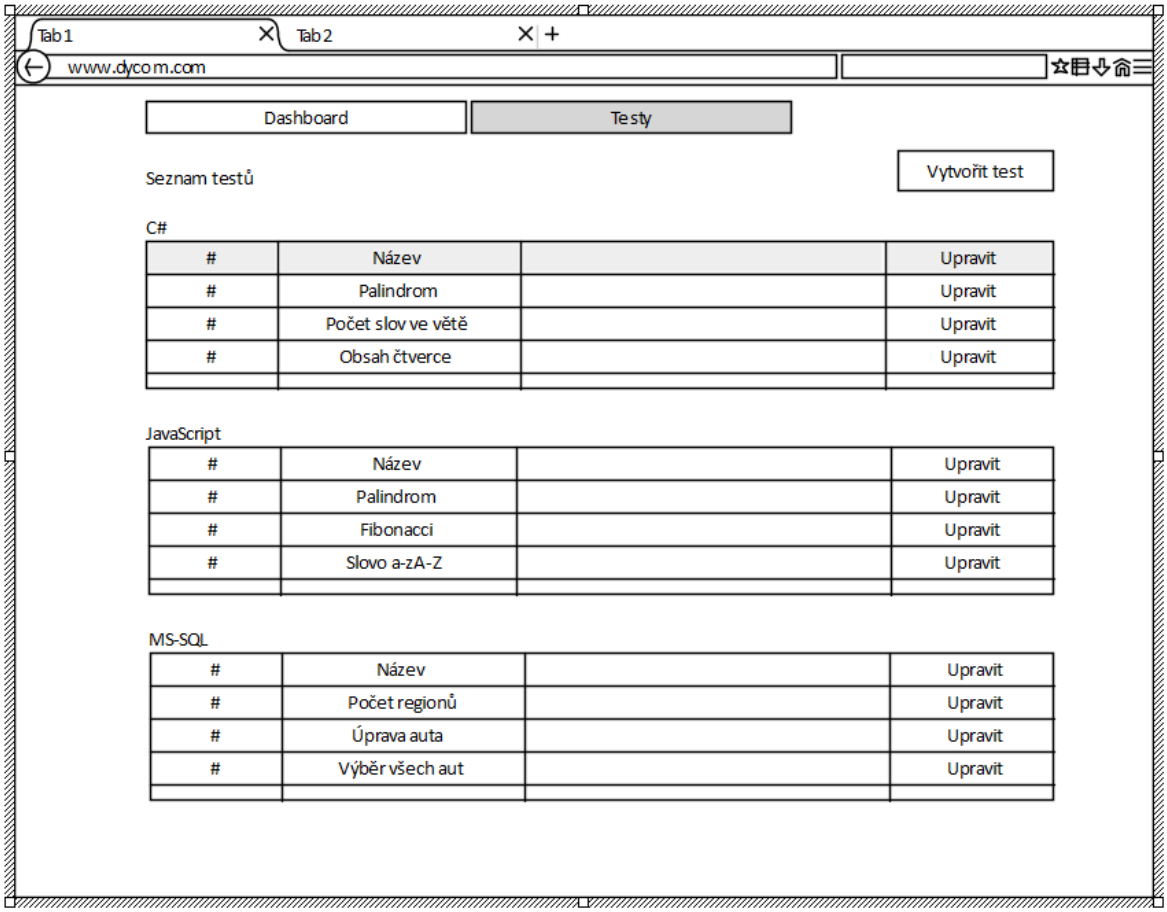
V sekci *Dashboard* jsou zobrazeny všechny testovací místnosti. Administrátor zde může přidat novou místnost, nebo si zobrazit detail již vytvořené místnosti, kde je zobrazen seznam uživatelů, kteří se registrovali do místnosti.



Obr. 20. Formulář pro vytvoření testovací místnosti.



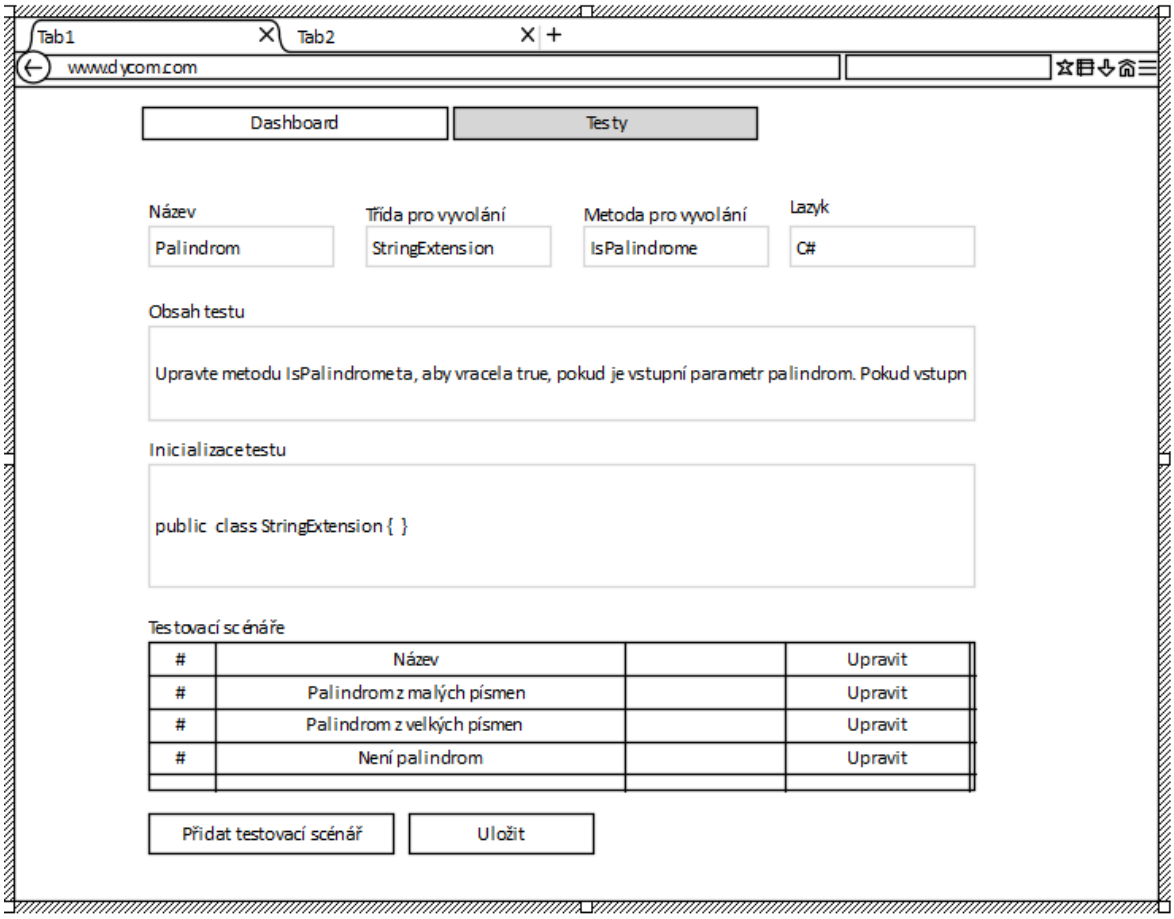
Obr. 21. Seznam uživatelů v testovací místnosti.



Obr. 22. Stránka se seznamem testů.

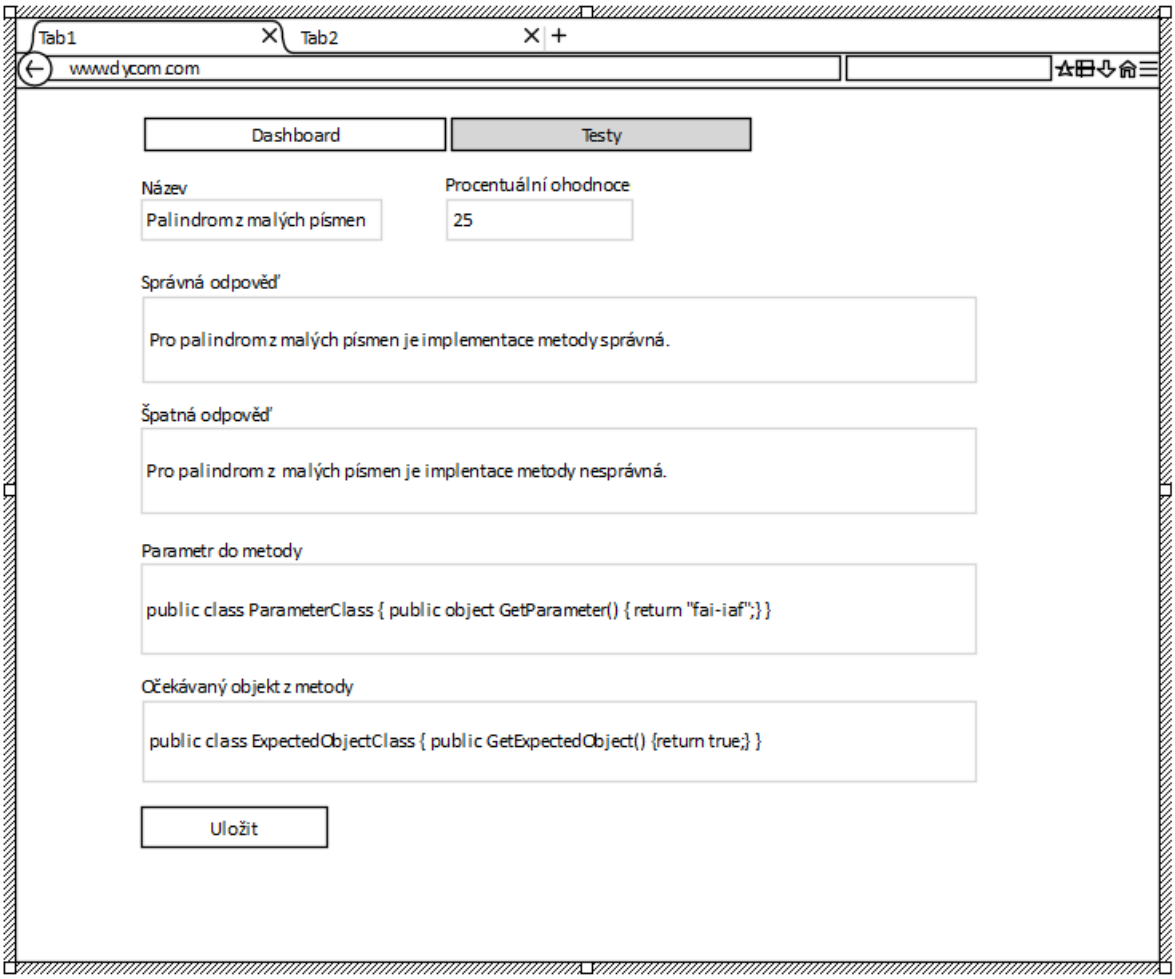
V sekci *Testy* vidí administrátor všechny testy, které jsou vytvořeny. Může zde vytvořit nový test, nebo zobrazit detail existujícího testu a popřípadě ho editovat.

V detailu testu je základní nastavení testu. Je zde nutné nastavit název testu, název třídy a metody pro vyvolání, druh programovacího jazyka, obsah testu a také základní program testu, který bude muset uživatel upravit.



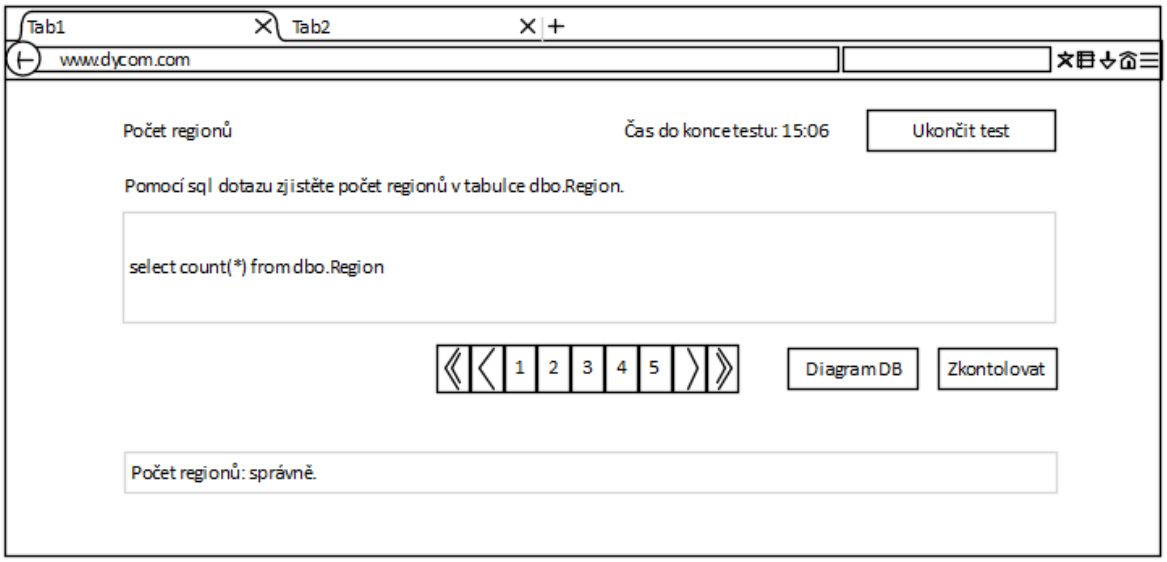
Obr. 23. Stránka pro správu testu.

Stránka pro správu testovacího scénáře obsahuje formulář, kde je potřeba vyplnit název, procentuální ohodnocení, text správné odpovědi a text špatné odpovědi. Další povinný údaj je kód vracející parametr, který bude vstupem do testované metody, a také kód vracející očekávanou hodnotu z metody.



Obr. 24. Stránka pro správu testovacího scénáře.

Po registraci uživatele do testovací místnosti se zobrazí zadání prvního úkolu. Po vyřešení úlohy může uživatel provést kontrolu správnosti. Pod formulářem se mu vypíše správná, nebo špatná odpověď. Pokud se jedná o úkol z SQL jazyka, může si uživatel pomocí tlačítka „Diagram DB“ zobrazit diagram relační databáze, pro kterou jsou testy psané.



Obr. 25. Testovací prostředí.

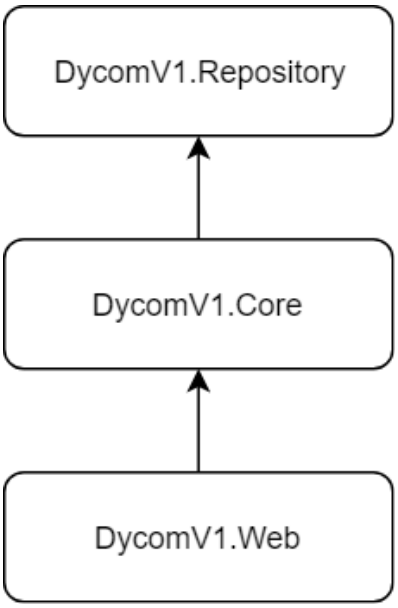
7 ARCHITEKTURA

7.1 Webová aplikace

Webová aplikace bude napsána v programovacím jazyce C#, konkrétně nad .NET Frameworkem verze 4.6.1. Dále bude použit návrhový vzor MVC, a to konkrétně ASP.NET MVC verze 5.

7.1.1 Architektura webové aplikace

Aplikace bude obsahovat celkově 3 projekty a to *DycomV1.Web*, *DycomV1.Core* a *DycomV1.Repository*. Tento návrh architektury nám umožní snadné budoucí rozšíření celkového systému nebo nahrazení konkrétní vrstvy za jinou technologii. Je zde dobrá možnost testování aplikace, jelikož se všechna logika nachází v projektu *DycomV1.Core*.



Obr. 26. Schéma architektury aplikace.

DycomV1.Repository

Jedná se o projekty, který slouží výhradně pro komunikaci s aplikační databází. Obsahuje jednotlivé třídy pro vkládání, výběr, mazání a editování konkrétních entit v aplikační databázi. Projekt obsahuje jednotlivé modely, se kterými aplikace pracuje.

DycomV1.Core

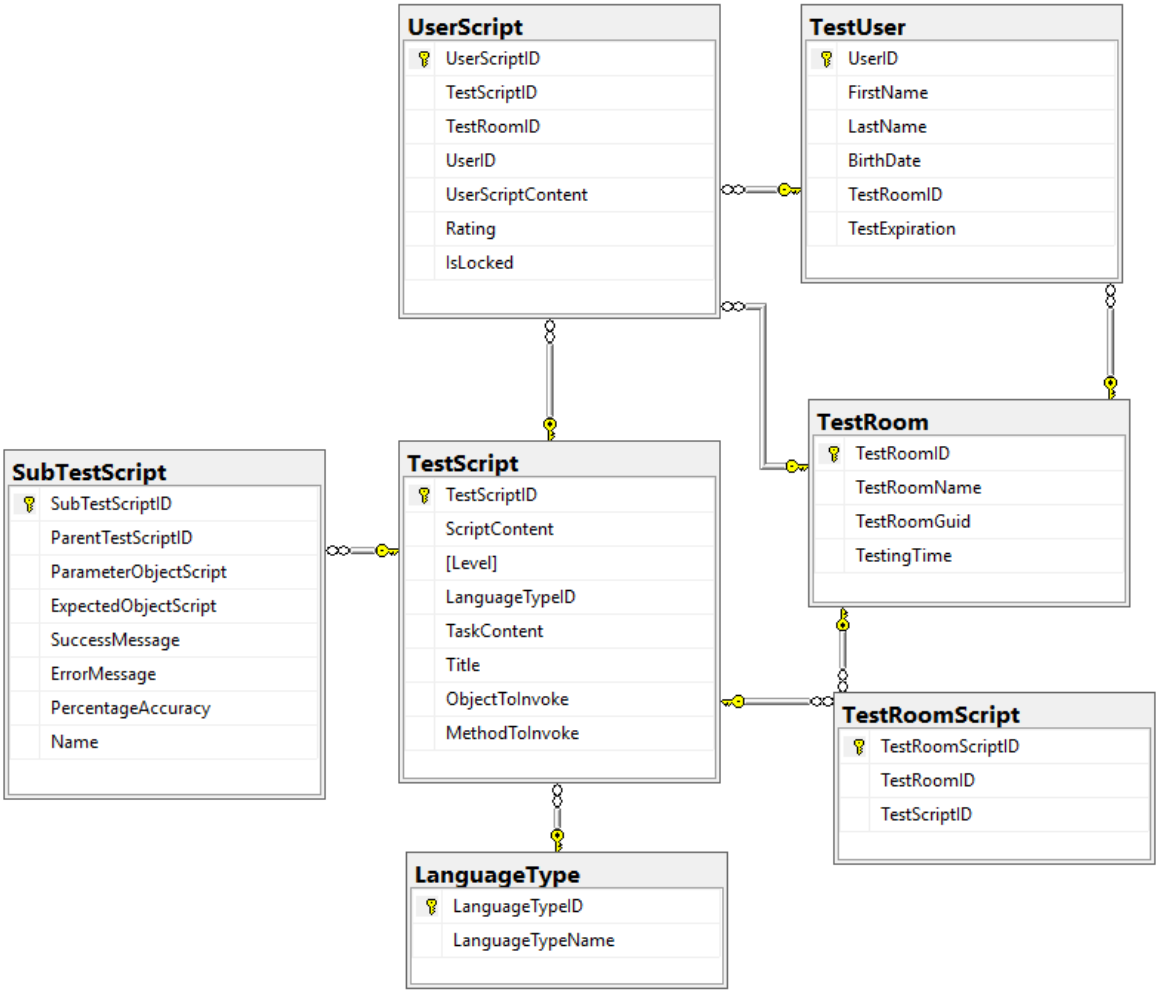
Jedná se o vrstvu, která spolupracuje s vyšší vrstvou aplikace (*DycomV1.Web*) a nižší vrstvou aplikace (*DycomV1.Repository*). Její hlavní úkol je zprostředkovávat data pro vyšší a nižší vrstvu aplikace. V téhle vrstvě se nachází celá logika aplikace.

DycomV1.Web

Tento projekt je samostatná webová aplikace, která komunikuje s nižší aplikační vrstvou (*DycomV1.Core*). Nižší vrstva tedy zprostředkovává služby jako například získání dat, úpravu a přípravu potřebných dat a tak podobně.

7.2 Aplikační databáze

Databázový systém je postaven na MS SQL serveru. Jedná se o databázi *DycomProgram*. Ukládají se zde uživatelé, testovací úkoly a vyřešené úkoly od uživatelů.



Obr. 27. Diagram relační databáze.

7.2.1 Tabulky

TestUser – tabulka obsahuje testované uživatele.

TestRoom – tabulka obsahuje základní informace o místnosti, do které se můžou uživatelé přihlásit.

TestScript – jedná se o základní nastavení pro testovací úkol.

TestRoomScript – vazební tabulka mezi *TestRoom* a *TestScript*, která určuje, jaké testovací úlohy obsahuje testovací místnost.

SubTestScript – tabulka, která obsahuje jednotlivé testovací scénáře pro úkoly.

LanguageType – tabulka s výčtem programovacích jazyků.

UserScript – úlohy, které uživatel vyřešil.

7.2.2 Procedury

Pro práci s databází jsou použity uložené procedury, což jsou uložené skripty na databázovém serveru, které se následně volají z aplikace.

dbo.ProcGetUsersForRoom – vrací všechny uživatele přihlášené do konkrétní místnosti.

dbo.ProcLaguageTypeGetAll – vrací všechny testované programovací jazyky.

dbo.ProcSubTestByID – vrací testovací scénář podle ID.

dbo.ProcSubTestGetAllByParentID – vrací všechny testovací scénáře konkrétní test.

dbo.ProcSubTestInsert – vloží nový testovací scénář do databáze.

dbo.ProcSubTestUpdate – edituje testovací scénář.

dbo.ProcTestDelete – smaže test včetně jeho testovacích scénářů.

dbo.ProcTestGetByID – vrátí test podle ID.

dbo.ProcTestInsert – uloží nový test do databáze.

dbo.ProcTestRoomByID – vrátí testovací místnost podle ID.

dbo.ProcTestRoomGetAll – vrátí všechny testovací místnosti.

dbo.ProcTestRoomGetByGuid – vrátí testovací místnost podle unikátního identifikátoru.

dbo.ProcTestRoomInsert – vloží novou testovací místnost do databáze.

dbo.ProcTestRoomScriptInsert – vloží data do vazební tabulky.

dbo.ProcTestScriptGetAll – vrátí všechny testy.

dbo.ProcTestScriptGetAllForUser – vrátí všechny testy pro konkrétního uživatele.

dbo.ProcTestScriptUpdate – upraví test.

dbo.ProcTestUserByID – vrátí uživatele podle ID.

dbo.ProcTestUserInsert – vloží nového uživatele do databáze.

dbo.ProcUserScriptUpdateRating – upraví hodnocení testu pro uživatele.

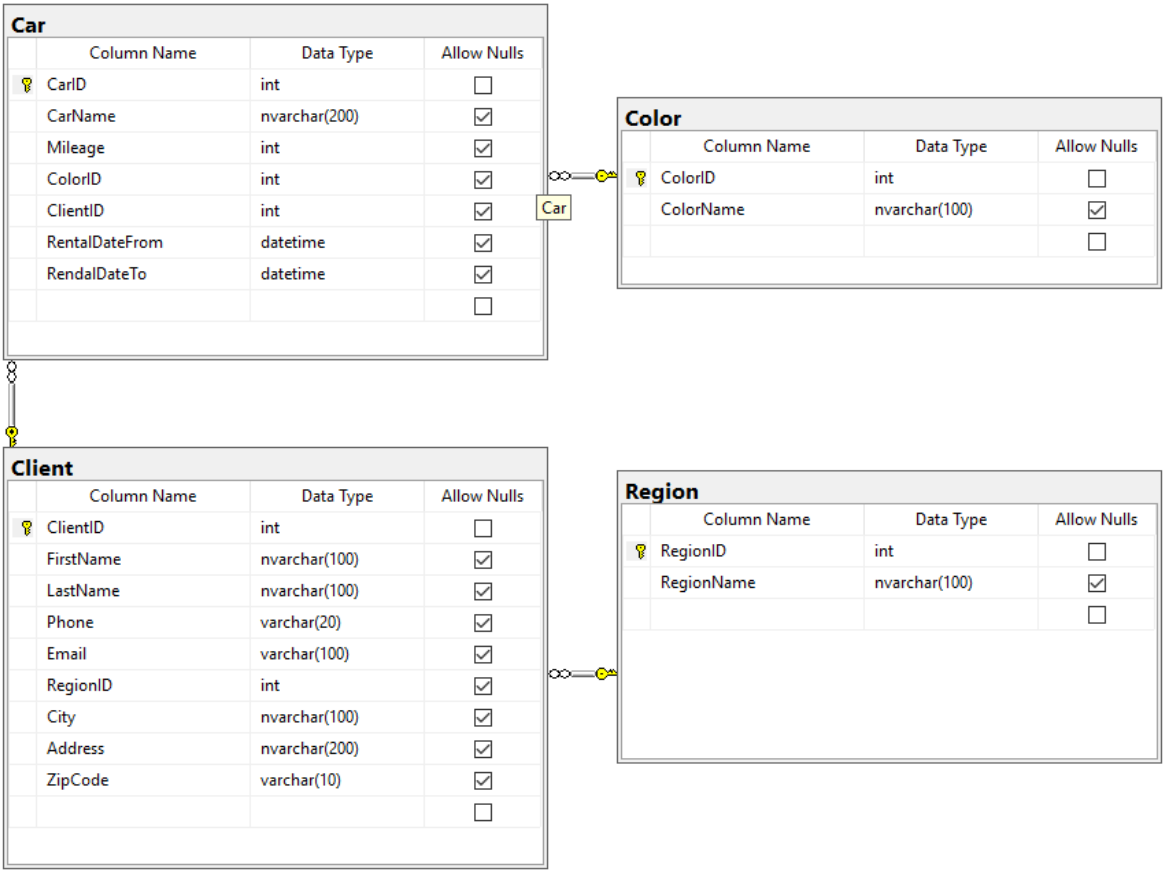
dbo.ProcUserScriptCreate – vygeneruje testy pro uživatele.

```
CREATE PROCEDURE [dbo].[ProcTestScriptGetForUser]
(
    @UserID int,
    @TestRoomID int
)
AS
BEGIN

    SET NOCOUNT ON;
    Select
    US.UserScriptID,
    US.TestScriptID,
    US.TestRoomID,
    US.UserID,
    US.UserScriptContent as ScriptContent,
    US.Rating,
    TS.TaskContent,
    TS.Level,
    TS.LanguageTypeID,
    TS.Title,
    US.IsLocked
    FROM UserScript US
    LEFT JOIN TestScript TS on TS.TestScriptID = US.TestScriptID
    WHERE UserID = @UserID AND TestRoomID = @TestRoomID
    ORDER BY TestScriptID
END
```

Obr. 28. Ukázka procedury pro získání testů pro uživatele.

7.3 Testovací databáze

Aplikace komunikuje s testovací databází, což je separátní databáze, pro testování jazyka SQL. Jedná se o databázi *DycomTesting*, která je jednoduchá, avšak není problém s rozšířením databáze a vytvořením složitějších testovacích úloh.



Obr. 29. Diagram relační databáze pro testování uživatelů.

8 IMPLEMENTACE

Tato kapitola popisuje podrobně jednotlivé projekty, použité technologie a nástroje pro vývoj celé webové aplikace.

8.1 DycomV1.Repository

Jedná se o nejnižší vrstvu aplikace. Jak již bylo řešeno, komunikuje s aplikační databází a také s databází testovací, kde jsou testovací data pro testování uživatelů. Dále projekt obsahuje modely, se kterými celá aplikace pracuje. Pro usnadnění práce pro komunikaci s databází, je zde využit NuGet balíček Dapper, což je jednoduchý objektový mapper pro .NET Framework, který je zodpovědný za mapování entit z databáze na modelové třídy a opačně.

```
public interface IDatabaseProvider
{
    IDbConnection GetConnection();
    IDbConnection GetTestingConnection();
}

public class DatabaseProvider : IDatabaseProvider
{
    public IDbConnection GetConnection()
    {
        return new SqlConnection(ConfigurationManager.
        ConnectionStrings["DycomProgramDB"].ConnectionString)
    }

    public IDbConnection GetTestingConnection()
    {
        return new SqlConnection(ConfigurationManager.
        ConnectionStrings["DycomTestingDB"].ConnectionString);
    }
}
```

Obr. 30. Ukázka implementace třídy, která vrací připojení do databáze.

Bylo vytvořeno rozhraní *IDatabaseProvider*, které má dvě metody. Následně toto rozhraní implementuje třída *DatabaseProvider*, která nám zprostředkovává připojení do aplikační databáze a do testovací databáze. Třídu *DatabaseProvider* následně využívají repository třídy, které formulují dotazy do databází.

```
public class UserRepository : IUserRepository
{
    private readonly IDatabaseProvider _dbProvider;

    public UserRepository(IDatabaseProvider databaseProvider)
    {
        _dbProvider = databaseProvider;
    }

    public TestUser GetByID(int userID)
    {
        using (var db = _dbProvider.GetConnection())
        {
            return db.QueryFirst<TestUser>("dbo.ProcTestUserByID",
                new { UserID = userID }, commandType: CommandType.StoredProcedure);
        }
    }

    public IList<TestUser> GetUsersForRoom(int roomID)
    {
        using (var db = _dbProvider.GetConnection())
        {
            return db.Query<TestUser>("dbo.ProcGetUsersForRoom",
                new { RoomID = roomID }, commandType: CommandType.StoredProcedure).ToList();
        }
    }

    public int Insert(TestUser user)
    {
        using (var db = _dbProvider.GetConnection())
        {
            return db.QueryFirst<int>("dbo.ProcTestUserInsert", new
            {
                FirstName = user.FirstName,
                LastName = user.LastName,
                BirthDate = user.BirthDate,
                TestExpiration = user.TestExpiration,
                TestRoomID = user.TestRoomID
            }, commandType: CommandType.StoredProcedure);
        }
    }
}
```

Obr. 31. Implementace třídy *UserRepository*.

8.2 DycomV1.Core

V tomhle projektu se ukrývá veškerá logika celé aplikace. Každá funkcionalita je umístěna do separátní třídy, aby bylo dodrženo SOLID principům.

8.2.1 Struktura DycomV1.Core

CompilationService

Obsahuje servisní třídy pro dynamickou kompilaci jazyka C#, interpretaci jazyka JavaScript a spouštění skriptů jazyka MS-SQL.

Constants

Obsahuje konstanty, se kterými aplikace pracuje.

Extension

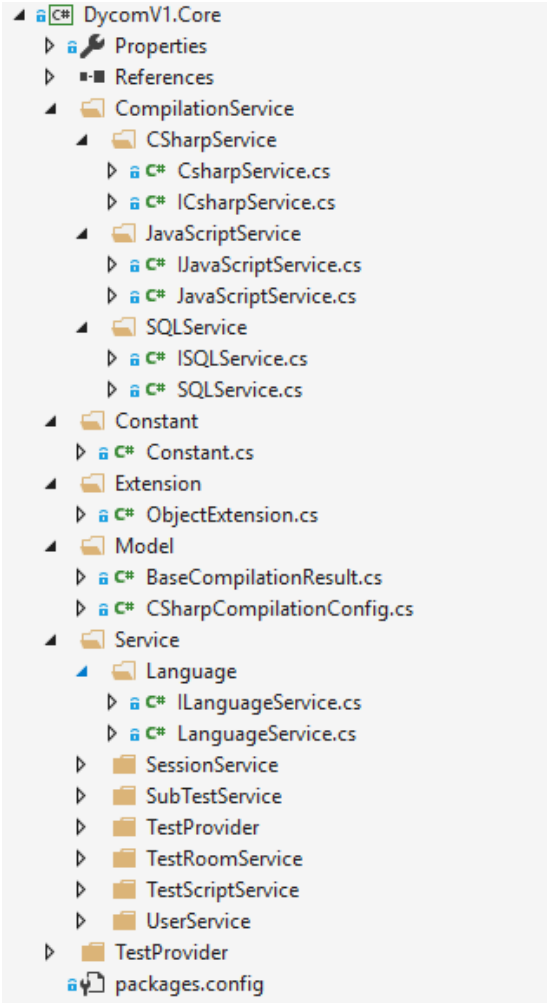
Zde se zachází třída *ObjectExtension*, která nám umožňuje volat rozšiřující metody nad objekty.

Model

Složka *Model* obsahuje rozšiřující modelové třídy, se kterými aplikace pracuje.

Service

Ve složce *Service* jsou jednotlivé servisní třídy, které pracují s konkrétní entitou. Ať už se jedná o testovací místnost, uživatele nebo o test.



Obr. 32. Struktura projektu *DycomV1.Core*.

```
public interface ILanguageService
{
    IList<LanguageType> GetAllLanguageTypes();
}
```

Obr. 33. Ukázka rozhraní *ILanguageService*.

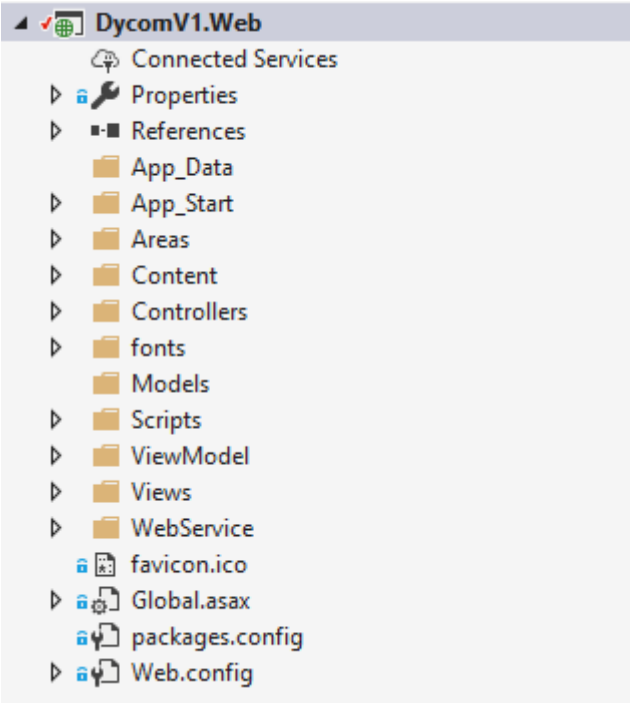
```
public class LanguageService : ILanguageService
{
    private readonly ILanguageRepository _languageRepository;

    public LanguageService(ILanguageRepository languageRepository)
    {
        _languageRepository = languageRepository;
    }

    public IList<LanguageType> GetAllLanguageTypes()
    {
        return _languageRepository.GetAllLanguageTypes();
    }
}
```

Obr. 34. Ukázka implementace rozhraní *ILanguageService*

ve třídě *LanguageService*.

8.3 DycomV1.Web

Jedná se webový projekt, založený na ASP.NET MVC5. Jeho hlavní účel je zpracovávat požadavky od uživatele. Projekt obsahuje konfiguraci projektu, controllery, viewmodely a views. Do projektu je zavedeno dependency injecton pomocí NuGet balíčku Autofac.



Obr. 35. Struktura projektu

DycomV1.Web.

8.3.1 Dependency Injection

Pro DI byl použit NuGet balíček Autofac. Jedná se o Inversion Of Control kontejner pro aplikace napsané v .NET Framework. Autofac zajišťuje správu instancí v aplikaci, a tudíž umožňuje rychlejší úpravu aplikace, popřípadě její rozšíření. Konfigurace kontejneru se provádí v *Global.asax* v metodě *Application_Start().*

Další výhoda kontejneru je, že umožňuje nastavení životnosti instancí, které si ostatní třídy žádají.

Instance Per Dependency – pro každý požadavek o instanci objektu, bude vrácená jedinečná instance třídy.

Single Instance – pro každý požadavek o instanci objektu je vrácena vždy jedna a ta samá instance třídy.

Instance Per Lifetime – pro každý požadavek je vrácena unikátní instance, ale tato instance je použita i pro vnořené závislosti.

Instance Per Request – pro každý požadavek na server je vrácena unikátní instance třídy.

Nastavení kontejneru

```
var builder = new ContainerBuilder();
builder.RegisterType<DatabaseProvider>().As<IDatabaseProvider>().InstancePerDependency();
builder.RegisterType<UserRepository>().As<IUserRepository>().InstancePerDependency();
builder.RegisterType<UserService>().As<IUserService>().InstancePerDependency();
builder.RegisterControllers(typeof(MvcApplication).Assembly);
var container = builder.Build();
DependencyResolver.SetResolver(new AutofacDependencyResolver(container));
```

Obr. 36. Ukázka registrace komponent do kontejneru.

Jako první se vytvoří instance třídy *ContainerBuilder*. Následně se registrují používané komponenty pomocí generické metody *RegisterType*. Poté se registrují controllery, pomocí metody *RegisterControllers*. Kontejner se sestaví a použije se jako parametr do konstruktoru *AutofacDependencyResolver*. *DependencyResolver* se následně stará o vytváření instancí na požádání.

Ukázka injektování závislosti

```
public class DashboardController : Controller
{
    private readonly ITestRoomService _testRoomService;

    public DashboardController(ITestRoomService testRoomService)
    {
        _testRoomService = testRoomService;
    }

    public ActionResult Index()
    {
        var viewModel = new DashboardViewModel()
        {
            TestRooms = _testRoomService.GetAll().ToList()
        };
        return View(viewModel);
    }
}
```

Obr. 37. Ukázka injektování *ITestRoomService* do třídy *DashboardController*.

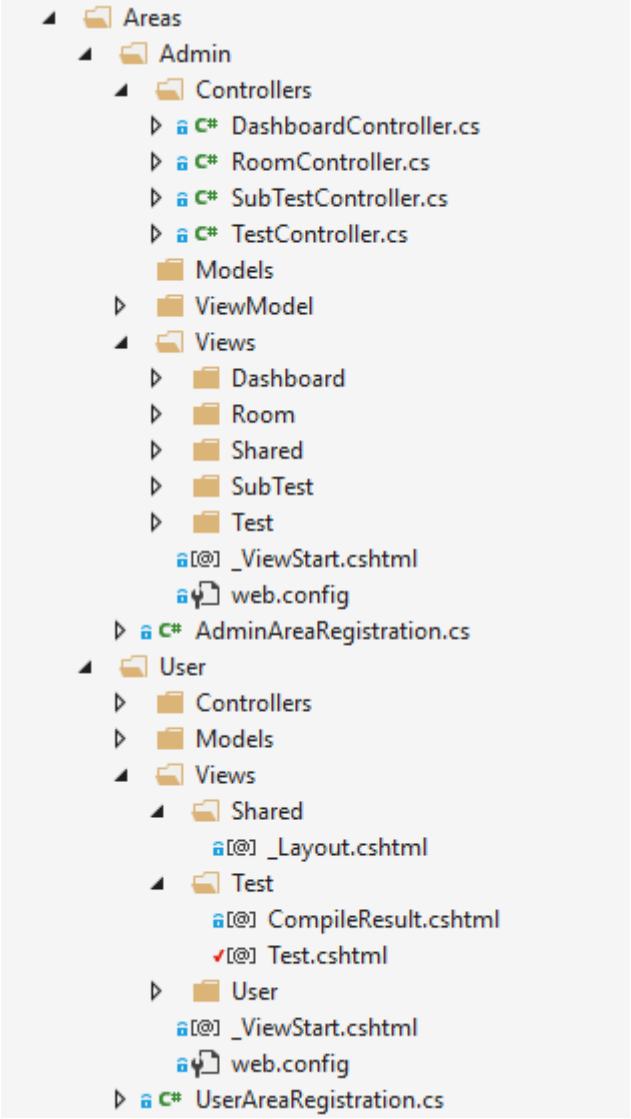
Při vytváření instance *DashboardController* vytvoří *DependencyResolver* instanci objektu, který implementuje rozhraní *ITestRoomService* a následně je tato instance použita v konstruktoru třídy *DashboardController* a přiřazena do privátní proměnné. Poté můžeme využívat privátní proměnou v celé třídě, bez toho aniž bychom se starali o vytváření požadovaného objektu.

8.3.2 Areas

Pro přehlednější vývoj byl projekt rozdělen na dvě části (*Areas*). Jedná se o dvě rozdílné sekce, kde jedna část je pro administrátora aplikace a druhá část je pro uživatele, který je testován. Každá area obsahuje controllery, viewmodely a views. Nastavení routování pro konkrétní areas se nachází ve třídě *AdminAreaRegistration* a *UserAreaRegistration*.

```
public class AdminAreaRegistration : AreaRegistration
{
    public override string AreaName
    {
        get
        {
            return "Admin";
        }
    }

    public override void RegisterArea(AreaRegistrationContext context)
    {
        context.MapRoute(
            "Admin_default",
            "Admin/{controller}/{action}/{id}",
            new { action = "Index", id = UrlParameter.Optional }
        );
    }
}
```

Obr. 38. Ukázka nastavení routování pro Admin area.



Obr. 39. Struktura *Areas*.

Admin area

Obsahuje funkcionalitu pro administrátora aplikace. Administrátor aplikace může vytvářet testovací místnosti pomocí *TestRoomController*, přidávat a upravovat nastavení testů pomocí *TestController* a také přidávat a nastavovat testovací scénáře pomocí *SubTestController*.

DashboardController

Metoda *Index* vrací seznam testovacích místností.

RoomController

Metoda *Detail* vrací seznam uživatelů, kteří byli testováni ve zvolené místnosti.

Metoda *CreateRoom* vytváří novou testovací místnost.

TestController

Metoda *Index* vrací seznam všech testů, rozdělených podle programovacího jazyka.

Metoda *Test* vrátí detail na zvolený test.

Metoda *SaveTest* uloží nový test, popřípadě upraví stávající.

SubTestController

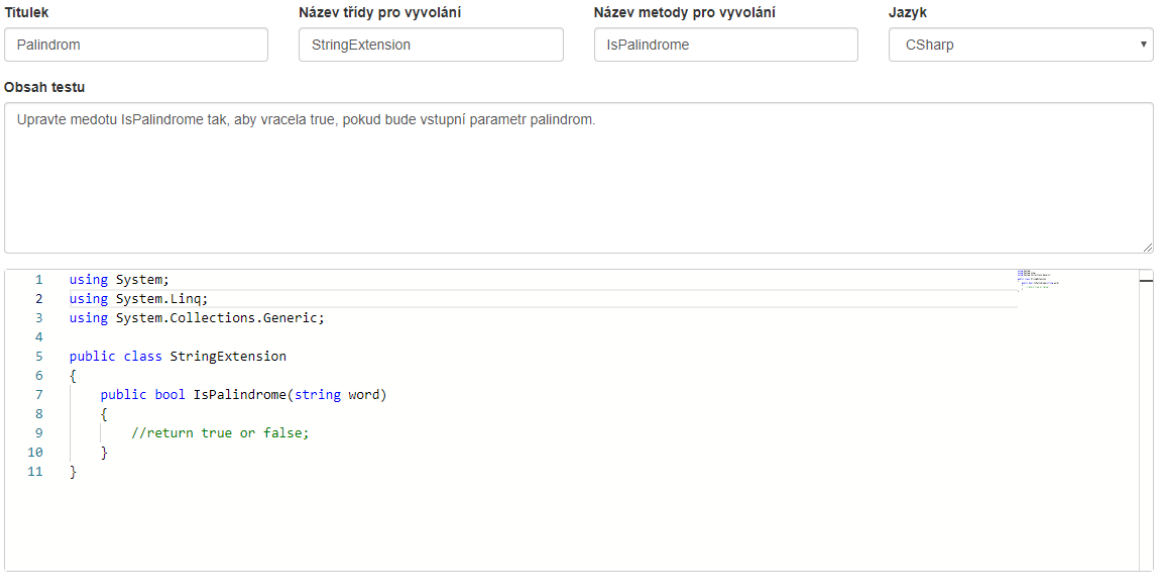
Metoda *Detail* vrátí detail testovacího scénáře.

Metoda *CreateSubTest* vytvoří nový testovací scénář.

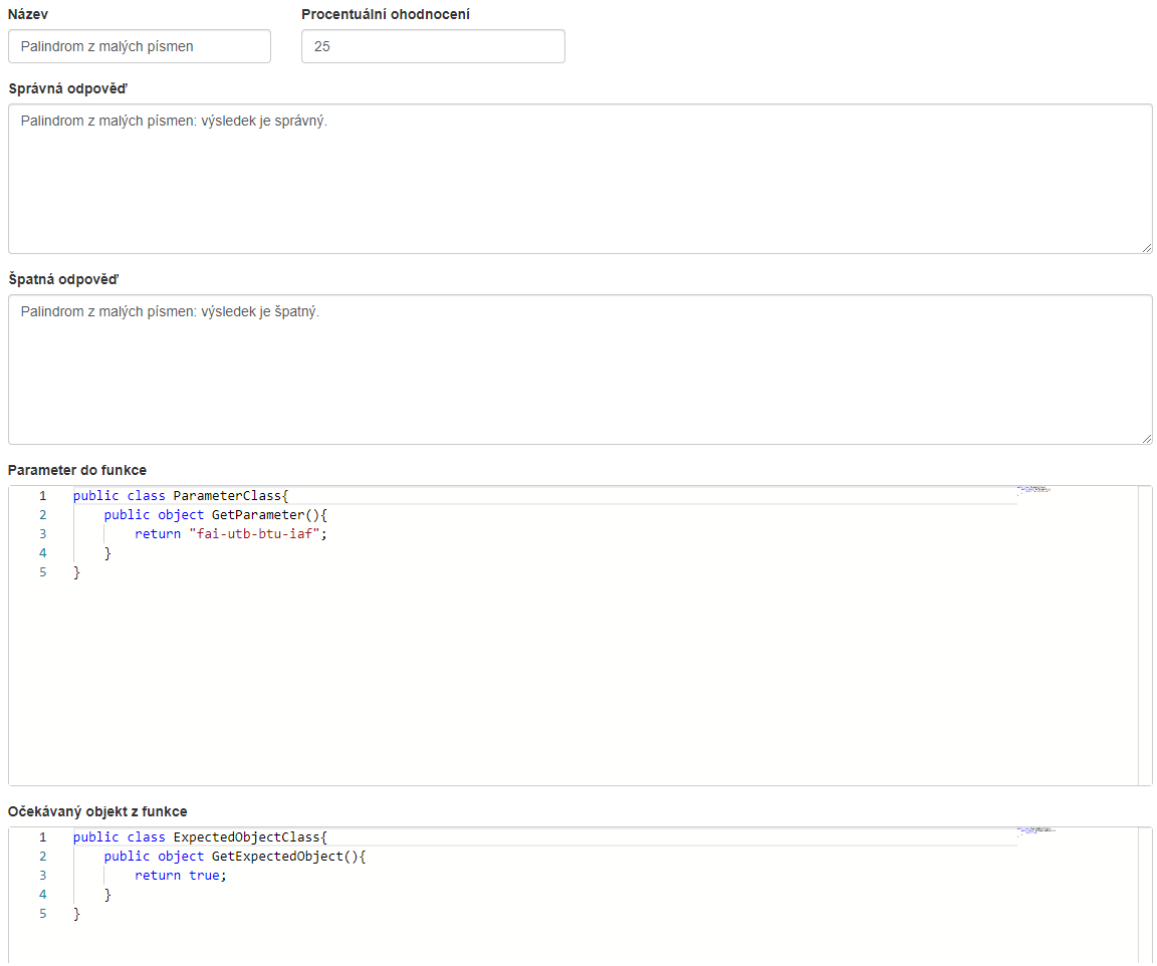
Metoda *SaveSubTest* uloží nový testovací scénář nebo upraví stávající.

```
public class RoomController : Controller
{
    private readonly ILanguageService _languageService;
    private readonly ITestRoomService _testRoomService;
    private readonly ITestScriptService _testScriptService;
    private readonly IUserService _userService;

    public RoomController(ILanguageService languageService,
        ITestRoomService testRoomService,
        ITestScriptService testScriptService,
        IUserService userService)
    {
        _languageService = languageService;
        _testRoomService = testRoomService;
        _testScriptService = testScriptService;
        _userService = userService;
    }

    public ActionResult Detail(int roomID)
    {
        RoomUsersViewModel viewModel = new RoomUsersViewModel()
        {
            Users = _userService.GetUsersForRoom(roomID)
        };
        return View(viewModel);
    }

    [HttpGet]
    public ActionResult CreateRoom()
    {
        var viewModel = new RoomViewModel()
        {
            LanguageTypes = _languageService.GetAllLanguageTypes(),
            SelectedLanguageTypes = new List<Enums.LanguageType>()
        };
        return View(viewModel);
    }

    [HttpPost]
    public ActionResult CreateRoom(RoomViewModel viewModel)
    {
        viewModel.TestRoom.TestRoomGuid = Guid.NewGuid();
        var testRoomID = _testRoomService.Insert(viewModel.TestRoom);
        _testScriptService.GenerateTests(testRoomID, viewModel.SelectedLanguageTypes,
viewModel.CountOfTest);
        return RedirectToAction("Index", "Dashboard", new {area = "Admin"});
    }

}
```

Obr. 40. Ukázka implementace třídy *RoomController*.



Obr. 41. Základní nastavení testu.



Obr. 42. Ukázka nastavení testovacího scénáře pro test.

User area

User area je určená pro uživatele, který je testován. Uživateli je zaslán URL adresa testovací místnosti. Uživateli se zobrazí formulář, kam následně vyplní své údaje, jako je jméno, příjmení a datum narození. Po přihlášení se uživateli zobrazí prostředí, kde se nachází zadání úlohy a editor pro řešení úlohy.

TestController

Metoda *Test* zobrazí uživateli testovací prostředí pro vyřešení úkolu.

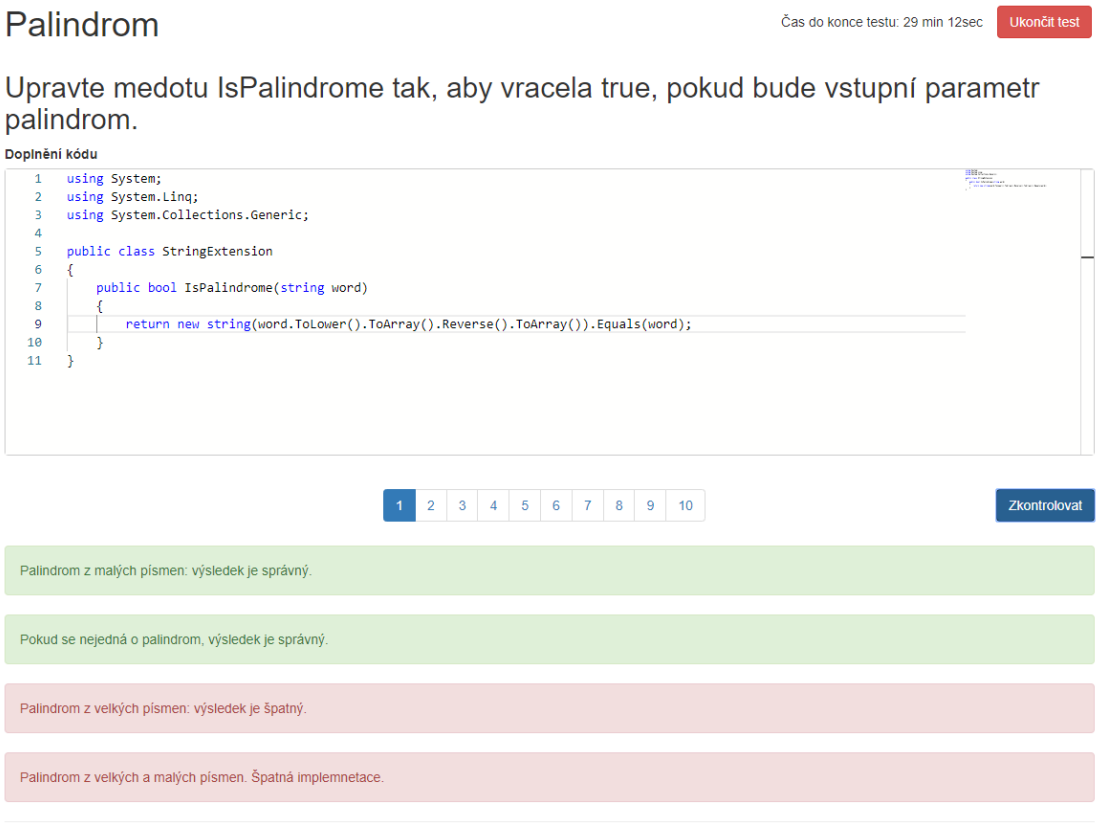
Metoda *Compile* provede kompilaci řešeného úkolu a provede vyhodnocení správnosti.

Metoda *CloseTest* provede uzavření testu po dovršení testovacího času.

UseController

Metoda *CreateUser* vytvoří nového uživatele, který se přihlásí do testovací místnosti a přesměruje ho na stránku pro vyřešení úlohy

Metoda *TestUserDetail* slouží pro zobrazení statistiky, jak si uživatel v jednotlivých úkolech vedl.



Obr. 43. Ukázka testovacího prostředí.

9 VYHODNOCENÍ TESTOVACÍCH ÚLOH

9.1 Online editor

Pro editaci kódu byl vybrán Monaco editor od firmy Microsoft. Jedná se o nejnovější editor, který je napsán v programovacím jazyce TypeScript. Další důvod, proč byl zvolen zrovna tento editor, je ten, že je na něm postaveno vývojové prostředí Microsoft Visual Code, které se ve statistikách o nejpoužívanější IDE umístilo na první příčce [19]. Monaco editor je k dispozici jako NPM balíček. Pro stažení je potřeba nainstalovat Node.js. Následně je nutné stáhnout editor pomocí příkazu na obrázku 26.

```
npm install monaco-editor
```

Obr. 44. Instalace Monaco editoru.

9.1.1 Implementace editoru

```
@{
    var url = Request.Url.Scheme + "://" + Request.Url.Authority +
Request.ApplicationPath.TrimEnd('/') + "/";
}
<script src="~/Scripts/node_modules/monaco-editor/min/vs/loader.js"></script>
<script>
    require.config({ paths: { 'vs': '@(url + "/Scripts/node_modules/monaco-editor/min/vs")' } });
</script>
```

Obr. 45. Načtení potřebných souborů pro spuštění Monaco editoru.

Abychom mohli používat editor, je potřeba načíst soubor loader.js. Pomocí funkce *require.config* nastavíme cestu, kde se nacházejí další potřebné soubory, pro běh editoru ve webovém prostředí.

```
@using DycomV1.Core.Extension
@model DycomV1.Web.ViewModel.EditorViewModel

<script>
    require(['vs/editor/editor.main'], function () {

        var editor = monaco.editor.create(document.getElementById('@Model.WrapperSelector'), {
            value: `@Html.Raw(Model.ScriptContent)`,
            language: `@Model.LanguageType.ToStringForEditor()`,
            theme: 'vs-dark'
        });

        editor.onDidChangeModelContent(() => {
            $("#@Model.ValueSelector").val(editor.getValue())
        });
    });
</script>
```

Obr. 46. Inicializace Monaco editoru.

Opět pomocí metody *require* si načteme soubory potřebné pro běh editoru. Vytvoříme instanci objektu pomocí metody *monaco.editor.create*. Jako parametr tato funkce přebírá HTML element a objekt, kde je popsáno nastavení editoru.

value – kód který bude zobrazen v editoru.

language – jazyk, ve kterém se bude programovat.

theme – prostřední editoru.

V případě změny kódu v editoru je volána funkce *editor.onDidChangeModelContent*, která vyvolá anonymní funkci. Tato funkce nám uloží kód do HTML elementu ve formuláři.

9.2 Testování jazyk C#

Pro dynamickou kompilaci jazyka C# byl použit NuGet balíček Microsoft.CodeDom.Providers.DotNetCompilerPlaform, což je kompletní platforma pro kompilování C# kódu.

```
try
  {
      var syntaxTree = SyntaxFactory.ParseSyntaxTree(userScript.ScriptContent);

      string assemblyName = Path.GetRandomFileName();
      MetadataReference[] references =
      {
          MetadataReference.CreateFromFile(typeof(object).Assembly.Location),
          MetadataReference.CreateFromFile(typeof(System.Linq.Queryable).Assembly.Location),
          MetadataReference.CreateFromFile(typeof(System.Collections.IList).Assembly.Location),
          MetadataReference.CreateFromFile(typeof(string).Assembly.Location),
          MetadataReference.CreateFromFile(typeof(Enumerable).Assembly.Location)
      };

      CSharpCompilation compilation = CSharpCompilation.Create(assemblyName, new[] {syntaxTree},
                  references,new CSharpCompilationOptions(OutputKind.DynamicallyLinkedLibrary));

      using (var memoryStream = new MemoryStream())
      {
          EmitResult resultCompilation = compilation.Emit(memoryStream);
          if (!resultCompilation.Success)
          {
              result.IsSuccessExecution = false;
              result.RunTimeErrors = resultCompilation.Diagnostics
                  .Where(diagnostic =>
                      diagnostic.IsWarningAsError ||
                      diagnostic.Severity == DiagnosticSeverity.Error)
                  .Select(x => $"{x.Id} - {x.GetMessage()}").ToList();
              return result;
          }

          memoryStream.Seek(0, SeekOrigin.Begin);
          var assembly = Assembly.Load(memoryStream.ToArray());
          var classType = assembly.GetType(userScript.ObjectToInvoke);
          var instance = Activator.CreateInstance(classType);


          result.ExpectedObject = classType.InvokeMember(userScript.MethodToInvoke,
           BindingFlags.InvokeMethod,null, instance, new[] { parameter });
          result.IsSuccessExecution = true;
          return result;
      }
  }
  catch (Exception e)
  {
      result.RunTimeErrors.Add(e.Message);
      result.IsSuccessExecution = false;
      return result;
  }
```

Obr. 47. Kompilace jazyka C#.

Statická třída *SyntaxFactory* obsahuje metodu *ParseSyntaxTree* která vytvoří syntaktický strom ze řetězce, který je jako parametr. Poté se vygeneruje náhodné jméno pro assembly. Do pole *references* jsou uloženy reference na assembly, které budou využity při spuštění zkompilovaného kódu. Nastavíme kompilaci pomocí statické třídy *CSharpCompilation* a

metody *Create*. Jako parametry tato metoda přebírá název assembly, syntaktický strom, reference použité při vykonávání zkompilovaného kódu a instanci třídy *CSharpCompilationOptions*. Výsledná kompilace bude zkompilovaná do DDL knihovny.

Poté se provádí samotná kompilace za pomocí metody *Emit*. V případě neúspěšné kompilace nám kompilátor vrátí chybové hlášky, které jsou uloženy do objektu, a tento objekt je vrácen. Pokud se kompilace provede správně, načteme zkompilovanou knihovnu pomocí *Assembly.Load*, zjistíme typ třídy, která se nachází ve zkompilované assembly a vytvoříme její instanci. Poté musíme vyvolat požadovanou metodu na této instanci objektu. To se provede za pomocí metody *InvokeMember*, která přebírá parametry jako název metody pro vyvolání, nastavení volání, instanci třídy na které je metoda spouštěna a pole objektů, což reprezentuje parametry do metody. Vrácený objekt z vyvolané metody se uloží do objektu *result.ExpectedObject*.

9.3 Kompilace jazyka JavaScript

Pro spouštění jazyka JavaScript je použit NuGet balíček Jurassic. Nejedná se však o interpreter jazyka JavaScript, nýbrž o kompilátor JS na kód pro .NET. To umožňuje snadnější kontrolu výsledných objektů z metod.

```
private BaseCompilationResult ExecuteUserScript(
    TestScript userScript,
    BaseCompilationResult result,
    bject parameter)
{
    try
    {
        var javaScriptEngine = new ScriptEngine();
        javaScriptEngine.Execute(userScript.ScriptContent);
        result.ExpectedObject = javaScriptEngine.
          CallGlobalFunction<object>(userScript.MethodToInvoke, parameter);
        result.IsSuccessExecution = true;
        return result;
    }
    catch (Exception e)
    {
        result.RunTimeErrors.Add(e.Message);
        result.IsSuccessExecution = false;
        return result;
    }
}
```

Obr. 48. Metoda *ExecuteUserScript* pro spuštění JS kódu v .NET.

Vytvoříme si instanci třídy *ScriptEngine*, následně pomocí metody *Execute* vložíme JavaScriptový kód, který je spuštěn metodou *CallGlobalFunction*. Tato metoda je generická a vrátí nám objekt, který je následně porovnán s očekávaným objektem.

9.4 Testování jazyka MS-SQL

Testování jazyka MS-SQL je realizováno voláním procedury *dbo.TestingProcedure*, která vrátí kolekci výsledků.

```
ALTER PROCEDURE [dbo].[TestingProcedure]
    @SqlCommand nvarchar(MAX)
AS
BEGIN
    BEGIN TRANSACTION TestingTransaction
        EXECUTE sp_executesql @SqlCommand
    ROLLBACK TRANSACTION TestingTransaction

    DECLARE @MaxColor int = (SELECT TOP 1 MAX(ColorID) FROM Color)
    DECLARE @MaxClient int = (SELECT TOP 1 MAX(ClientID) FROM Client)
    DECLARE @MaxCar int = (SELECT TOP 1 MAX(CarID) FROM Car)
    DECLARE @MaxRegion int = (SELECT TOP 1 MAX(RegionID) FROM Region)


    DBCC CHECKIDENT (Color, reseed, @MaxColor)
    DBCC CHECKIDENT (Car, reseed, @MaxCar)
    DBCC CHECKIDENT (Client, reseed, @MaxClient)
    DBCC CHECKIDENT (Region, reseed, @MaxRegion)
END
```

Obr. 49. Procedura *dbo.TestingProcedure* pro spouštění skriptů od uživatele.

Spouštění uživatelských skriptů je prováděno za pomocí procedury *dbo.TestingProcedure*. Parametr *SqlCommand* je uživatelský skript pro spuštění. Při vykonávání procedury se spustí transakce. Poté se pomocí systémové procedury *sp_executesql* spustí uživatelský skript. Následně se provede rollback transakce, což nám vrátí celou databázi do původního stavu. V poslední řadě musíme provést reset sloupců ID v každé tabulce, jelikož jsou tyto sloupce nastaveny jako identity.

```
public class TestingRepository : ITestingRepository
{
    private readonly IDatabaseProvider _dbProvider;

    public TestingRepository(IDatabaseProvider databaseProvider)
    {
        _dbProvider = databaseProvider;
    }

    public IEnumerable<dynamic> ExecuteQeury(string sqlCommand)
    {
        using (var db = _dbProvider.GetTestingConnection())
        {
            return db.Query<dynamic>("dbo.TestingProcedure",
                new { SqlCommand = sqlCommand }, commandType: CommandType.StoredProcedure);
        }
    }
}
```

Obr. 50. Implemetace třídy *TestingRepository* pro volání testovací procedury.

Třída *TestingRepository* slouží pro volání procedury *dbo.TestingProcedure* s parametrem *SqlCommand*, což je uživatelem definovaný skript.

```
public BaseCompilationResult Compile(TestScript testScript)
  {
      var testResult = new BaseCompilationResult();
      var test = _testScriptService.TestSettingByID(testScript.TestScriptID);
      var subTests = _testScriptService.GetSubTests(test.TestScriptID);

      foreach (var subTest in subTests)
      {
          try
          {
              var expectedResult = _testingRepository.ExecuteQeury(testScript.ScriptContent).
                  Select(x => (ExpandoObject)ObjectExtension.ToExpandoObject(x)).ToList().
                  Select(x => x.ToList()).ToList();

              var userResult = _testingRepository.ExecuteQeury(subTest.ExpectedObject).
                  Select(x => (ExpandoObject)ObjectExtension.ToExpandoObject(x)).ToList().
                  Select(x => x.ToList()).ToList();

              if (expectedResult.IsEqual(userResult))
              {
                  testResult.SuccessMessage.Add(subTest.SuccessMessage);
                  testResult.Percentage += subTest.PercentageAccuracy;
              }
              else
              {
                  testResult.ErrorMessage.Add(subTest.ErrorMessage);
              }
          }
          catch (Exception e)
          {
              testResult.RunTimeErrors.Add(e.Message);
              return testResult;
          }
      }
      return testResult;
  }
```

Obr. 51. Implementace pro řízení správnosti testu.

Metoda *Compile* ve třídě *SqlService* provádí vyhodnocení testů. Při zavolání metody *ExecuteQuery* nám vrátí metoda kolekcí objektů, které jsou poté přetypované na *ExpandoObject* pomocí extension metody *ToExpandoObject*. Dále je spuštěn skript v testovacím scénáři, provede se stejný proces jako pro uživatelsky definovaný skript a výsledné objekty se porovnají metodou *IsEqual*.

```
public static dynamic ToExpandoObject(object value)
{
    IDictionary<string, object> dapperRowProperties = value as IDictionary<string, object>;

    IDictionary<string, object> expando = new ExpandoObject();

    foreach (KeyValuePair<string, object> property in dapperRowProperties)
        expando.Add(property.Key, property.Value);

    return expando as ExpandoObject;
}
```

Obr. 52. Extension metoda *ToExpandoObject*.

9.5 Kontrola správnosti výsledku

Každá kompilace uživatelského testu vrátí objekt, který je potřeba porovnat s očekávaným objektem. K tomu slouží extension metoda *IsEqual*. Pro porovnání objektů je použit NuGet balíček CompareNETObjects. Jde o velice užitečný nástroj, který porovnává objekty do hloubky za pomocí reflexe. Jeho další výhoda je nastavení porovnávání.

```
public static class ObjectExtension
{
    public static bool IsEqual(this object baseObject, object objectToCompare)
    {
        var configuration = new ComparisonConfig()
        {
            IgnoreObjectTypes = true,
            IgnoreUnknownObjectTypes = true,
        };
        var comparer = new CompareLogic(configuration);
        return comparer.Compare(baseObject, objectToCompare).AreEqual;
    }
}
```

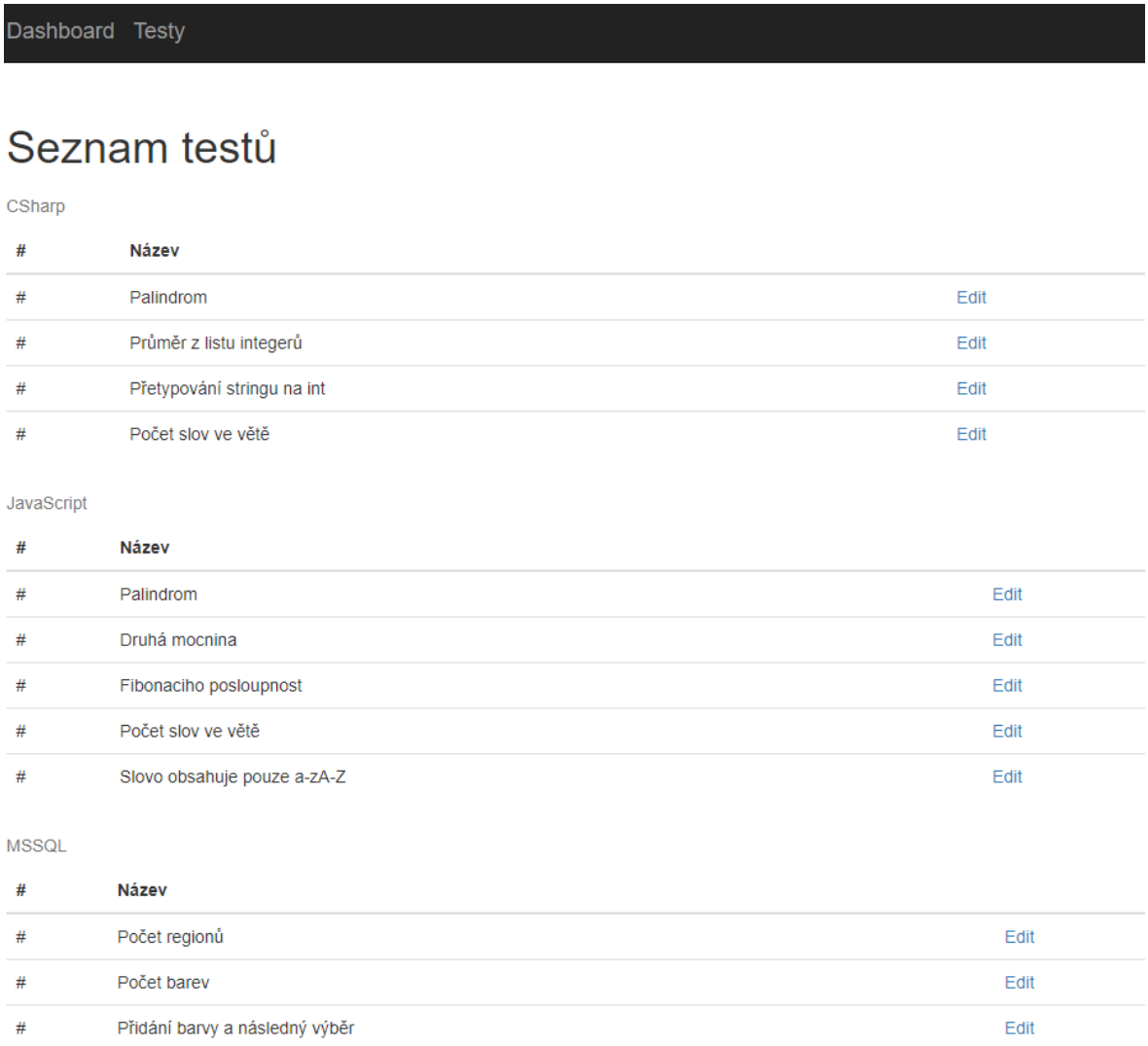
Obr. 53. Implementace porovnávání objektů.

Jelikož se jedná o extension metodu, je volána nad porovnávaným objektem a jako parametr přijímá další objekt pro porovnání. V konfiguraci porovnávání je nastaveno ignorování typů objektů a ignorování neznámých typů objektů. Poté se vytvoří instance třídy *CompareLogic*,

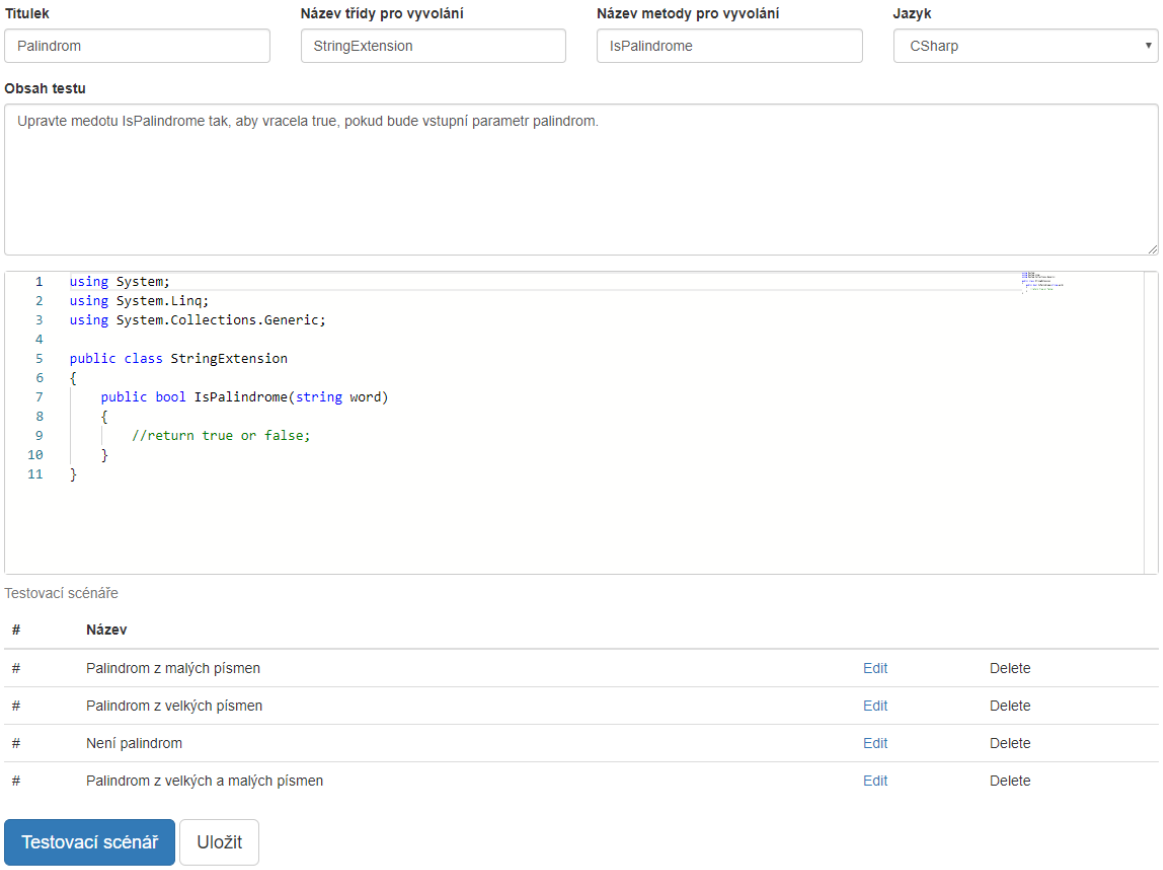
která přebírá jako parametr v konstruktoru konfiguraci. Pro porovnání je volána metoda *Compare*, která dva objekty porovná.

10 TESTOVACÍ ÚLOHY

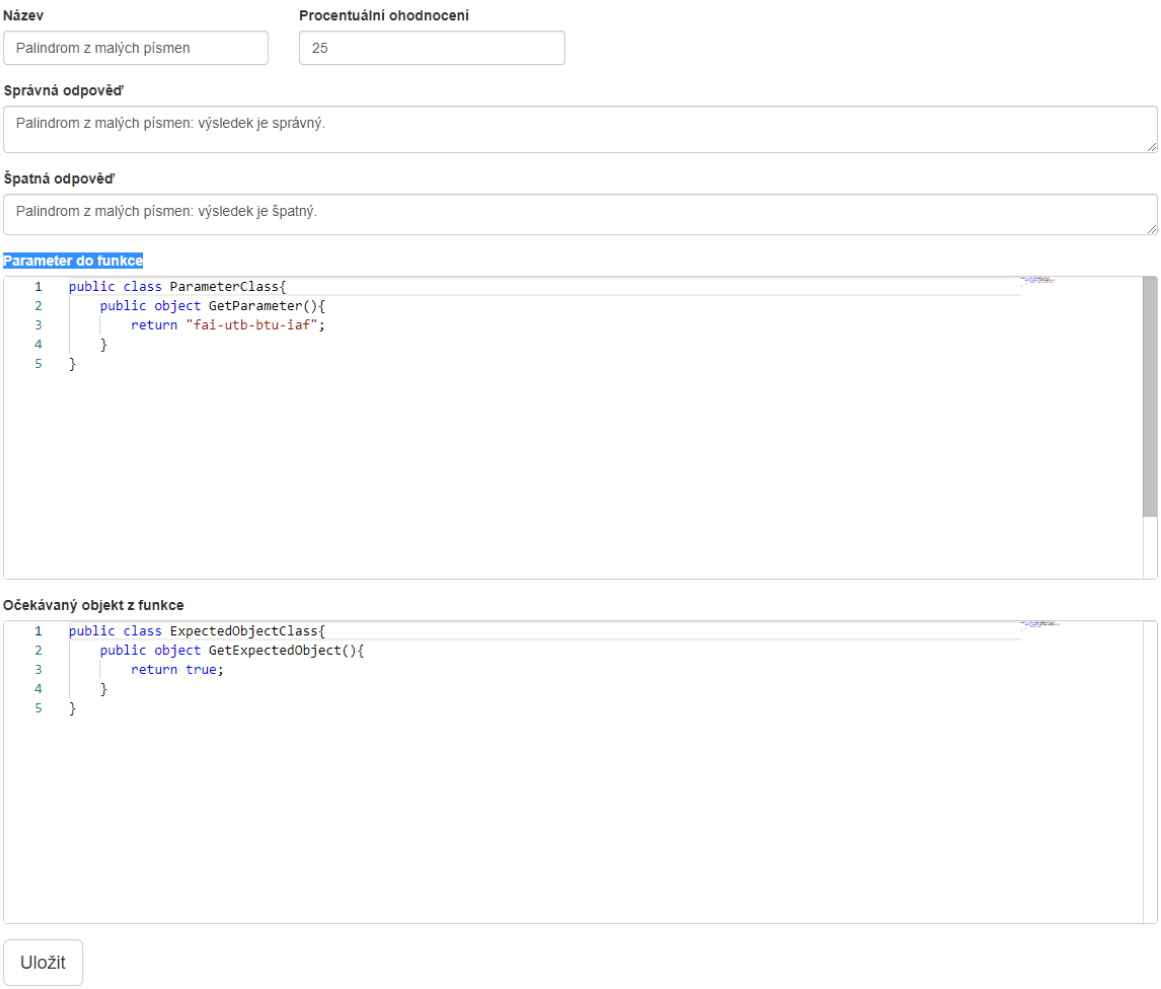
Pro demonstraci aplikace, byli vytvořeny testovací úkoly pro každý programovací jazyk.



Obr. 54. Stránka pro se seznamem testů.



Obr. 55. Stránka pro vytvoření testu.



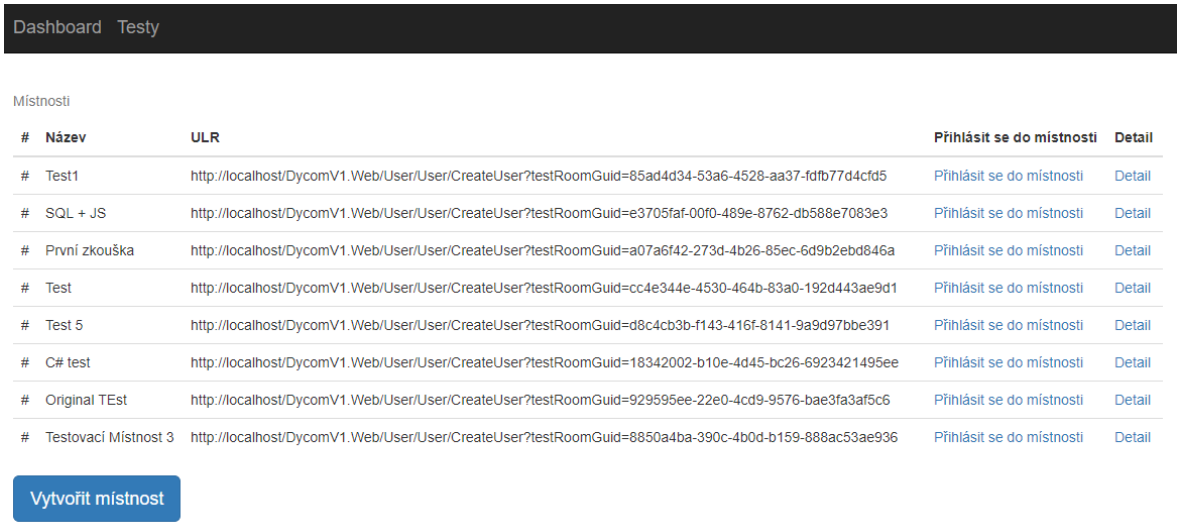
Obr. 56. Detail testovacího scénáře.

11 OVĚŘENÍ FUNKČNOSTI

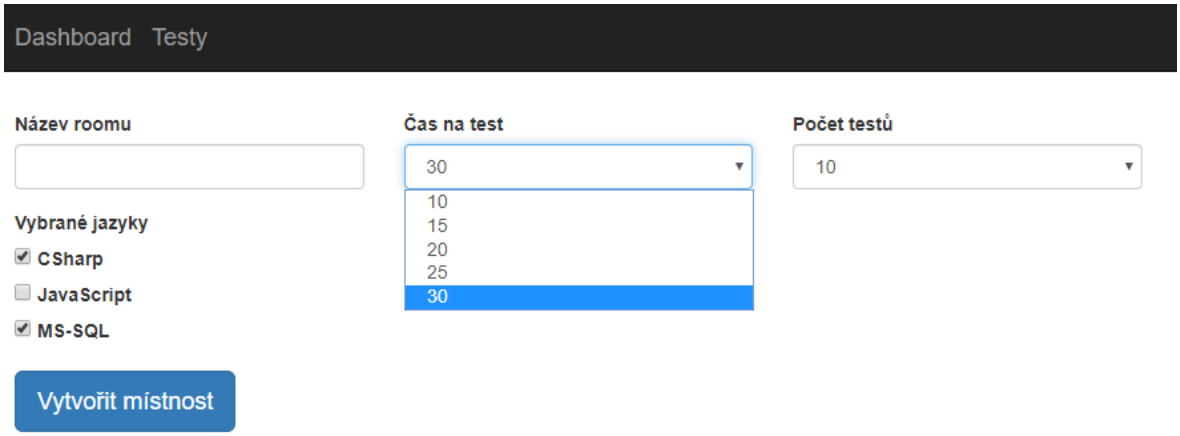
Při realizaci webové aplikace bylo postupováno podle zadání. Nejdříve byly vytvořeny grafické návrhy, jak by aplikace mohla vypadat. Dále byla navržena vhodná architektura aplikace a následně se provedla implementace webové aplikace. Webová aplikace obsahuje administrátorskou a klientskou část. V administrátorské části lze přidávat místnosti a zobrazit si jejich detail, dále je zde možné vytvářet testovací úlohy a scénáře pro ně. Klientská část obsahuje registraci do testovací místnosti a také přehledné a srozumitelné testovací prostředí.

Po vytvoření testovacích úkolů bylo vytvořeno několik testovacích místností různých variant. Poté byly provedeny jednotlivé testy s několika uživateli, pro ověření testovacích úloh.

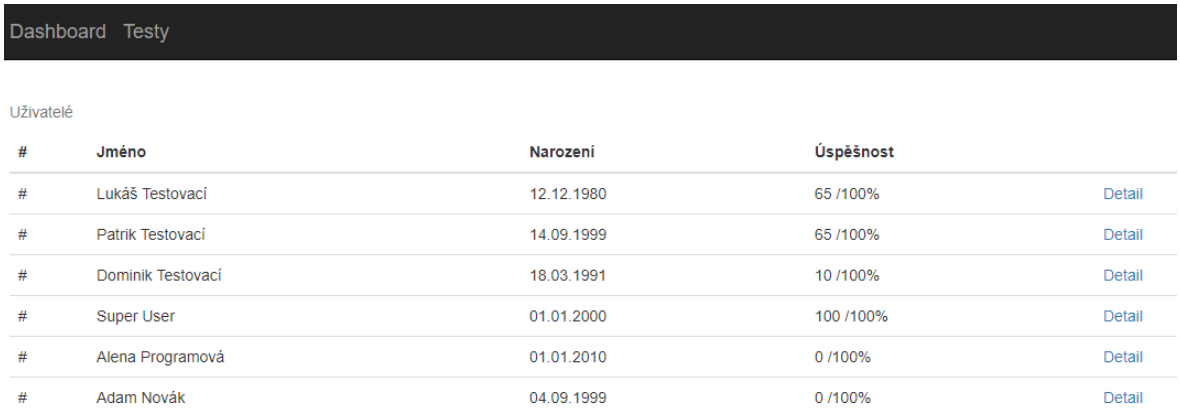
11.1 Administrátorská část



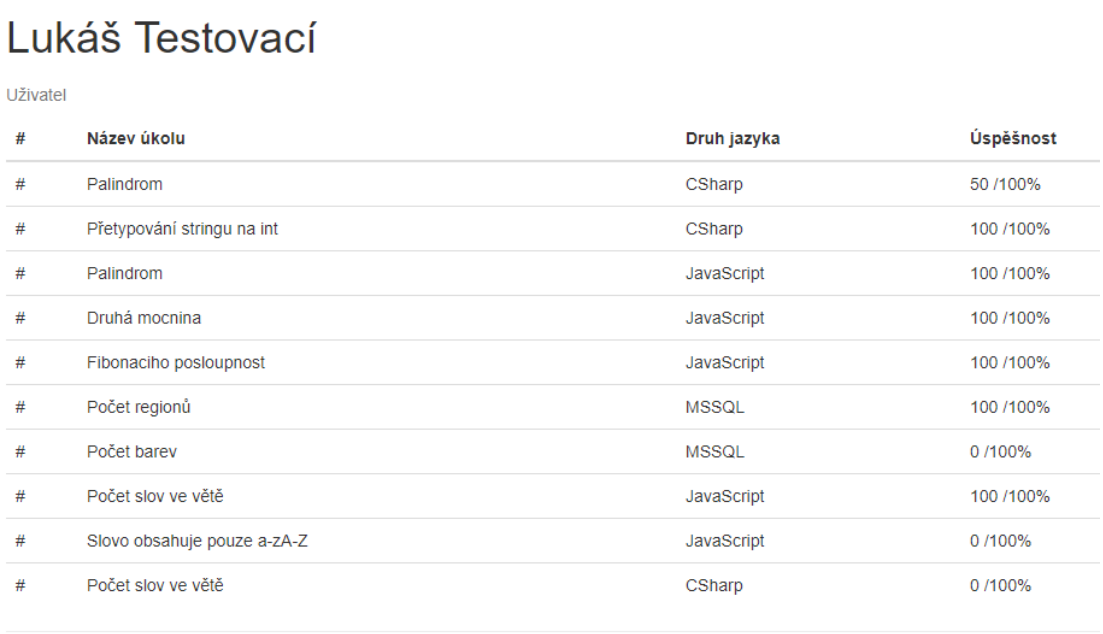
Obr. 57. Seznam testovacích místností.



Obr. 58. Stránka pro vytvoření místnosti.



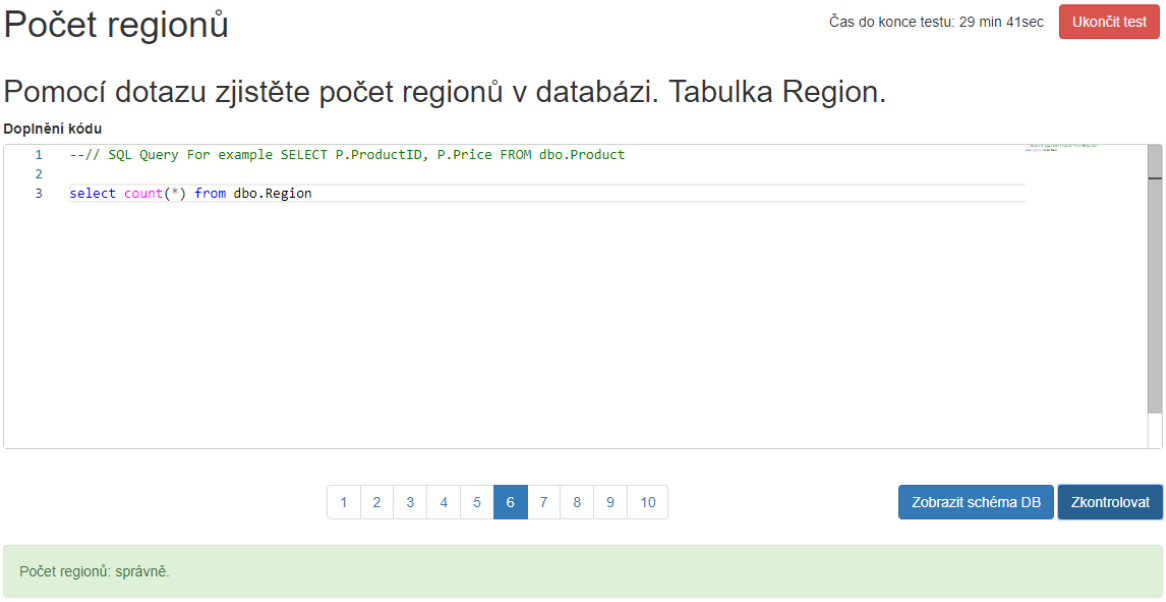
Obr. 59. Stránka se seznamem uživatelů v testovací místnosti.



Obr. 60. Stránka se seznamem úspěšnosti pro jednotlivé testy uživatele.

11.2 Uživatelská část

Obr. 61. Stránka pro registraci uživatele do místnosti.



Obr. 62. Stránka prostředí pro testování.

ZÁVĚR

Cílem diplomové práce bylo navrhnout a realizovat webovou aplikaci, jejíž hlavní úlohou bude testování uživatelů z více programovacích jazyků. Jako testovací jazyky byly zvoleny C#, JavaScript a MS-SQL.

Před samostatným vývojem bylo potřeba rozhodnout, které nástroje a technologie budou použity pro vývoj aplikace, a jak bude celkově aplikace ve výsledku vypadat. Pro aplikaci byl zvolen .NET Framework, respektive ASP.NET Framework MVC. Dále bylo nutné navrhnout vhodnou architekturu celé aplikace, která umožňuje testování více programovacích jazyků. Pro přehlednost jsou v solution celkem tři projekty, a každý z nich má specifickou funkci. Projekt *DycomV1.Repository* se stará o komunikaci s databází. Projekt *DycomV1.Core* obsahuje veškerou logiku celé aplikace a projekt *DycomV1.Web* je webový projekt. Pro rychlejší a přehlednější vývoj bylo zavedeno do projektu dependency injection. Díky správně navržené architektuře, je systém snadno rozšiřitelný a v případě potřeby není problém implementovat novou servisní třídu pro nový programovací jazyk.

Pro editor zdrojového kódu byl použit Monaco editor od firmy Microsoft, jelikož je na něm založen projekt Microsoft Visual Code a mnoho lidí tento editor používá. Další důvod, proč byl tento editor zvolen, je ten, že je napsán v TypeScriptu, což zaručuje typovou bezpečnost pro případnou další úpravu systému.

Vytvořená aplikace je rozdělena na dvě separátní sekce. Je zde administrátorská část, kde administrátor může přidávat testovací místnosti, testy a jejich testovací scénáře. V případě přidávání testů a testovacích scénářů, je nutné znát programovací jazyky, pro které jsou testy vytvářeny. Část pro klienta obsahuje registraci do testovací místnosti a testovací prostředí, kde jsou testy řešeny.

Pro ověření funkčnosti celé aplikace byly vytvořeny testy pro každý programovací jazyk. Následně byly vytvořeny testovací místnosti pro uživatele. Uživatel se poté mohl registrovat do místnosti a zkusit test, kde si v průběhu řešení testu může provést správnost řešení. V případě výskytu chyb za běhu programu, jsou tyto chyby uživateli zobrazeny, aby mu dopomohly ke správnému vyřešení zadání.

SEZNAM POUŽITÉ LITERATURY

[1] AGARWAL, Tarun. *What is the Programming Language and Differences* [online]. [cit. 2019-05-18]. Dostupné z: https://www.edgefxkits.com/blog/types-of-programming-languages-with-differences/

[2] ALBRIGHT, Dann. *High-Level vs. Low-Level Programming Languages: Which Should You Learn?*[online]. 9 November 2017 [cit. 2019-05-18]. Dostupné z: https://www.makeuseof.com/tag/high-level-low-level-programming-languages/

[3] *Computer programming languages and its types*[online]. [cit. 2019-05-18]. Dostupné z: https://www.includehelp.com/basics/computer-programming-languages.aspx

[4] WODEHOUSE, Carey. *The Basics of Compiled Languages, Interpreted Languages, and Just-in-Time Compilers* [online]. [cit. 2019-05-18]. Dostupné z: https://www.upwork.com/hiring/development/the-basics-of-compiled-languages-interpreted-languages-and-just-in-time-compilers/

[5] NAGEL, Christian, Bill EVJEN, Jay GLYNN, Morgan SKINNER a Karli WATWON. *C# 2008: Programujeme profesionálně*. Brno: Computer Press, 2009. Programujeme profesionálně. ISBN 978-80-251-2401-7.

[6] JANÁČEK, Ondřej. *Seznamte se s roslynem* [online]. 26. června 2014 [cit. 2019-05-18]. Dostupné z: https://www.dotnetportal.cz/blogy/12/Ondrej-Janacek/6389/Seznamte-se-s-Roslynem

[7] PEHLIVANIAN, Ara a Don NGUYEN. *JavaScript okamžitě*. Brno: Computer Press, 2014. ISBN 978-802-5141-632.

[8] *About JavaScript: What is JavaScript?* [online]. [cit. 2019-05-18]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/JavaScript/About_JavaScript

[9] HAHNEKAMP, Rainer. *JavaScript essentials: why you should know how the engine works* [online]. 4. července 2018 [cit. 2019-05-18]. Dostupné z: https://medium.freecodecamp.org/javascript-essentials-why-you-should-know-how-the-engine-works-c2cc0d321553

[10] ROUSE, Margeret. *T-SQL (Transact-SQL)* [online]. In: . [cit. 2019-05-18]. Dostupné z: https://searchsqlserver.techtarget.com/definition/T-SQL

[11] FENTON, Steve. *Pro TypeScript: Application-Scale JavaScript Development*. New York, N.Y.: Apress, [2014]. ISBN 978-1-4302-6791-1.

[12] BROWN, Simon. *Software Architecture for Developers*. Lean Publishing, 2014.

[13] MARTIN, Robert C. *Design Principles and Design Patterns* [online]. 2000 [cit. 2019-05-18]. Dostupné z: https://fi.ort.edu.uy/innovaportal/file/2032/1/design_principles.pdf

[14] *Dependency injection in ASP.NET Core* [online]. In: . 7. dubna 2019 [cit. 2019-05-18]. Dostupné z: https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-2.2&viewFallbackFrom=aspnetcore-2.0

[15] *Dependency Injection* [online]. In: . 5. listopadu 2013 [cit. 2019-05-18]. Dostupné z: https://docs.microsoft.com/en-us/previous-versions/msp-n-p/dn178469(v=pandp.30)

[16] *Get started with the .NET Framework* [online]. In: . 2. dubna 2019 [cit. 2019-05-18]. Dostupné z: https://docs.microsoft.com/cs-cz/dotnet/framework/get-started/

[17] MACDONALD, Matthew, Adam FREEMAN a Mario SZPUSZTA. *ASP.NET 4 a C# 2010: tvorba dynamických stránek profesionálně*. Brno: Zoner Press, 2011. Encyklopedie Zoner Press. ISBN 978-80-7413-131-8.

[18] GALLOWAY, Jon, Brad WILSON, K. Scott ALLEN a David MATSON. *Professional ASP.NET MVC 5*. Indianapolis, IN: Wrox, a Wiley brand, [2014]. Wrox professional guides. ISBN 978-1-118-79475-3.

[19] Developer Survey Results 2019. *Stack Overflow*[online]. [cit. 2019-05-18]. Dostupné z: https://insights.stackoverflow.com/survey/2019

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

API	Application Programming Interface
ASP	Active Server Page
CLI	Common Intermediate Language
CLR	Common Language Runtime
CSS	Cascading Style Sheets
DI	Dependency Injection
DLL	Dynamic-Link Library
GC	Garbage Collector
HTML	Hypertext Markup Language
ID	Identity
IDE	Integrated Development Enviroment
JIT	Just In Time
JS	JavaScript
JSON	JavaScript Object Notation
MVC	Model View Controller
NPM	Node Package Manager
PHP	Hypertext Preprocesor
SQL	Structured Query Language
TS	TypeScript
T-SQL	Transact Structured Query Language
VB	Visual Basic
WCF	Windows Common Foundation
XML	Extensible Markup Language

SEZNAM OBRÁZKŮ

SEZNAM PŘÍLOH

PI CD disk se zdrojovými kódy a diplomovou prací