Využitie technológie Webassembly pre výuku programovania

Bc. Tomáš Tuska

Diplomová práca
2020

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně

Fakulta aplikované informatiky

Ústav informatiky a umělé inteligence

Akademický rok: 2019/2020

ZADÁNÍ DIPLOMOVÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Bc. Tomáš Tuska**
Osobní číslo:	**A18279**
Studijní program:	**N3902 Inženýrská informatika**
Studijní obor:	**Informační technologie**
Forma studia:	**Kombinovaná**
Téma práce:	**Využití technologie WebAssembly pro výuku programování**
Téma práce anglicky:	**The Use of WebAssembly Technology for Teaching Programming**

Zásady pro vypracování

1. Popište webový standard WebAssembly a související technologie.
2. Popište technologii Try.Net a její možnosti.
3. Zhodnoťte možnosti podpory výuky programování s využitím technologie Try.Net.
4. Ověřte možnosti automatické kontroly testů vypracovaných studenty s pomocí technologie Try.Net.
5. Vytvořte a popište klíčové prvky navrženého řešení.

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

Seznam doporučené literatury:

1. WebAssembly [online]. [cit. 2019-11-14]. Dostupné z: https://webassembly.org
2. Try .NET: Runnable .NET code on your site [online]. [cit. 2019-11-14]. Dostupné z: https://dotnet.microsoft.com/platform/try-dotnet
3. NAGEL, Christian. Professional C# 7 and .NET Core 2.0. 11th edition. Indianapolis: Wrox, a Wiley Brand, 2018. ISBN 978-1119449270.
4. ALBAHARI, Joseph a Ben ALBAHARI. C# 7.0 in a nutshell. 7th edition. Sebastopol: O'Reilly, 2018. ISBN 978-1491987650.
5. V. HAAS, Andreas, Andreas ROSSBERG, Derek L. SCHUFF, et al. Bringing the web up to speed with WebAssembly. HAAS, Andreas, Andreas ROSSBERG, Derek L. SCHUFF, et al. Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation. New York, NY: ACM, 2017, s. 185-200. ISBN 978-1-4503-4988-8.

Vedoucí diplomové práce: **Ing. Erik Král, Ph.D.**
Ústav počítačových a komunikačních systémů

Datum zadání diplomové práce: **28. listopadu 2019**
Termín odevzdání diplomové práce: **15. května 2020**

doc. Mgr. Milan Adámek, Ph.D.
děkan

prof. Mgr. Roman Jašek, Ph.D.
ředitel ústavu

Ve Zlíně dne 9. prosince 2019

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 29.7.2020 Tomáš Tuska, v. r.

ABSTRAKT

Práca sa venuje opisu technológií WebAssembly, .NET, Blazor a Try .NET. Hlavným cieľom práce je zhodnotiť a demonštrovať možnosti využitia technológie Try .NET v rámci výuky .NET technológií ako je napríklad výuka jazyka C#. Tieto ciele boli naplnené v praktickej časti práce kde bola navrhnutá a vytvorená webová aplikácia na automatickú kontrolu testov vypracovaných študentami.

Kľúčové slová: WebAssembly, WASM, .NET, Blazor, Try .NET, Mono, Monaco

ABSTRACT

The work deals with the description of technologies as WebAssembly, .NET, Blazor and Try .NET. The main goal of this work is to evaluate and demonstrate the possibilities of using Try .NET technology in the process of teaching .NET technologies such as teaching C# programming language. These goals were fulfilled in the practical part of the work where a web application was designed and created for automatic control of tests developed by students.

Keywords: WebAssembly, WASM, .NET, Blazor, Try .NET, Mono, Monaco

OBSAH

ÚVOD

Diplomová práca sa venuje opisu technológie Try .NET od spoločnosti Microsoft. Táto technológia poskytuje možnosť užívateľom vytvárať interaktívne dokumentácie so spustiteľným C# kódom a integrovať spustiteľný kód do ľubovoľných stránok pomocou editora zdrojového kódu monaco a webového štandardu WebAssembly. Hlavným cieľom práce je preskúmať možnosti využitia technológie Try .NET v rámci výuky .NET technológií ako je napríklad jazyk C# a následne tieto možnosti overiť vytvorením webovej aplikácie.

Teoretická časť práce poskytuje čitateľovi všeobecný prehľad o technológiách, ktoré sú priamo využité technológiou Try .NET. Medzi tieto technológie patria napríklad WebAssembly, UI framework Blazor a prekladač Roslyn. Posledná kapitola teoretickej časti sa venuje opisu samotnej technológie Try .NET. V tejto kapitole bude čitateľ oboznámený s procesmi slúžiacimi na preklad a spustenie užívateľského kódu, príkladmi použitia a rozdielmi jednotlivých módov tejto technológie.

V rámci praktickej časti budú navrhnuté možnosti využitia technológie Try .NET v rámci výuky .NET technológií. Jednotlivé návrhy budú demonštrované jednoduchou ukážkou. Za účelom overenia možnosti využitia technológie Try .NET na automatickú kontrolu testov vypracovaných študentmi budú v rámci praktickej časti vytvorené dve aplikácie. Obe aplikácie budú mať definované rovnaké ciele. Každá aplikácia na splnenie týchto cieľov bude využívať iný mód technológie Try .NET.

Hlavným dôvodom výberu tejto témy bola možnosť bližšieho zoznámenia sa s technológiou WebAssembly po teoretickej stránke. Ďalším dôvodom bola možnosť získať praktické skúsenosti o vytváraní webových aplikácií pomocou módu WASM UI frameworku Blazor, ktorý využíva WebAssembly na beh C# kódu vo webovom prehliadači.

I. TEORETICKÁ ČASŤ

1 WEBASSEMBLY

WebAssembly, skrátene WASM, je štandardizovaný formát binárnych inštrukcií určený pre virtuálny zásobníkový počítač. WebAssembly je navrhnutý ako prenosný cieľ kompilátorov vysoko-úrovňových jazykov ako sú napríklad C, C++ a Rust, čím umožňuje nasadenie klientských a serverových aplikácií na webe [1].

WebAssembly je nový typ kódu, spustiteľný pomocou moderných webových prehliadačov. Poskytuje nové funkcie a významné zvýšenie výkonu vykonávaného kódu. Ďalšou výhodou je, že pre využitie WebAssembly užívateľ nemusí vedieť ako vytvoriť WebAssembly kód, pretože moduly WebAssembly je možné importovať do webovej aplikácie alebo Node.js a komunikovať s nimi pomocou rozhrania v jazyku JavaScript. JavaScript frameworky ako React, Angular a Vue môžu využívať výkon WebAssembly na poskytnutie nových funkcií, pričom vývojári webových stránok môžu bez obmedzení využívať ľahko dostupné funkcie pomocou rozhrania [2].

WebAssembly je tvorené vo forme otvoreného štandardu v rámci *W3C WebAssembly Community Group* (viac informácií na [3]) s nasledujúcimi abstraktnými cieľmi [1]:

- Rýchlosť a efektívnosť – Zásobníkový počítač WebAssembly je navrhnutý tak, aby bol efektívne zakódovaný z hľadiska veľkosti a času načítania do binárneho formátu. Zameriava sa na vykonávanie kódu takmer natívnou rýchlosťou, využitím bežne dostupných výhod hardvérových funkcií dostupných na širokej škále platforiem.
- Čitateľnosť a laditeľnosť – Textový formát WebAssembly je navrhnutý s dôrazom na čitateľnosť pre účely ladenia, testovania, experimentovania, učenia, optimalizácie a priameho písania programov vo WebAssembly. Textový formát je využitý pri skúmaní zdrojových kódov WASM modulov na webe.
- Bezpečnosť – Štandard WebAssembly opisuje pamäťovo bezpečné a izolované behové prostredie, ktoré môže byť implementované aj do existujúcich virtuálnych strojov JavaScriptu. V prípade embeddovania WebAssembly do webového prostredia bude WebAssembly presadzovať zásady zabezpečenia rovnakého pôvodu a oprávnení webového prehliadača.

V nasledujúcich podkapitolách bude vysvetlené aké úlohy má WebAssembly v dnešnom ekosystéme webových technológií spĺňať, jeho formáty a generovanie. Posledné časti tejto kapitoly budú venované bezpečnosti a porovnaniu výkonu WebAssembly oproti jazykom C a JavaScript.

1.1 Vzťah medzi jazykom WASM a JavaScript

Webové platformy môžu byť rozdelené na dve časti [2]:

- Virtuálny stroj (VM – virtual machine), ktorý vykonáva kód webovej aplikácie. Vykonávaný kód je v tomto kontexte JavaScript poháňajúci aplikáciu.
- Súprava webových rozhraní, ktoré môže webová aplikácia volať na to, aby ovládala funkčnosť webového prehliadača alebo zariadenia (DOM, CSSOM, WebGL, Web Audio API a atď.).

V minulosti bol virtuálny stroj schopný načítať a spracovať iba jazyk JavaScript. Táto limitácia nepredstavovalo prekážku, pretože JavaScript dokázal vyriešiť väčšinu problémov, s ktorými sa programátori pri vývoji webových aplikácií mohli stretnúť. Výkonnostné problémy sa vyskytujú až pri náročných prípadoch použitia ako sú napríklad 3D hry, virtuálna a rozšírená realita, počítačové videnie, editovanie obrázkov a videa, prípadne pri ďalších doménach vyžadujúce natívny výkon [2].

WebAssembly nie je určený ako náhrada jazyka JavaScript. Je navrhnutý tak, aby tento jazyk dopĺňal a spolupracoval s ním. Týmto prístupom umožňuje vývojárom webových stránok využívať silné stránky oboch jazykov, ktorými sú [2]:

- JavaScript je vysoko-úrovňový jazyk, dostatočne flexibilný a expresívny pre účely webových aplikácií. Medzi výhody jazyka patria dynamická typovosť, nevyžaduje žiadny kompilačný medzi-krok a má obrovský ekosystém, poskytujúci výkonné frameworky, knižnice a nástroje.
- WebAssembly je nízko-úrovňový jazyk podobný jazykom symbolických adries s kompaktným binárnym formátom, poskytujúci kompilačný cieľ pre jazyky ako sú C, C++ a Rust, aby sa mohli spúšťať aj na webe.

Jednotlivé jazyky sa môžu navzájom podľa potreby volať. Exportovaný WebAssembly kód je zapuzdrený prostredníctvom JavaScript API. Cez túto API môžu JavaScript aplikácie komunikovať s WebAssembly modulmi. Naopak, WebAssembly môže importovať a synchrónne spúšťať JavaScript funkcie [2].

1.1.1 Súborové formáty WebAssembly

WebAssembly môže byť reprezentovaný v dvoch formátoch, ktorými sú textový a binárny formát. Prvý uvedení formát je ľahko čitateľný a slúži na priame čítanie, úpravu a vytváranie WebAssembly kódu. Súbory v tomto formáte majú príponu *.wat*. Druhý formát je binárny

formát a je určený na prenos a spracovanie webovými prehliadačmi. Prípona tohto formátu je *.wasm*. Pomocou rôznych nástrojov, ako je napríklad *wabt* [4], je možné súbory konvertovať z prvého formátu do druhého a naopak.

```
(module
  (import stdlib print (func $print (param i32 i32)))
  (import js memory (memory 20)) ;; initial allocation of 20 pages
  (data (i32.const 0) Hello world!") ;; store string in data segment at
offset=0 (length=strlen("Hello world!")=12)

  (func (export main")
    i32.const 0  ;; offset=0
    i32.const 12 ;; length=12
    call $print
  )
)
```

Obrázok 1 Ukážka textového formátu (*.wat*) [5]

```
00000000: 0061 736d 0100 0000 0109 0260 027f 7f00  .asm.......`....
00000010: 6000 0002 1d02 0673 7464 6c69 6205 7072  ......stdlib.pr
00000020: 696e 7400 0002 6a73 066d 656d 6f72 7902  int...js.memory.
00000030: 0014 0302 0101 0708 0104 6d61 696e 0001  ..........main..
00000040: 0a0a 0108 0041 0041 0f10 000b 0b12 0100  .....A.A........
00000050: 4100 0b0c 4865 6c6c 6f20 576f 726c 6421  A...Hello World!
```

Obrázok 2 Ukážka binárneho formátu (*.wasm*) [5]

Binárna reprezentácia má formu modulov, pričom každý modul obsahuje sekcie definujúce funkcie, globálne premenné, tabuľky a pamäť. Každú z uvedených definícií je možné exportovať alebo importovať do iného WebAssembly modulu [2]. Moduly musia začínať hlavičkou o veľkosti 8B. Hlavička obsahuje konštantu identifikujúcu súborový formát. V prípade WASM je táto konštanta vo formáte DWORD rovná 0x00617364 (ASCII \0asm). Za konštantou nasleduje verzia kompatibility taktiež vo formáte DWORD [5].

Obrázok 3 Štruktúra súboru WebAssembly [5]

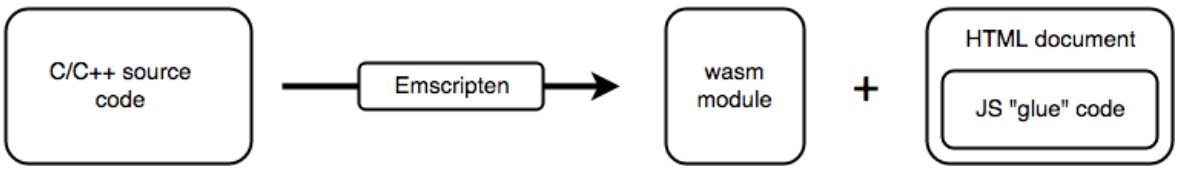
1.2 Tvorenie kódu WebAssembly

WebAssembly ekosystém sa nachádza v počiatočnej fáze svojho životného cyklu. Predpokladá sa, že s odstupom času pribudnú nové metódy tvorenia kódu, momentálne však existujú štyri hlavné spôsoby, ktorými sú [2]:

- Portovanie C/C++ aplikácií pomocou Emscripten.
- Písanie alebo generovanie WebAssembly kódu na úrovni assembly.
- Písanie aplikácie v jazyku Rust a následné preloženie do jazyka WebAssembly.
- Použitím AssemblyScript, ktorý kompiluje TypeScript do binárneho formátu WebAssembly.

1.2.1 Portovanie z jazyka C/C++

Dve z mnohých možností portovania WebAssembly kódu sú online WASM assembler alebo Emscripten. Existuje celý rad možnosti online WASM assemblerov, ako sú napríklad WasmFiddle, WasmFiddle++ a WasmExplorer. Emscripten je nástroj schopný skompilovať skoro akýkoľvek zdrojový kód v jazyku C/C++ do modulu .wasm spolu s potrebným JavaScript kódom (tzv. „glue“ kód), slúžiacim na načítanie modulu, spustenie modulu a zobrazenie HTML dokumentu na zobrazenie výsledku kódu [2].



Obrázok 4 Proces portovania C/C++ do WebAssembly [2]

V stručnosti zhrnutý proces portovania kódu C/C++ je nasledovný [2]:

1. Zdrojový kód C/C++ je skompilovaný pomocou prekladača clang/LLVM.

2. Emscripten transformuje výsledok kompilácie na binárny súbor .wasm.
3. WebAssembly v súčasnosti nedokáže priamo pristupovať k DOM. Prístup k DOM-u je umožnení pomocou JavaScript funkcií, ktoré sú spúšťané so vstupnými parametrami ako sú celé číslo alebo číslo s pohyblivou desatinnou čiarkou. Z tohto dôvodu na prístup k akémukoľvek webovému rozhraniu musí WebAssembly použiť JavaScript ako prostredníka. Za týmto účelom vytvorí nástroj Emscripten HTML dokument a JavaScript kód (tzv. „glue“ kód).

Vygenerovaný HTML dokument načíta JavaScript „glue“ kód, ktorý následne slúži ako rozhranie medzi WebAssembly modulmi a API webového prehliadača [2].

1.2.2 Písanie kódu WebAssembly

Rovnakým spôsobom ako v jazykoch symbolických adries má WebAssembly binárny formát textovú reprezentáciu. Tieto formáty si navzájom korešpondujú 1:1. Textová reprezentácia môže byť písaná alebo generovaná ručne a následne pomocou ľubovoľného nástroja WebAssembly môže byť prevedená do binárneho formátu [2].

1.2.3 AssemblyScript

AssemblyScript je nástroj kompilujúci TypeScript (jazyková nadstavba nad jazykom JavaScript) do WebAssembly za pomoci Binaryen [6]. Binaryen je prekladač a súprava knižníc vytvorená za účelom portovania jazykov ako sú JavaScript a Haskell do WebAssembly [7].

1.3 Výkon WebAssembly

Porovnanie výpočtového výkonu WebAssembly s výkonom natívnemu kódu C a JavaScriptu na rôznych platformách a rôznych prehliadačoch bolo vykonané v práci *WebAssembly and JavaScript Challenge* [8]. Práca sa venuje porovnaniu výkonov pomocou kolekcie numerických programov, pričom každý program predstavuje zástupcu jedného z trinástich tzv. dwarfov. Pomocou dwarfov sú definované kategórie programov, ktoré zdieľajú spoločné výpočtové a komunikačné vzorce [9]. Implementácie programov, zastupujúce jednotlivých dwarfov, boli implementované v jazyku C a JavaScript. Programy WebAssembly boli vytvorené pomocou nástroja Emscripten z implementácií v jazyku C.

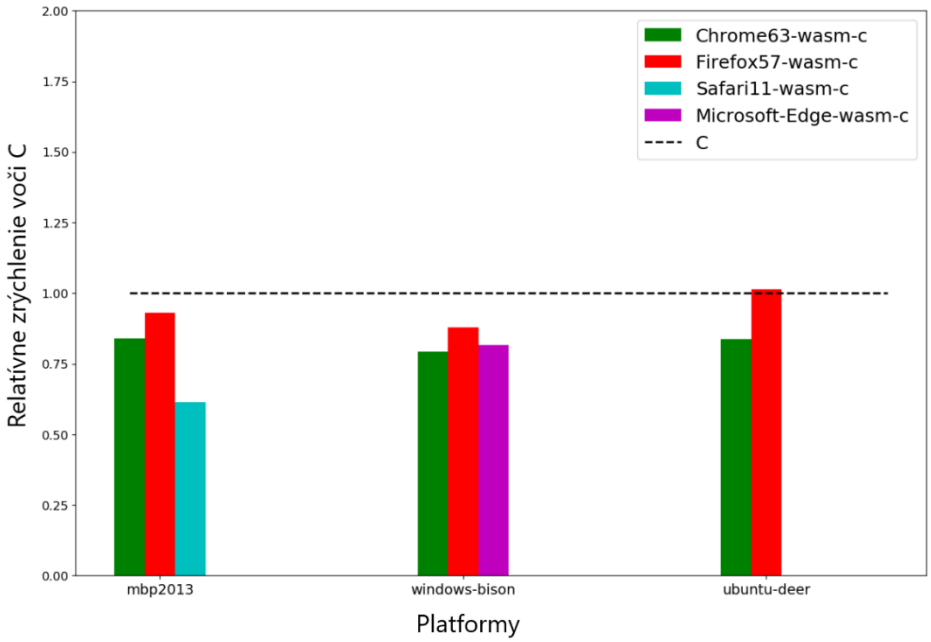
Zariadenia využité na porovnanie výkonu medzi jazykmi JavaScript, C a WebAssembly sú uvedené v tabuľke nižšie.

Platforma	Zariadenie	CPU	RAM [GB]	OS	GCC
Notebooky a pracovné stanice	MacBook Pro 2013 (mbp-2013)	Intel Core i5 @ 2,4 GHz	8	MacOS 10.13.1	LLVM-GCC 4.2.1
	Ubuntu Workstation (ubuntu-deer)	Intel Core i7-3820 @ 3,60 GHz	16	Ubuntu 16.04	GCC 5.4.0
	Windows Workstation (windows-bison)	Intel Core i7-3820 @ 3,69 GHz	16	Windows 10 Enterprise	Cygwin GCC 6.4.0
Jednodoskový počítač	Raspberry Pi (raspberry-pi-3)	ARM Cortex A53 @ 1,20 GHz	1	Linux Raspberry Pi 4.9.35-v7	GCC 4.9.2
Tablety	Apple iPad Pro (ipad-pro)	Apple Fusion @ 2,36 GHz	4	OSX 11.0.3	-
	Samsung Tablet S3 (samsung-tab-s3)	Quad Code @ 1,6-2,15 GHz	4	Android 8.0.0	-
Smartphony	Samasung Galaxy S8 (samsung-s8)	Octa-core @ 1,70-2,30 GHz	4	Android 8.0.0	-
	Google Pixel 2 (pixel2)	Qualcomm Snapdragon 835 @ 1,90-2,35 GHz	4	Android 8.0.0	-
	Apple iPhone X (iphone10)	Apple Fusion @ 2,36 GHz	4	OSX 11.0.3	-

Tabuľka 1 Zariadenia použité v porovnaní WebAssembly výkonu [8]

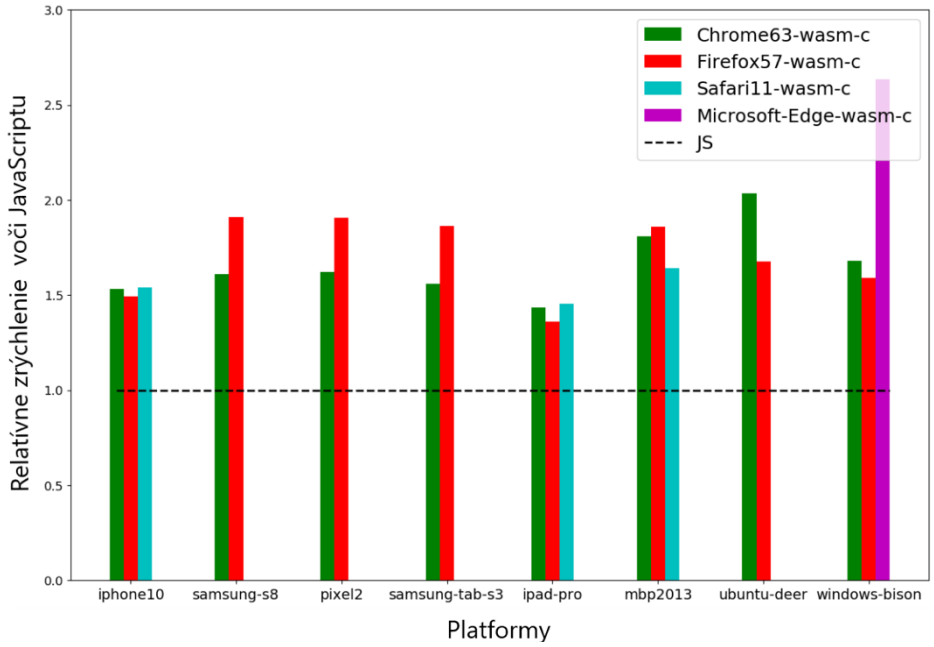
Na obrázku číslo 5 je zobrazené grafické porovnanie relatívneho zrýchlenia programov implementovaných vo WebAssembly oproti programom v jazyku C na rôznych notebookoch, pracovných staniciach a prehliadačoch. Prerušovaná čiara značí výkon programu v jazyku C. Hodnoty pod touto čiarou značia spomalenie oproti programu v jazyku C.

Väčšina prehliadačov vykonáva WebAssembly najmenej 0,75-násobnou rýchlosťou natívneho kódu v jazyku C. Výnimkou je prehliadač Safari vo verzií 11 na zariadení MacBook Pro 2013 [8].



Obrázok 5 Porovnanie výkonu WebAssembly oproti C na rôznych platformách [8]

Na obrázku číslo 6 je zobrazené grafické porovnanie relatívneho zrýchlenia programov napísaných vo WebAssembly oproti programom v jazyku JavaScript na všetkých zariadeniach a prehliadačoch, schopných vykonávať WebAssembly a JavaScript. Prerušovaná čiara značí výkon programu v jazyku JavaScript. Hodnoty pod touto čiarou značia spomalenie oproti programu v jazyku JavaScript.



Obrázok 6 Porovnanie výkonu WebAssembly oproti JavaScriptu na rôznych platformách [8]

Všetky prehliadače demonštrujú výrazné zvýšenie výkonu programov napísaných v jazyku WebAssembly. Zvýšenie výkonu oproti JavaScript programom je takmer dvojnásobné [8].

1.4 Použitie

WebAssembly je možné využiť na vývoj aplikácií alebo knižníc bežiacich vo webovom prehliadači, ale aj mimo neho. Neusporiadaný a neúplný zoznam aplikácií, domén a algoritmov, ktoré slúžili pri návrhu WebAssembly ako prípady použitia je [10]:

- Editovanie obrázkov a videa
- Počítačové videnie
- CAD aplikácie
- VPN
- Šifrovanie
- Úprava živého videa
- Virtuálna a rozšírená realita
- Vizualizácia a simulácia vedeckých procesov
- Hry

Analýze praktického využitia a účelu využitia WebAssembly vo webových stránkach sa venovala štúdia s názvom *New Kid on the Web: A Study on the Prevalence of WebAssembly in the Wild* [11] z roku 2019. Skúmanú vzorku tvorilo prvých milión stránok z Amazon rebríčka najobľúbenejších stránok. Z tejto vzorky bolo identifikovaných 1639 stránok, ktoré načítavali WebAssembly moduly a iba 506 z týchto stránok využívalo WebAssembly vo väčšej miere. Účely využitia boli klasifikované do nasledujúcich kategórií: Custom, Game (Hry), Libraries (knižnice), Mining (ťaženie kryptomien), Obfuscation a Test [11]. Prvé tri kategórie odpovedajú prípadom použitia, pre ktoré bol formát WebAssembly vytvorený. Kategórie Mining a Obfuscation obsahujú moduly, ktorých činnosť je pre užívateľa nežiadúca.

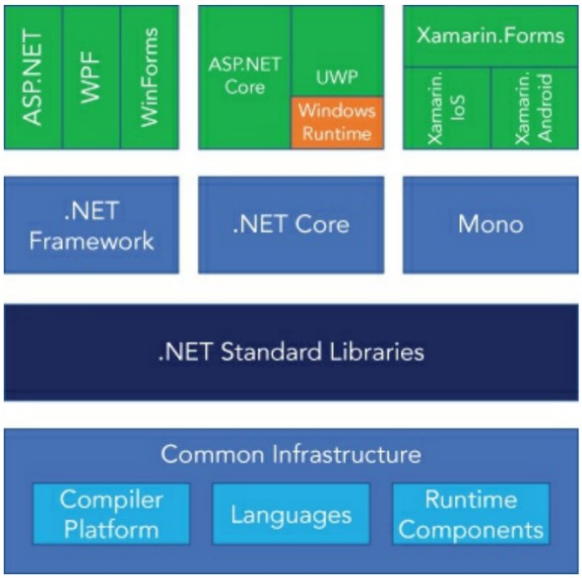
Kategória	Počet stránok	Počet stránok [%]
Custom	14	0,9
Game	58	3,5
Library	636	38,8
Mining	913	55,7
Obfuscation	4	0,2
Test	244	14,9
Unknown	5	0,3

Tabuľka 2 Rozpad modulov na webových stránkach podľa kategórie [11]

Z tabuľky číslo 2 je možné vidieť, že 917 (55,9%) navštívených stránok využívajúce WebAssembly načítavalo moduly z kategórií Mining a Obfuscation. Zo štúdie [11] je možné usúdiť, že WebAssembly si postupne nachádza uplatnenie vo svete moderných webových technológií, a že spolu s jeho výhodami prichádzajú taktiež nevýhody vo forme nových metód útoku na systém užívateľa.

2 .NET

.NET je bezplatná, multiplatformná, open source platforma pre vývojárov, určená pre tvorbu rozličných aplikácií od webov, desktop aplikácií až po IoT. Aplikácie .NET sú vyvíjané a spúšťané v rámci jednej alebo viacerých implementácií rozhrania .NET. Do implementácií rozhrania .NET patria .NET Framework, .NET Core a Mono [12]. Prehľad platformy je zobrazený na obrázku číslo 7.



Obrázok 7 Prehľad platformy .NET [5]

Aktuálne verzie jednotlivých implementácií .NET Framework, .NET Core a Mono sú v uvedenom poradí 4.8, 3.1 a 6.10.0 [13] [14] [15]. Týmito verziami končí vývojový cyklus implementácií .NET Framework a .NET Core, ktoré budú nahradené novou implementáciou .NET 5. Jej vydanie je naplánované na november roku 2020 [16].

Platforma nepredpisuje použitie konkrétneho programovacieho jazyka. Aplikácia sa bez ohľadu na to, v čom bola pôvodne implementovaná, preloží do jazyka Common Intermediate Language (CIL alebo IL). Programovacie jazyky pre vývoj .NET aplikácií sú C#, F# a Visual Basic .NET (VB.NET) [17].

V nasledujúcich podkapitolách budú v krátkosti opísané jednotlivé implementácie rozhrania .NET.

2.1 .Net framework

.NET Framework je najrozsiahlejší a najzrelší framework z pomedzi implementácií .NET. Jeho najväčšou nevýhodou je limitácia na platformu Microsoft Windows (desktop/server) [18]. .NET Framework sa skladá z dvoch hlavných častí, ktorými sú modul Common Language Runtime a Framework Class Library.

2.1.1 Modul CLR

Jadrom platformy .NET Framework je runtime prostredie, ktoré sa označuje ako modul Common Language Runtime (skrátene CLR) alebo runtime systém .NET. Kód spustení pod kontrolou tohoto modulu sa často označuje ako riadený kód (managed code) [19].

Pred spustením v modulu CLR musí byť ľubovoľný zdrojový kód preložený. Preklad v prostredí .NET prebieha v dvoch krokoch [19]:

1. Preklad zdrojového kódu do jazyka Intermediate Language (skrátene IL).
2. Preklad jazyka IL do kódu špecifického pre cieľovú platformu pomocou modulu CLR.

Jazyk IL je založený na rovnakej myšlienke ako bajtový kód jazyka Java. Jedná sa o nízkoúrovňový jazyk s jednoduchou syntaxou, ktorá nepracuje s textom, ale s číselnými kódmi. Jazyk je rýchlo preložiteľný do natívneho strojového kódu cielenej platformy pomocou Just-In-Time (skrátene JIT) prekladača [19].

CLR obsahuje typový systém (type system) spolu s typovým zavádzačom (type loader), ktorý je zodpovedný za načítanie typov zo zostavy, garbage collector, ktorý zabezpečuje uvoľňovanie zdrojov pamäte, ktoré nie sú ďalej referencované [20].

2.1.2 Framework Class Library

Framework Class Library (FCL) je kolekcia opakovane použiteľných typov, ktoré sa integrujú spolu s modulom CLR. Knižnica tried je objektovo orientovaná a poskytuje typy, z ktorých riadený kód odvodí funkčnosť [21].

2.2 .Net Core

.NET Core je bezplatne dostupný, open-source framework určený pre platformy ako Microsoft Windows, Linux a macOS. Je to multiplatformný nástupca .NET Framework-u vyvíjaný

primárne firmou Microsoft, ktorý je dostupný pod licenciou MIT[1]. .NET Core funguje na obdobnom princípe ako .NET Framework a skladá sa z dvoch častí CoreCLR a CoreFX, ktoré slúžia ako náhrady za CLR a FCL .NET Framework-u [22].

2.3 Mono

Mono je open source implementácia .NET Framework-u založená na štandarde ECMA[2] pre jazyk C# a Common Language Runtime [23].

Mono vzniklo za účelom poskytnutia možnosti spúšťať .NET aplikácie naprieč viacerými platformami a za účelom poskytnutia lepších vývojových nástrojov pre vývojárov na operačnom systéme Linux. Mono môže bežať na systémoch ako sú Android, macOS, BSD, Windows, Solaris, množstve Linux distribúcií a dokonca na hracích konzolách ako sú PlayStation 3 a Xbox 360 [24].

Mono pozostáva z nasledujúcich súčastí [25]:

- C# prekladač – Podporuje plnú funkcionalitu pre C# 1.0, 2.0, 3.0, 4.0, 5.0 a 6.0 (ECMA).
- Mono runtime – Nahrádza funkcionalitu modulov CLR, CoreCLR frameworkov .NET a .NET Core. Runtime disponuje Just-In-Time (JIT) a Ahead-of-Time (AOT) prekladačmi, zavádzačom knižníc, garbage collectorom a systémom na správu vlákien.
- .NET Framework Class Library – Platforma Mono obsahuje sadu tried kompatibilnú s Microsoft .NET FLC.
- Mono Class Library – Mono poskytuje sadu tried nad rámec Base Class Library (BCL, jedná sa o základnú podmnožinu FCL), ktorú poskytuje firma Microsoft. Táto súprava tried nachádza uplatnenie pri vývoji aplikácií na platformách iných ako Microsoft Windows.

[1] MIT licencia je slobodná softvérová licencia, pod ktorou užívateľ získava neobmedzené právo kopírovať, používať, modifikovať program za predpokladu uvedenie autora a informácií o licencií.
[2] Štandard ECMA-334 označuje špecifikáciu jazyka C#, štandard ECMA-335 označuje špecifikáciu CLI

2.3.1 Mono WASM

Zdrojový kód .NET implementácie Mono je napísaný v jazyku C++. Aplikácie a programy napísané v tomto jazyku je možné kompilovať do WebAssembly pomocou Emscripten. Z toho dôvodu vznikla úspešná iniciatíva vývojového tímu Mono preložiť Mono do WebAssembly. Existujú dva prístupy prekladu. Prvým prístupom je kompilovanie .NET kódu programu spolu s Mono runtimom do jednej veľkej WebAssembly aplikácie. Z pohľadu časovej náročnosti je tento prístup nepraktický, pretože všetky zdrojové kódy musia byť preložené do WebAssembly. Druhým prístupom je skompilovať do WebAssembly iba Mono runtime, ktorý následne beží vo webovom prehliadači a je schopný vykonávať IL tak ako .NET bežiaci mimo prehliadač. Veľkou výhodou je skutočnosť, že výsledok z .NET prekladu nemusí byť preložený do WebAssembly. Tento prístup je využitý aj v technológií Blazor [26].

3 BLAZOR

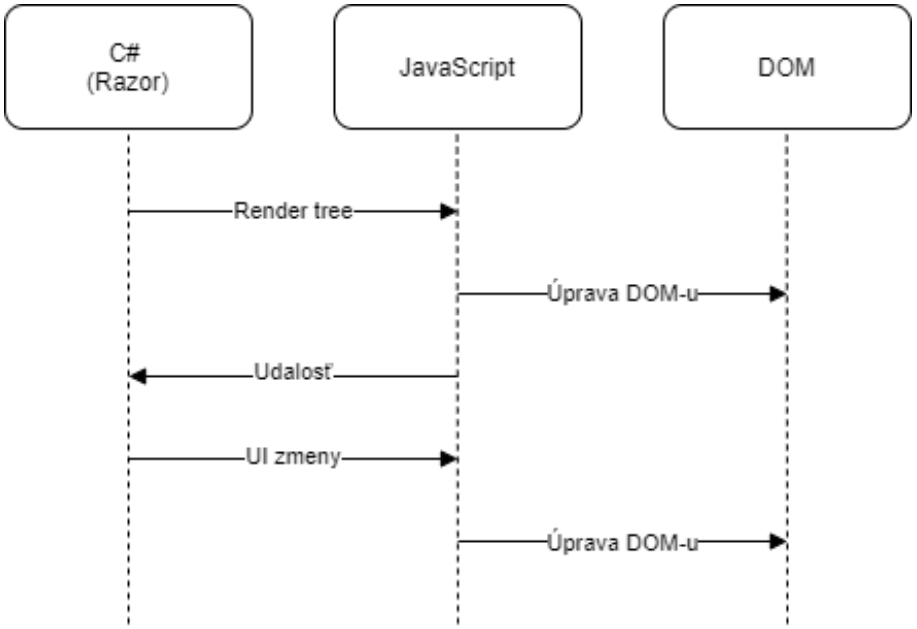
Blazor je open source, multiplatformový UI framework vyvíjaný firmou Microsoft určený pre vytváranie webových single-page aplikácií (SPA) pomocou jazyka C# [27]. Framework je založený na znovupoužiteľných blokoch kódu (tzv. komponenty), ktoré sú implementované v jazykoch C# a Razor. Tieto komponenty majú za cieľ zapúzdrovať frekventovane používané UI elementy a tým znížiť duplicitu kódu a zvýšiť jeho čitateľnosť.

Framework podporuje dva modely hostingu. Prvým podporovaným modelom je Blazor WebAssembly, bežiaci na strane klienta v prehliadači pomocou technológie WebAssembly. Druhý model, menom Blazor Server, beží na strane ASP.NET Core servera. Vytvorené komponenty UI frameworku sú nezávislé na vybranom modeli hostingu a preto môžu byť použité v oboch modeloch [28].

3.1 Blazor WebAssembly

Blazor WebAssembly bol oficiálne vydaný v máji roka 2020 [29]. Tento model beží pomocou WebAssembly vo webom prehliadači klienta, do ktorého sa stiahne Blazor aplikácia spolu s .NET runtimom. Aplikácia beží priamo na UI vlákne prehliadača. Prostriedky aplikácie, ako napríklad css súbory, sú nasadené na webový server alebo služby, ktoré sú schopné poskytovať tento statický obsah klientom, ako statické súbory [28].

WebAssembly nedokáže priamo manipulovať DOM webovej stránky. Preto bol implementovaný proces, pri ktorom sa výsledok prekladu Razor súborov posiela do Blazor enginu, ktorý vytvára stromovú štruktúru na vykreslenie (render tree). Táto stromová štruktúra je následne spracovaná pomocou JavaScript kódu, ktorý na jej základe upravuje DOM (vytvára, upravuje a vymazáva HTML elementy a atribúty). Celý opísaný proces prebieha vo webovom prehliadači užívateľa [26].

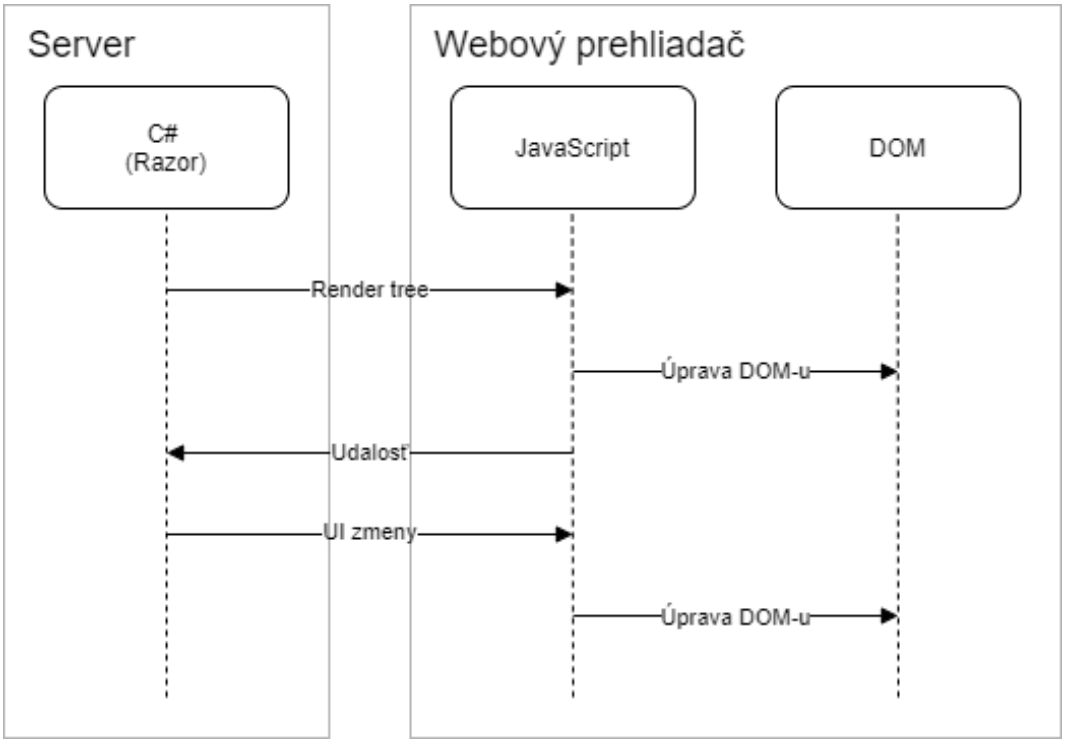


Obrázok 8 Proces manipulácie DOM-u pomocou WebAssembly Blazoru [26]

Výhodou Blazor WebAssembly je nezávislosť na servery po kompletnom stiahnutí aplikácie. To poskytuje užívateľovi možnosť pracovať s načítanou stránkou aj bez pripojenia. Ďalšou výhodou je presunutie záťaže zo servera na klienta. Medzí hlavné nevýhody patrí to, že model hostingu je limitovaný možnosťami webového prehliadača, ktorý musí podporovať WebAssembly. K nevýhodám taktiež patrí aj celková veľkosť aplikácie, ktorá je väčšia a jej stiahnutie trvá dlhšiu dobu [28].

3.2 Blazor Server

Proces manipulácie DOM-u webovej stránky je obdobný ako pri modeli WebAssembly. Na strane servera sú preložené Razor súbory a následne sa vytvára stromová štruktúra na vykreslenie. Štruktúra je serializovaná a následne poslaná do webového prehliadača pomocou spojenia SignalR, kde je deserializovaná a spracovaná prostredníctvom JavaScript kódu. Udalosti vyvolané vo webovom prehliadači sú serializované a zaslané na server, kde sú spracované pomocou .NET kódu, ktorý aktualizuje stromovú štruktúru na vykresľovanie a opätovne ju posiala do webového prehliadača [26].



Obrázok 9 Proces manipulácie DOM-u pomocou Server Blazoru [24]

Medzi výhody tohto modelu patria menšia veľkosť aplikácia na stiahnutie, prehliadače nemusia podporovať WebAssembly a zdrojový kód nie je posielaný do webového prehliadača klienta. Medzi nevýhody patria zvýšená latencia, zlé škálovanie aplikácie a nemožnosť práce s načítanou stránkou po strate pripojenia [28].

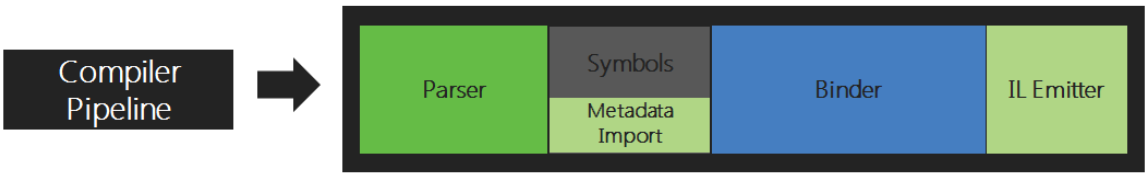
4 ROSLYN

.NET Compiler Platform, známy aj pod prezývkou Roslyn, je open source, modulárny prekladač určený pre jazyky C#, F# a Visual Basic .Net od spoločnosti Microsoft [30]. Zdrojový kód prekladača písaný v jazyku C# je verejne dostupný na adrese [31] pod licenciou Apache 2 [18]. Roslyn obsahuje sadu prekladačov pre jednotlivé podporované jazyky.

Roslyn sa líši od tradičných prekladačov otvorením procesu prekladu. Tradične proces prekladu spracoval kód na vstupe a na výstupe poskytoval výsledok z prekladu. Roslyn proces prekladu otvoril prostredníctvom rozhraní a servís, ktoré poskytujú informácie o spracovávanom kóde v jednotlivých krokoch prekladu. Využitím týchto rozhraní a servís bola dramaticky znížená náročnosť tvorenia nástrojov určených na analýzu, opravu kódu alebo aj na generovanie kódu [32].

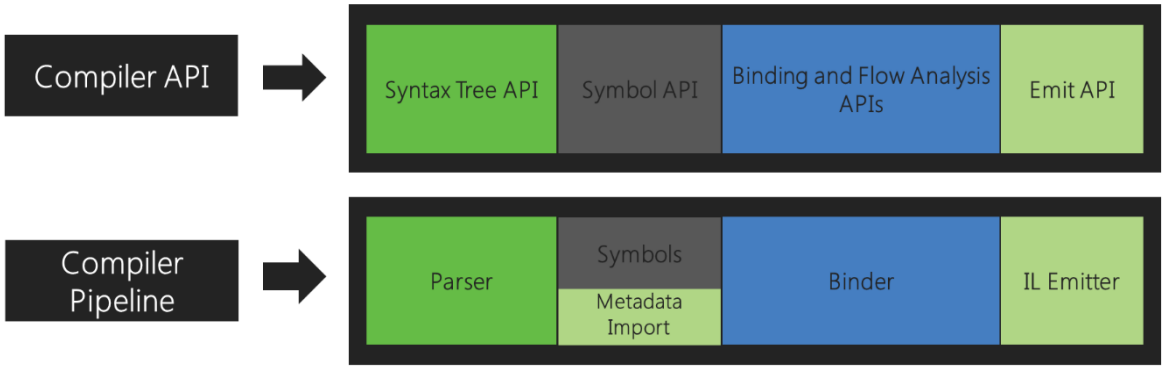
4.1 Pipeline prekladača

Roslyn sprostredkováva analýzu kódu vykonanú C# a VB prekladačmi pomocou vrstvy API, ktorá odzrkadľuje tradičnú pipeline prekladača [33].



Obrázok 10 Tradičná pipeline prekladača [33]

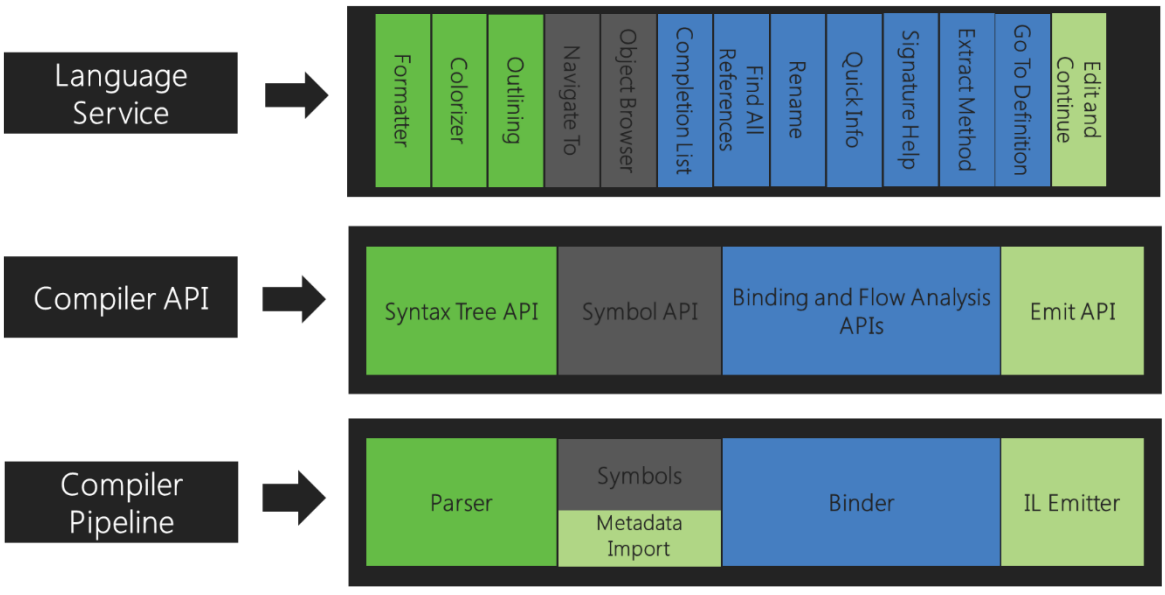
Každá etapa prekladu z pipeline je implementovaná ako samostatný komponent. Jednotlivé komponenty sú zobrazené na obrázku číslo 11. Prvou etapou je etapa analýzy (parser), kde je zdroj analyzovaný a transformovaný na syntax, ktorá splňuje jazykovú gramatiku. Následne sa v etape deklarácie zo zdroja analyzujú deklarácie a importované metadáta za účelom vytvorenia pomenovaných symbolov. Tretia etapa (binder) priraďuje identifikátory v kóde ku symbolom. Posledná etapa (emitter) spracováva všetky informácie o vstupnom kóde do výstupnej zostavy [33].



Obrázok 11 Porovnanie Roslyn API ku tradičnej pipeline prekladača [33]

Všetky etapy poskytujú prístup k ich informáciám pomocou objektových modelov. Parse etapa poskytuje model syntax tree, declaration etapa hierarchical symbol table, bind etapa model poskytujúci výsledok zo sémantickej analýzy a emit etapa API, ktorá vytvára IL byte kódy [33].

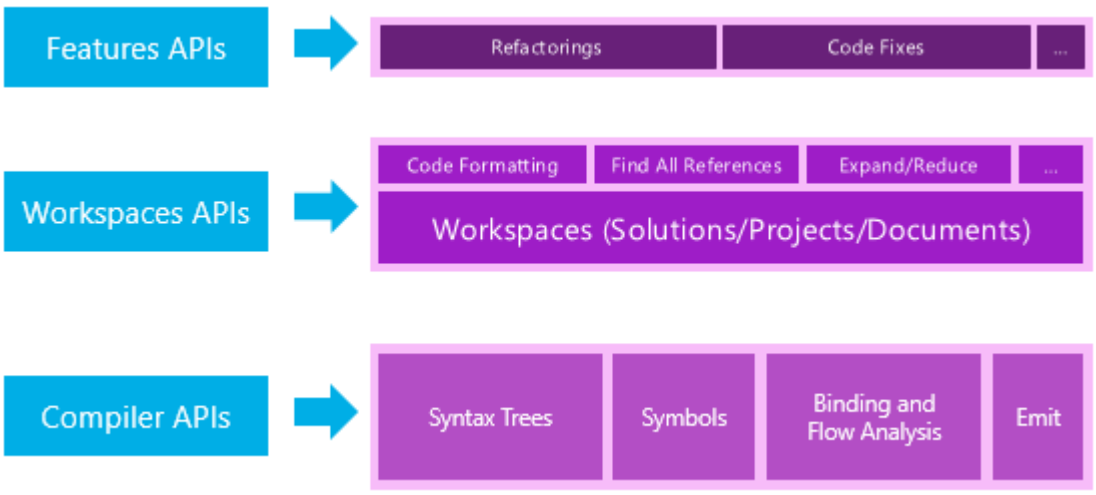
Na overenie dostatočnej robustnosti verejnej API prekladača na vytváranie nástrojov pre integrované vývojové prostredia, bola vytvorená jazyková servisa (language service), ktorá je použitá vo vývojom prostredí Visual Studio pre jazyky C# a VB. Príkladom týchto nástrojov sú napríklad zvýrazňovač kódu (Outlining) a formátovanie textu (Formatter), ktoré využívajú syntax tree, okno prehliadača objektov (Object Browser) a navigačné doplnky (Navigate To), ktoré využívajú symbol table [33].



Obrázok 12 Jazyková servisa využívajúca API prekladača [33]

4.2 Vrstvy API

Roslyn sa skladá z troch vrstiev API. Jednou z nich je vyššie opísaná Compiler API. Zvyšné dve vrstvy sú Workspaces a Features API. Vrstvy API môžeme vidieť na obrázku číslo 13.



Obrázok 13 API vrstvy prekladača Roslyn [33]

4.2.1 Compiler API

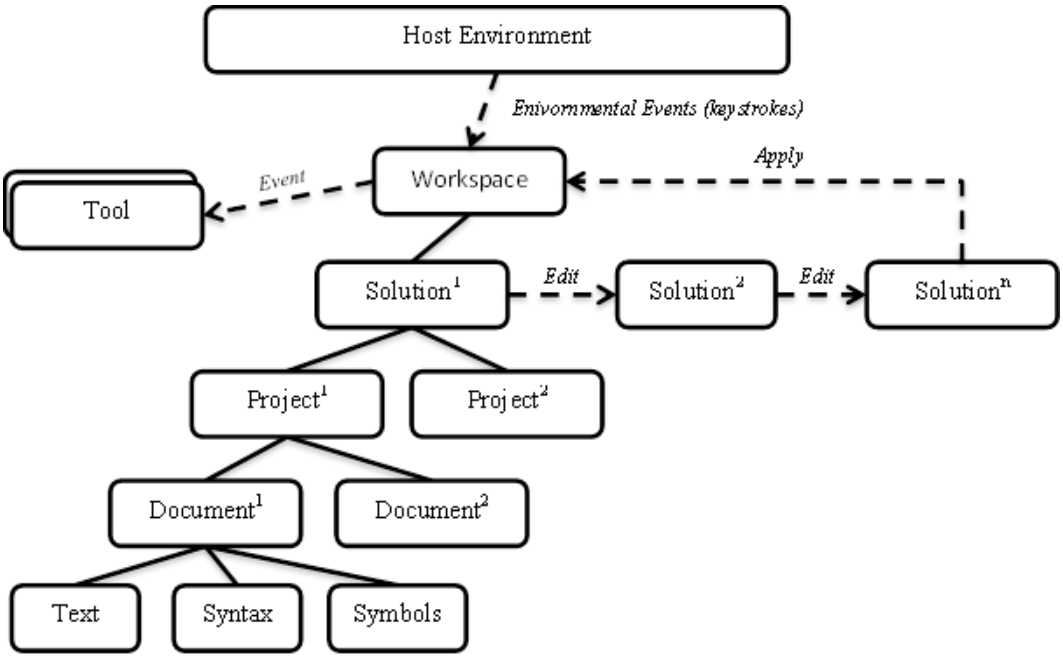
Vrstva obsahuje objektový model, korešpondujúci so syntaktickými a sémantickými informáciami, ktoré poskytujú jednotlivé etapy pipeline prekladača. Vrstva taktiež obsahuje nemenný (immutable) snapshot invokácie prekladača spolu s referenciami na zostavy, nastavením prekladu a súbormi zdrojového kódu. Pre jazyky C# a VB sú vytvorené separátne API na dosiahnutie maximálnej prispôsobivosti prekladačov pre jednotlivý jazyk [33].

4.2.2 Workspace API

Obsahuje základné funkcionality na analýzu kódu a refaktorovanie kódu nad celým .NET riešením. API asistuje pri organizovaní všetkých informácií o projektoch a riešení do jedného objektového modelu. [33]. Poskytuje prístup do projektov, zdrojových kódov a metadát pomocou objektov z Compiler API [34].

4.2.2.1 Workspace

Workspace API poskytuje pracovné prostredie (workspace) predstavujúce aktívnu reprezentáciu .NET riešenia. Reprezentácia je vo forme zbierky projektov, kde každý projekt obsahuje vlastnú zbierku dokumentov. Pracovné prostredia sú bežne načítané pomocou vývojových prostredí, v ktorých zmena vyvolá zmenu pracovného prostredia tak, aby bola reprezentácia neustále aktuálna. Prostredia je možné načítať programovo aj mimo vývojové prostredia [33].



Obrázok 14 Grafické znázornenie pracovného prostredia [33]

5 TRY .NET

Try .NET je open source nástroj vyvíjaný firmou Microsoft. Jeden z primárnych účelov tohoto nástroja je pomocou interaktívnej dokumentácie zoznámiť širšiu verejnosť s programovacím jazykom C# a platformou .NET. Dokumentácia obsahujúca spustiteľný kód má pre nováčikov výrazne znížiť bariéru vstupu do technológií .NET, odstránením nutnosti inštalovania akýchkoľvek softvérov na zobrazovacie zariadenie. Užívatelia nie sú nútený inštalovať a konfigurovať vývojové prostredia. Tým pádom je nástroj vhodný na vzdelávacie účely ako sú napríklad workshopy alebo prednášky.

Zdrojový kód nástroja je šírený pod licenciou MIT a je distribuovaný pomocou verejne dostupného repozitára na adrese [35]. Z kódu a dostupných informácií v uvedenom repozitári bolo zistené, že nástroj je vytvorený tak, aby bol schopný pracovať v dvoch módoch. Jednotlivé módy nemajú momentálne oficiálne mená, dostupné sú iba popisy módov v [35], ktoré sú:

1. Webové prostredie využívajúce Blazor
2. Interaktívna dokumentácia využívajúca značkovací jazyk Markdown a .NET Core

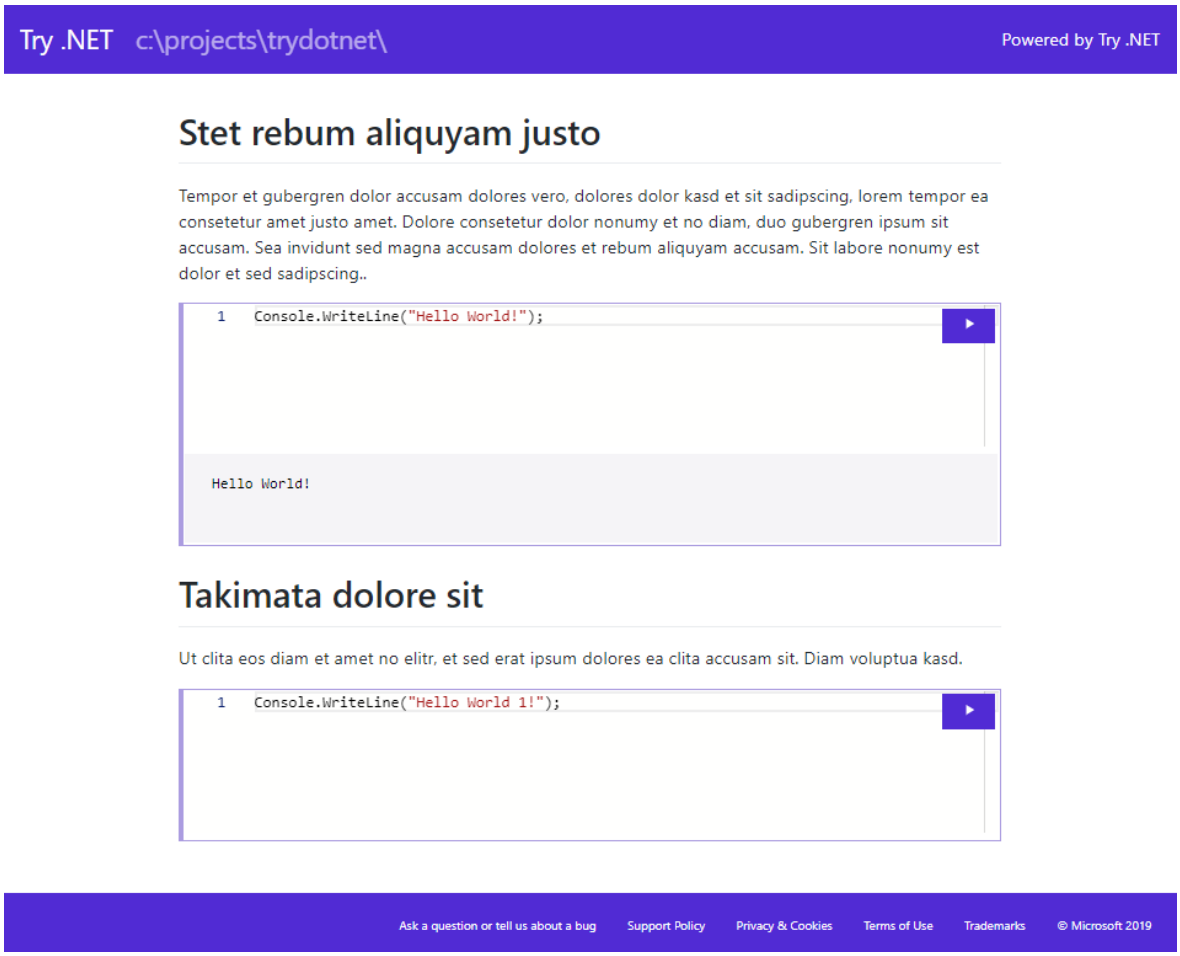
Módy sa implementáciou, využitými technológiami a hlavne účelom značne líšia a v záujme prehľadnosti ich opisu budú pomenované ako **Try** (webové prostredie) a **Hosted** (dokumentácia). Pomenovania sú prevzaté z konfiguračných enumerácií zo zdrojového kódu nástroja.

Mód Hosted slúži na tvorbu interaktívnych dokumentácií za pomoci jazyka C# a značkovacieho jazyka Markdown. V dokumentácií má užívateľ možnosť pomocou textového editora spúšťať C# kód a skúmať jeho výstup. Naopak mód Try má poskytovať užívateľom možnosť integrovať textový editor so spustiteľným kódom do užívateľských stránok. Dualita účelu a nami opísané módy neboli zo strany Microsoftu jasne komunikované. Z tohto dôvodu vznikali dotazy ako sú napríklad tieto [36], [37] a [38].

Nasledujúce kapitoly sa budú venovať módom Try a Hosted. Bude opísané ako fungujú, z akých prvkov sa skladajú, čím sa líšia a čo majú spoločné.

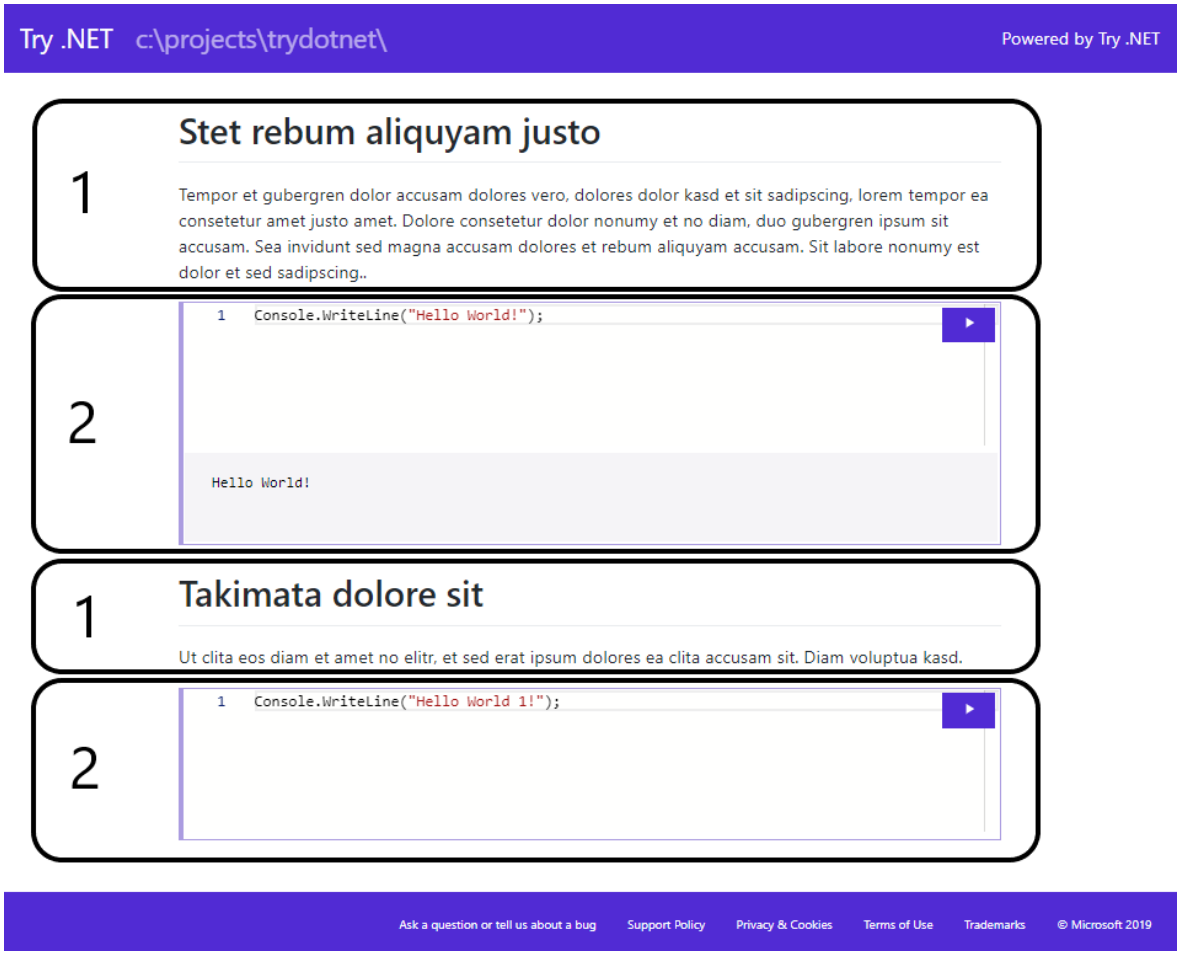
5.1 Hosted

Je mód Try .NET-u slúžiaci na vytváranie interaktívnej dokumentácie pomocou nástroja menom *dotnet try global*. Dokumentácia tvorená týmto nástrojom sa skladá po vizuálnej stránke z dvoch základných elementov, ktoré sa môžu v dokumente neobmedzene opakovať. Tieto elementy sú text, vytvorený pomocou značkovacieho jazyka Markdown a editor spustiteľného kódu spolu s výstupom spusteného kódu. Ukážka dokumentácie vytvorenej pomocou tohto módu je zobrazená na obrázku číslo 15.



Obrázok 15 Ukážka interaktívnej dokumentácie Try .NET módu Hosted

Dokumentácia sa skladá z vyššie uvedených elementov, vyznačených číselne na nasledujúcom obrázku. V sekciách označených číslom je text tvorený pomocou značkovacieho jazyka Markdown. Ďalej v sekciách označených číslom 2 sú editory spustiteľného kódu spolu s výstupom spusteného kódu. V prvej sekcií číslo 2 je zobrazený výsledok spustenia kódu editora *„Hello World!“*.



Obrázok 16 Vyznačené elementy interaktívnej dokumentácie Try .NET módu Hosted

Pred opisom nástroja *dotnet try global* a procesu jeho činnosti je potreba opísať súbory potrebné na vytvorenie interaktívnej dokumentácie. Tieto súbory sú:

- Markdown súbory
- Visual Studio .NET C# projekt

Markdown súbory definujú obsah a formát dokumentácie. Tieto súbory sú nástrojom spracované do HTML formátu pomocou Markdown procesoru. Za účelom vykreslenia editora spustiteľného kódu a jeho výstupu do dokumentácie bola rozšírená syntax jazyka Markdown. Pomocou tohto rozšírenia je možné zakomponovať do HTML výstupu editor monaco.

Rozšírenie spočíva v pridaní parametrov k ohraničenému bloku kódu (fenced code blocks), ktorý je následne nahradení vykresleným editorom. Oblasť ohraničeného bloku kódu začína a končí kľúčovými slovami ```, za ktorými musí byť určený jazyk C# (cs, csharp, c#). Za deklaráciou bloku nasledujú parametre identifikujúce kód, ktorý má byť načítaný do editora v dokumentácií z VS .NET C# projektu. Rozšírená syntax ohraničeného bloku kódu je:

```
```cs --source-file <path to source file> --project <path to .csproj file> --region <region>
```
```

Kde význam jednotlivých parametrov je nasledujúci [24]:

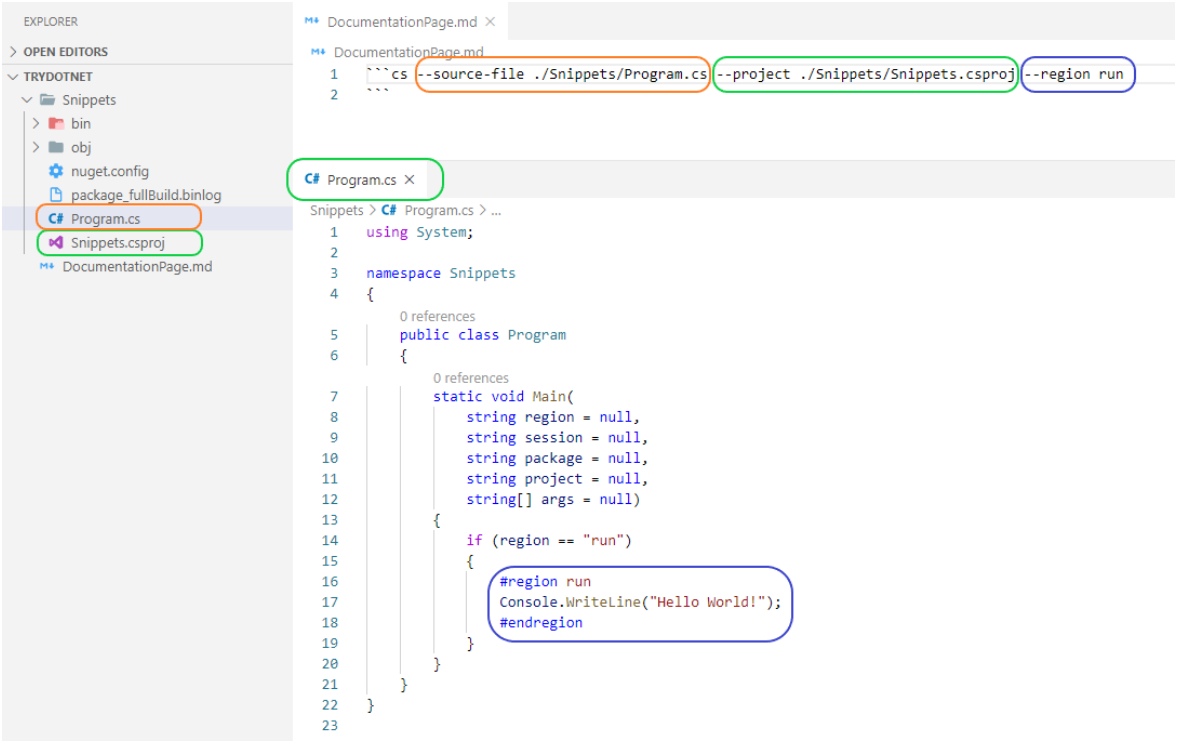
- --project <path> – relatívna cesta k .csproj súboru VS .NET C# projektu
- --source-file <path> – relatívna cesta k súboru so zdrojovým kódom
- --region <name> – identifikuje región v zdrojovom kóde, ktorého kód bude načítaný do editora v dokumentácií
- --hidden – editor bude v dokumentácií skrytý
- --editable <bool> - v prípade hodnoty *false* bude editor iba na čítanie, inak bude editovateľný (východzia hodnota je *true*)
- --destination-file <path> - relatívna ceste k súboru, do ktorého bude vložený obsah súboru (východzia hodnota je rovná hodnote parametra source-file)
- --session <name> - slúži na oddelenia prekladu jednotlivých blokov, všetky session s rovnakým menom sa prekladajú súčasne

Príklad ohraničeného bloku kódu s rozšírenými parametrami:

```
```cs --source-file ./Snippets/Program.cs --project ./Snippets/Snippets.csproj --region run
```
```

Obrázok 17 Príklad ohraničeného bloku kódu s rozšírenými parametrami

Parametre v príklade ohraničeného bloku kódu na obrázku číslo 17 určujú obsah editora, ktorý bude načítaný z C# regiónu *run* a zo súboru *Program.cs*, ktorý sa nachádza v projekte *Snippets.csproj*. Ukážková štruktúra spolu s Markdown súbormi a VS .NET C# projektom je zobrazená na obrázku číslo 18. Na obrázku sú farebne vyznačené parametre a objekty, ktoré jednotlivé parametre identifikujú.



Obrázok 18 Ukážka štruktúry súborov a ich obsah pre mód Hosted

Výsledok po spustení nástroja *dotnet try global* je dokumentácia o jednej stránke s editorom, ktorý je naplnený spustiteľným kódom *Console.WriteLine("Hello World!");*.

Obrázok 19 Výsledná dokumentácia z ukážky štruktúry súborov a ich obsahu

VS .NET C# projekt neslúži iba na definovanie obsahu editorov, ale aj na samotný preklad a spustenie kódu užívateľa v dokumentácií. Celý VS .NET C# projekt identifikovaný pomocou parametrov bloku ohraničeného kódu sa preloží a následne spustí s podmnožinou parametrov z bloku. Pomocou vstupných parametrov je možné spúšťať kód, ktorý patrí iba spus-

tenému editoru. Na obrázku číslo 18 je zobrazené využitie parametru *–region run* z textového súboru Markdown. Metóda *Main* je spustená s parametrom *region*, ktorý je rovný hodnote „run“. Na základe tohoto parametra je vykonaný patričný kód. Pokiaľ v textovom dokumente Markdown nie je určený parameter región, tak sa v editore zobrazuje celý obsah identifikovaného súboru z VS .NET C# projektu.

Pri každom spustení kódu užívateľom v dokumentácií je vytvorený request, ktorý je poslaný na endpoint API */workspace/run*. Request obsahuje celý obsah spusteného editora na to, aby jeho obsah nahradil originálny kód na blokom identifikované miesto v VS .NET C# projektu. Ďalším zaujímavým polom requestu je *runArgs*. Tieto argumenty sú neskôr transformované na vstupné parametre pre metódy v projekte (napríklad parameter *region* na obrázku číslo 18).

```
{
 "runArgs": "--source-file ./Snippets/Program.cs --project ./Snippets/Snippets.csproj --region run",
 "workspace": {
  "workspaceType": "C:\\Projects\\TryDotNet\\Snippets\\Snippets.csproj",
  "language": "csharp",
  "files": [],
  "buffers": [
   {
    "id": "C:\\Projects\\TryDotNet\\Snippets\\Program.cs@run",
    "content": "Console.WriteLine(\"Hello from changed World!\");",
    "position": 0
   }
  ]
 },
 "activeBufferId": "C:\\Projects\\TryDotNet\\Snippets\\Program.cs@run",
 "requestId": "trydotnet.client_1"
}
```

Ukážka kódu 1 Request o preklad a spustenie kódu módu Hosted

API prijatý request spracuje, nahradí originálny kód užívateľským kódom, preloží celý VS .NET C# projekt pomocou prekladača Roslyn a následne preložený projekt spustí s parametrami *runArgs*. Výsledok programu je zaslaný späť v rámci response na request.

```
{
  "requestId": "trydotnet.client_1",
  "succeeded": true,
  "output": ["Hello from changed World!", ""],
  "exception": null,
  "diagnostics": []
}
```


Ukážka kódu 2 Response prekladu a spustenia kódu módu Hosted

Výsledok spustenia užívateľského kódu (pole *output* z ukážky kódu číslo 2) je následne spracovaný a zobrazený v dokumentácii.

Obrázok 20 Výsledok spustenia upraveného kódu užívateľom

5.1.1 Nástroj dotnet try global

Dotnet try global je program spúšťaný pomocou príkazového riadka. Príkaz spúšťajúci nástroj má nasledovný tvar:

```
dotnet-try [options] [<RootDirectory>] [command]
```

Parametre príkazu sú:

- options:
  - --add-package <source> - určuje dodatočné zdroje NuGet balíčkov
  - --package <name> - určuje Try .NET balíčky alebo cestu k .NET projektu
  - --package-version <version> - určuje verziu Try .NET balíčkov
  - --uri <uri> - určuje URL alebo relatívnu cestu k Markdown súborom
  - --enable-preview-features – povoľuje preview funkcie
  - --log-path <dir> - povoľuje logovanie do špecifikovaného súboru
  - --verbose – povoľuje obsiahle logovanie do konzoly
  - --port <port> - určuje port aplikácie
  - --version – zobrazuje verziu nástroja
  - -?, -h, --help – zobrazuje pomocné informácie
- Argumenty:
  - <RootDirectory> - určuje cestu k potrebným súborom dokumentácie
- commands:
  - demo – vytvorenie demo projektu na demonštráciu Try .NET-u
  - verify <rootDirectory> - overenie správnosti dokumentácie

V rámci tejto práce bude nižšie opísaný základný príkaz *dotnet-try*, ktorým je vytváraná dokumentácia z obsahu aktuálneho priečinka, z ktorého je príkaz spúšťaný.

V prvom kroku po spustení program vykoná rozbor vstupných argumentov, na základe ktorých inicializuje servisy a konfiguračné premenné potrebné na vstupom definovanú činnosť. Ďalším krokom programu pri vykonávaní opisovaného príkazu *dotnet-try* je spustenie webového servera Kestrel, ktorý počúva na voľne dostupnom porte. Spustenie webového serveru je zobrazené na obrázku číslo 21, kde server počúva na porte číslo 64747.

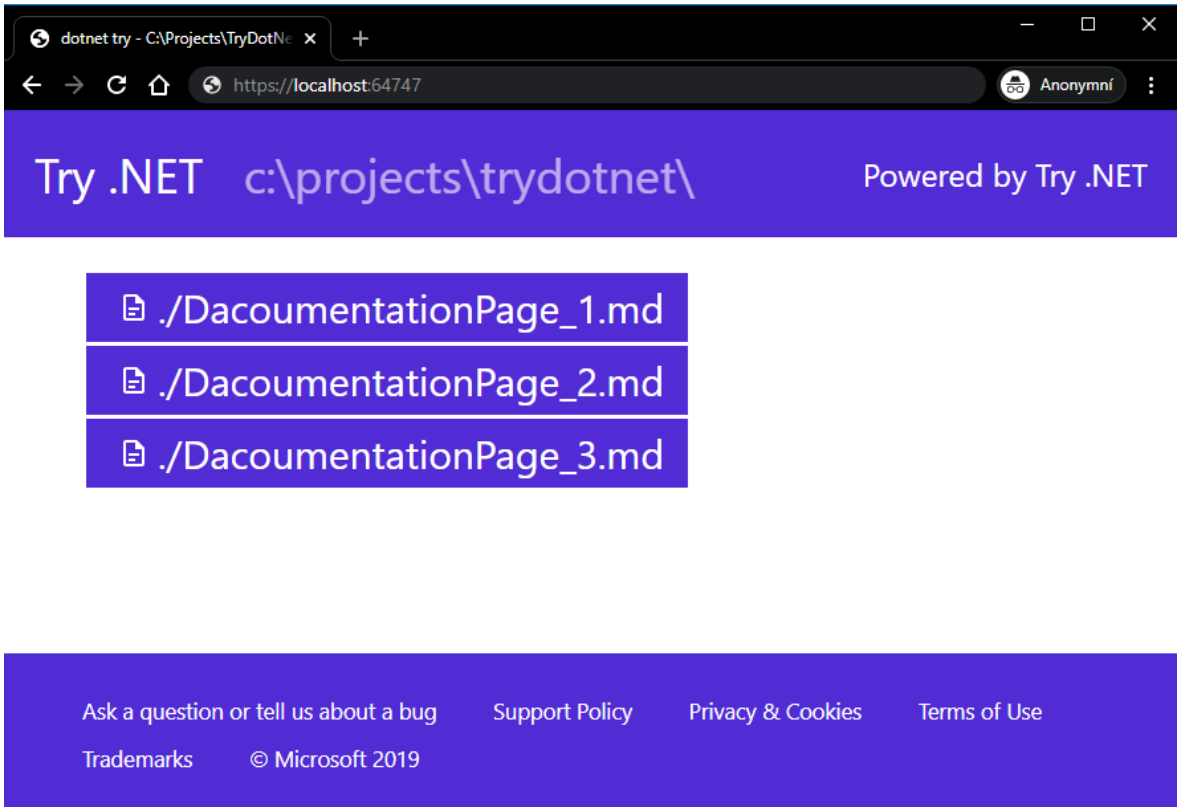
```
netstat -aon | findstr '64747'
  TCP    127.0.0.1:64747        0.0.0.0:0              LISTENING       17840
  TCP    [::1]:64747            [::]:0                 LISTENING       17840

tasklist | findstr '17840'
dotnet-try.exe                17840 Console                    1     48,128 K
```

Obrázok 21 Webový server počúvajúci na porte 64747, spustený nástrojom *dotnet try global*

Webový server hostuje webovú aplikáciu, ktorá po spustení automaticky otvorí novú kartu vo webom prehliadači s domovskou adresou webovej aplikácie (https://localhost:64747/). Obsah domovskej stránky je načítaný z kontroléra *DocumentationController* z endpointu na ceste „/“ (index). Tento endpoint spracováva metóda *ShowIndex*, ktorá dynamicky vytvorí obsah HTML stránky. Stránka zobrazuje mená všetkých dostupných stránok dokumentácie (jeden Markdown súbor odpovedá jednej stránke dokumentácie). Meno stránky dokumentácie odpovedá menu Markdown súboru.

Na obrázku číslo 22 nižšie je zobrazená domovská stránka dokumentácie, skladajúca sa z 3 stránok. Každá zo stránok má názov odpovedajúci názvu súboru, z ktorého bola vygenerovaná.

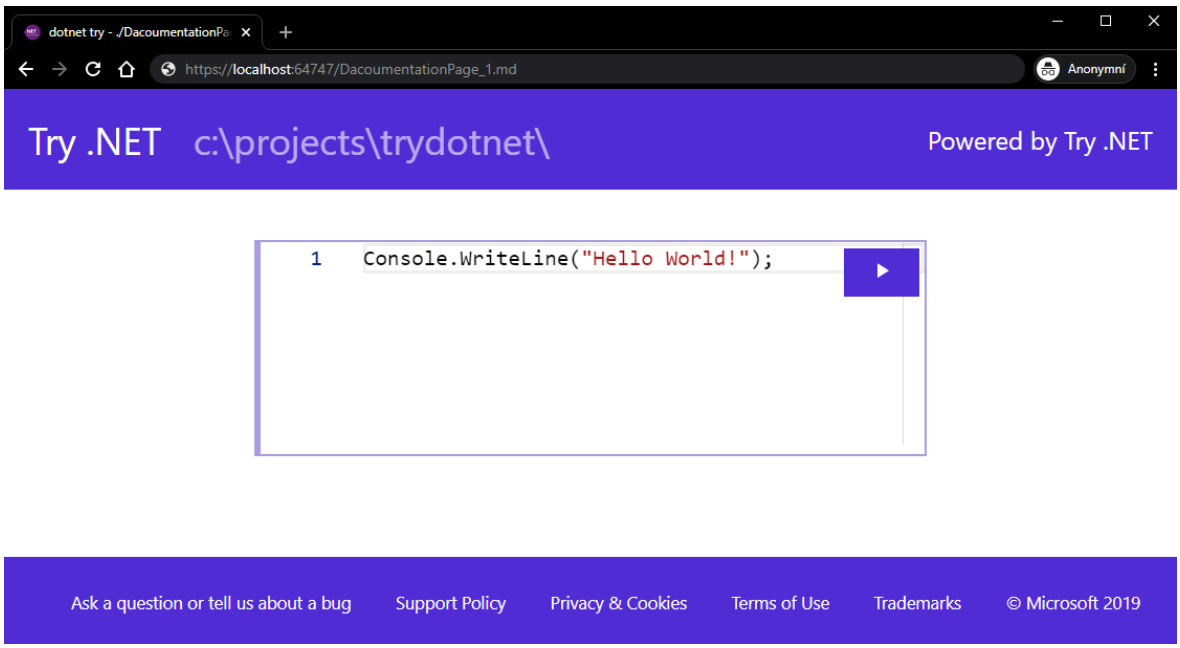


Obrázok 22 Domovská stránka dokumentácie módu Hosted

Po výbere konkrétnej stránky je užívateľ presmerovaný na webovú stránku s adresou vo formáte:

```
{host}//{scheme}:{port}//{docPageName}.md
```

Obsah tejto stránky je opäť načítaný pomocou kontroléra *DocumentationController*. V tomto prípade z metódy *ShowMarkdownFile*. Táto metóda načíta požadovaný Markdown súbor. Vyhľadá a spracuje v ňom ohraničené bloky kódu a pred odoslaním obsahu stránky spracuje[3] obsah Markdown súboru do HTML výstupu, ktorý následne posiela. Vykreslený obsah stránky dokumentácie je zobrazený na obrázku číslo 23.



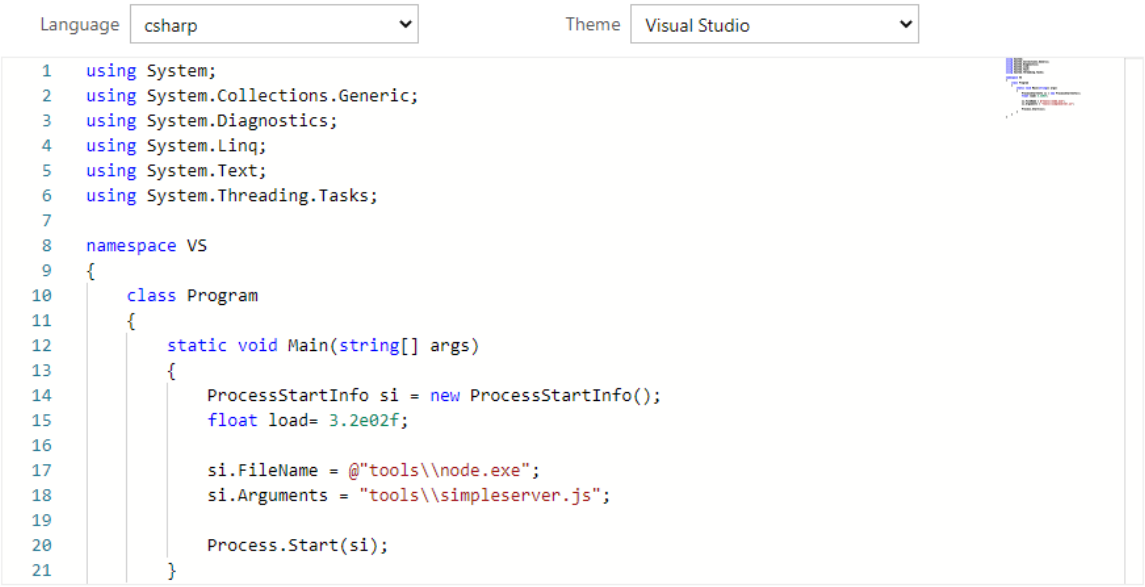
Obrázok 23 Stránka dokumentácie módu Hosted

Po vykreslení stránky dokumentácie je užívateľ schopný editovať a spúšťať kód procesmi, ktoré boli opísané na začiatku tejto kapitoly.

[3] Konverzia Markdown súboru na HTML je vykonaná pomocou Markdown procesoru pre .NET s názvom Markdig.

5.1.2 Editor monaco

Na editáciu zdrojového kódu v dokumentácií je použitý editor monaco, ktorý je integrovateľný do webového prehliadača. Vzhľad editora je zobrazený na obrázku nižšie.



Obrázok 24 Ukážka vzhľadu editora monaco [39]

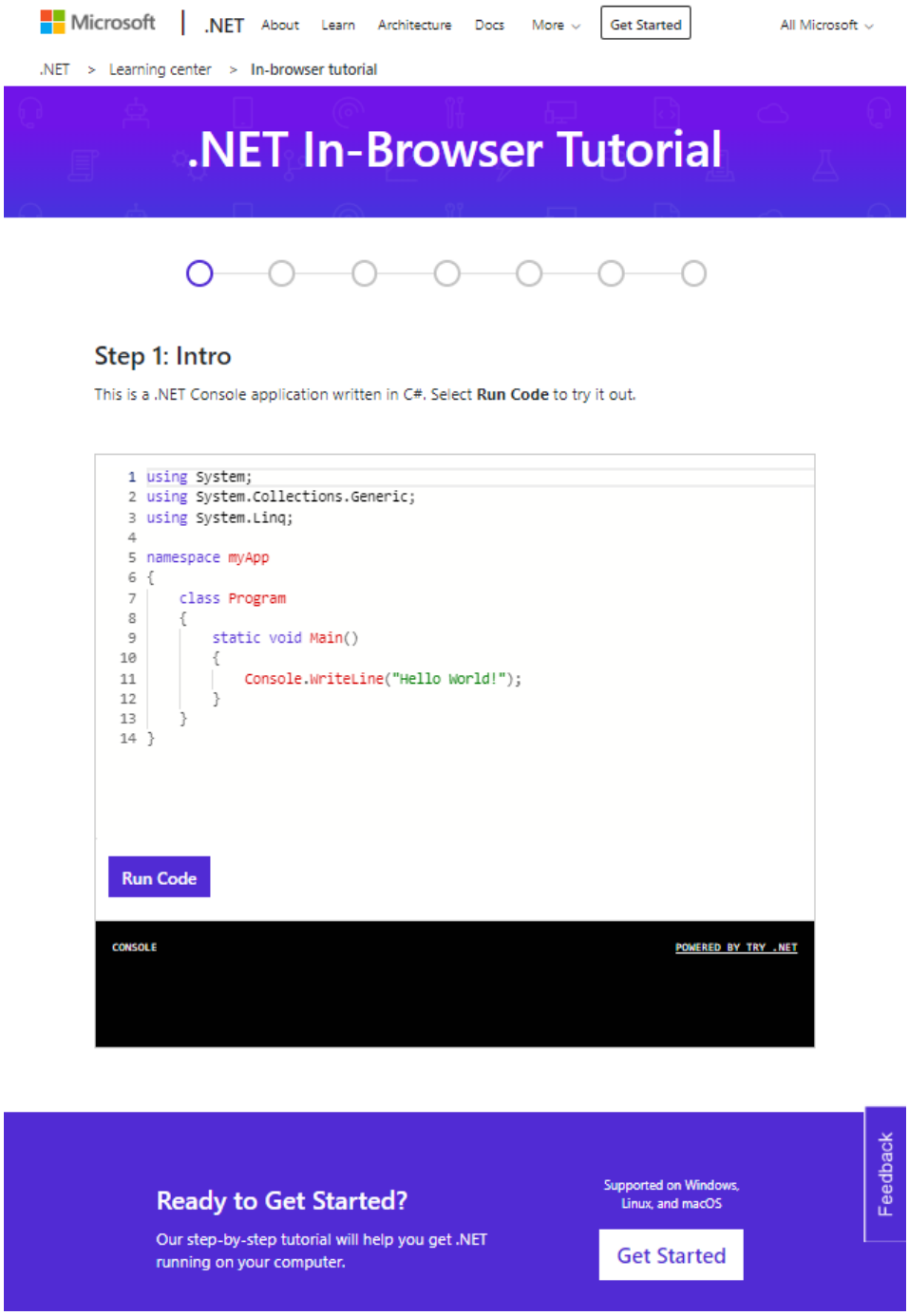
Zdrojový kód editora je priamo generovaný zo zdrojových kódov editora, ktorý je použitý vo vývojovom prostredí Visual Studio Code (VS Code). Generovanie kódu pre editor bežiaci vo webovom prehliadači z kódu editora určeného pre desktop je možný pomocou frameworku Electron[4], ktorý bol použitý na vývoj VS Code [40].

Vygenerovaný kód editora nie je priamo pripravený na použitie. Pred jeho nasadením do webového prostredia musí byť upravený doplnením web workerov, ktoré vykonávajú časovo náročné operácie mimo hlavné vlákno prehliadača pre servisy potrebné na beh editora [41].

[4] Electron je softvérový open source framework umožňujúci vývoj desktopových GUI aplikácií pomocou webových technológií [48].

5.2 Try

Try je mód Try .NET-u poskytujúci možnosť integrovať editor so spustiteľným kódom do ľubovoľnej webovej stránky za pomoci HTML elementu i-frame. Grafický výstup tohto módu je na rozdiel od módu Hosted iba monaco editor spolu s konzolovým výstupom. Príklad použitia módu Try nástroja Try .NET je zobrazený na obrázku číslo 25.



Obrázok 25 Príklad použitia módu Try nástroja Try .NET [42]

V čase písania tejto diplomovej práce nie je mód Try nástroja Try .NET prístupný pre širokú verejnosť. Integrácia editora je možná iba do webových stránok, ktoré povoľuje firma Microsoft. Z tohto dôvodu bude opísaný mód iba v krátkosti na základe jeho vstupov, výstupov a použitých technológií.

Proces prekladu a spustenia kódu sa značne líši od módu Hosted. Po spustení kódu užívateľom sa užívateľský kód z editora pošle z webového prehliadača na server. Server užívateľský kód preloží pomocou prekladača Roslyn a výsledok z prekladu zakóduje do formátu base64. Pri preklade nie je užívateľský kód vložený do predpripraveného kódu VS .NET C# projektu. Preto musí byť užívateľský kód úplný a musí obsahovať statickú metódu *Main*, ktorá slúži ako vstupný bod programu. Výsledok prekladu vo forme base64 sa zašle späť do webového prehliadača, kde sa kód spustí a vyhodnotí jeho výstup. Mód Try nástroja Try .NET je postavený na UI frameworku Blazor. Za pomoci tohoto frameworku je po dekódovaní výstupu prekladu tento výstup spúšťaný na strane klienta vo webovom prehliadači.


```
{
  "workspace": {
    "workspaceType": "blazor-console",
    "buffers": [
      {
        "id": "Program.cs",
        "content": "using System;\nusing System.Collections.Generic;\nusing System.Linq;\n\nnamespace myAp
p\n{\nclass Program\n{\nstatic void Main()\n{\nConsole.WriteLine(\"Hello World!\");\n}\n}\n}",
        "position": 0
      }
    ],
    "files": [],
    "includeInstrumentation": false
  },
  "activeBufferId": "Program.cs",
  "requestId": "2OeWQ"
}
```


Ukážka kódu 3 Request o preklad kódu módu Try


```
{
  "requestId": "2OeWQ",
  "base64assembly": "TVqQAAMAAAAEAAAA//8AALg...AAA==",
  "succeeded": true,
  "diagnostics": [],
  "projectDiagnostics": []
}
```


Ukážka kódu 4 Response prekladu kódu módu Try

Requeste o preklad užívateľského kódu zobrazený v ukážke kódu číslo 3 obsahuje pole *content*. Toto pole obsahuje celý užívateľský kód z obrázku číslo 25. Response servera na tento request je zobrazený v ukážke kódu číslo 4. Pole s názvom base64assembly obsahuje výsledok prekladu vo formáte base64.

5.3 Porovnania módov Try a Hosted

Módy Try a Hosted zdieľajú zdrojový kód. Ich účel a použité technológie pre splnenie tohoto účelu sú však odlišné. Mód Hosted slúži primárne na vytváranie interaktívnej dokumentácie so spustiteľným kódom v jazyku C#. Preklad a spustenie kódu sa odohráva na servery. Naopak mód Try má poskytovať jednoduchú možnosť integrovania editora so spustiteľným kódom v ľubovoľnej webovej aplikácií. Preklad sa odohráva na strane servera a spustenie kódu prebieha v prehliadači užívateľa. Mód Hosted je verejne dostupný, zatiaľ čo mód Try je povolený iba pre webové stránky povolené firmou Microsoft.

II. PRAKTICKÁ ČASŤ

6 MOŽNOSTI VYUŽITIA TECHNOLÓGIE TRY .NET PRI VÝUKY PROGRAMOVANIA

Technológia Try .Net má vysoký potenciál využitia pri výuke programovacích jazykov z rodiny .NET. Jedno z primárnych využití môže byť vytvorenie interaktívnej dokumentácie slúžiacej na prednášky pomocou módu Hosted. Klasická prednáška je prezentovaná pomocou statického dokumentu vo forme *.pdf* alebo *.pptx*. Pri nutnosti demonštrácie prednášajúci spúšťa vývojové prostredie a na príkladoch ozrejmuje nejasnosti študentom. Jednou z výhod interaktívnej prednášky by bolo odstránenie rozptýlenia študentov počas prepínania okien medzi samotnou prednáškou a vývojovým prostredím. Ďalšou výhodou by bola značná úspora času prednášajúceho, ktorý by pomocou dokumentácie mal všetky príklady predpripravené a spustiteľné priamo v dokumentácií. Vyhol by sa tak problémom so spustením alebo inštaláciou vývojového prostredia na zobrazovacom zariadení.

Príklad prednášky zaoberajúcou sa dedičnosťou objektových tried je uvedený na obrázku nižšie.

Obrázok 26 Ukážka prednášky dedičnosti v jazyku C# pomocou Try .NET

Editor so spustiteľným kódom poskytovaný módom Try nástroja Try .NET môže nájsť uplatnenie v špecifických aplikáciách slúžiacich na výuku, kde je potrebná vysoká flexibilita riešenia, ktorú mód Hosted neposkytuje. Príkladom špecifickej aplikácie môže byť aplikácia určená na automatické testovanie programov vypracovaných študentmi, ktorej sa venuje táto práca. Mód nie je v čase písania práce verejne dostupný, preto jeho využitie nie je možné priamo demonštrovať.

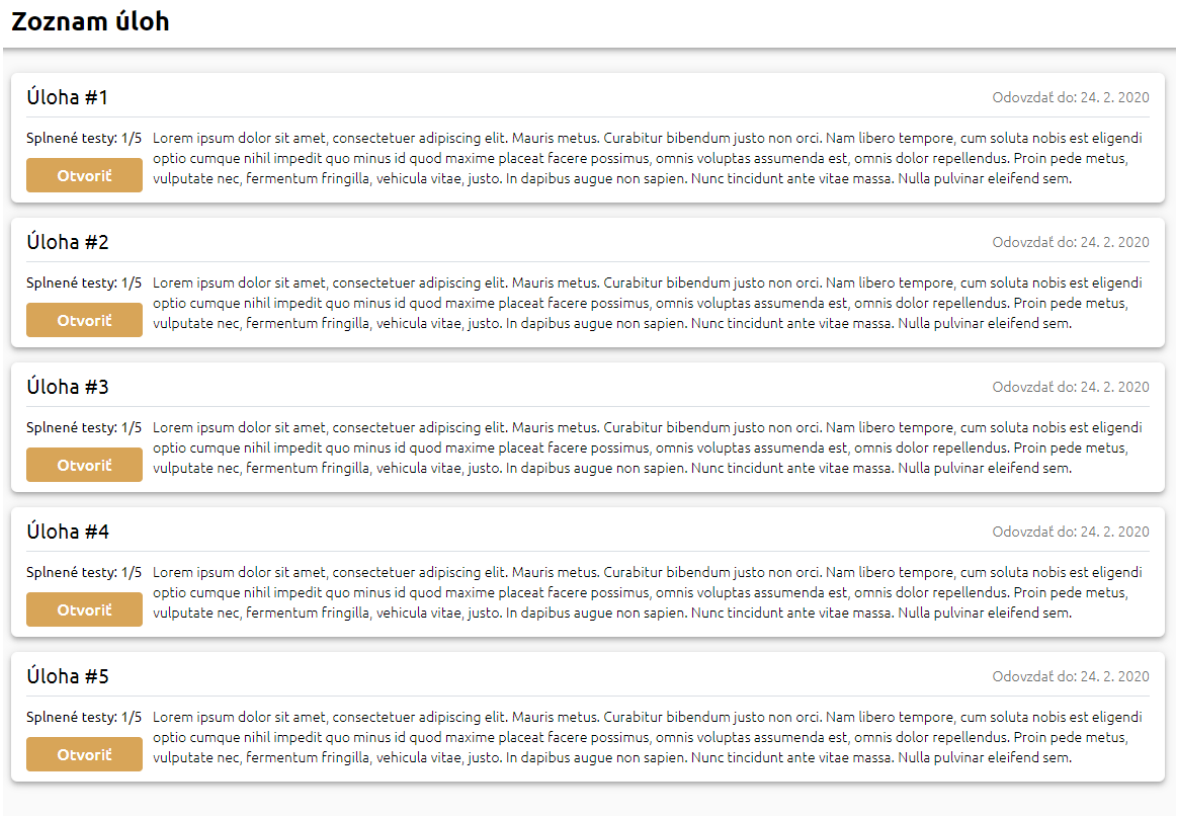
7 AUTOMATICKÁ KONTROLA TESTOV

V nasledujúcich podkapitolách budú zrealizované dve webové aplikácie určené na automatické vyhodnocovanie testov vypracovaných študentmi. Prvá aplikácia bude využívať verejne dostupný mód Hosted a bude demonštrovať možnosti a nevýhody módu, ktoré vyplývajú z jeho veľmi špecifického zamerania. Druhou aplikáciou bude webová aplikácia, obsahujúca vlastnú implementácia módu Try nástroja Try .NET. Pomocou vlastnej implementácie je možné demonštrovať využitie módu Try napriek tomu, že originálna implementácia nie je momentálne verejne dostupná.

7.1 Ciele webových aplikácií

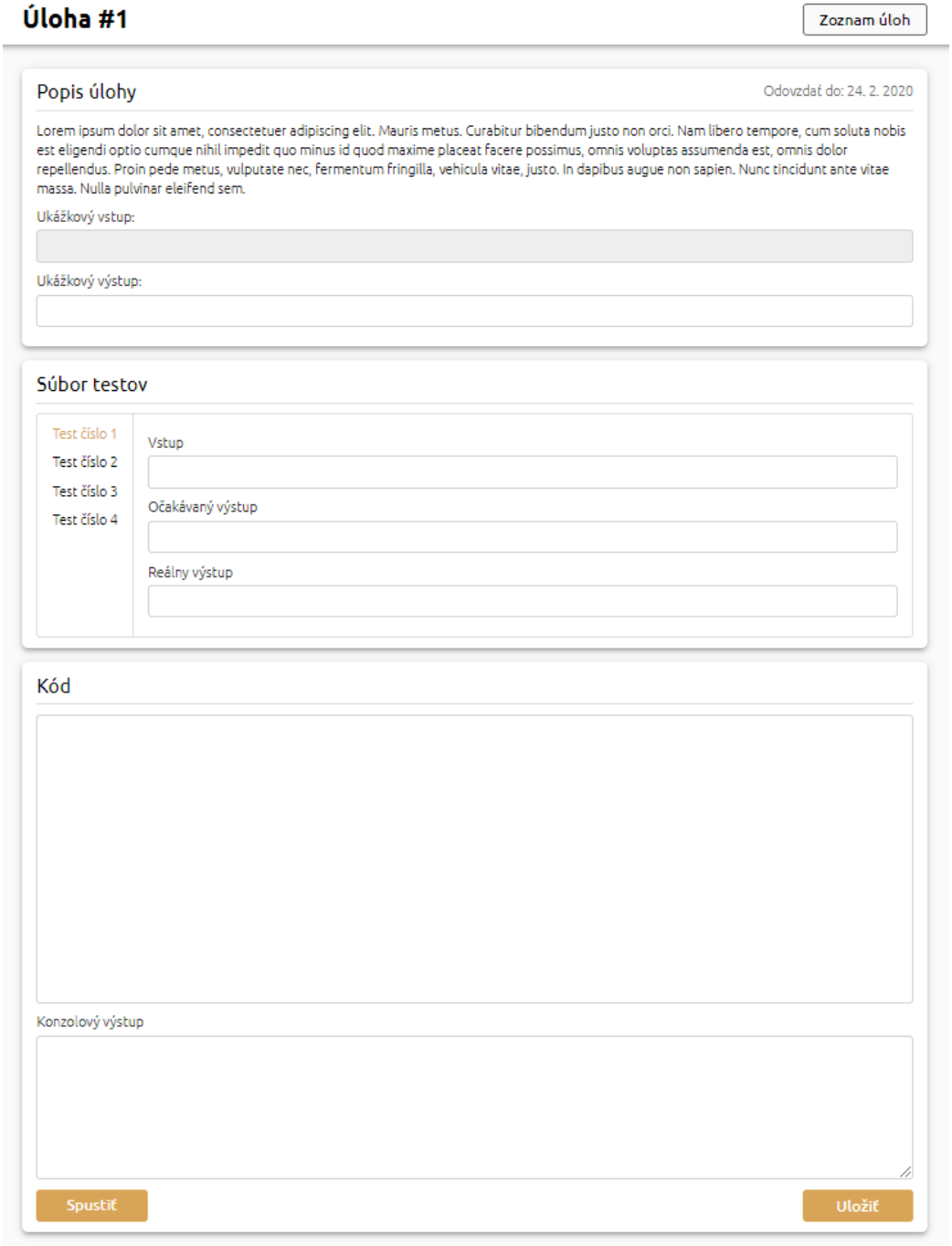
Cieľom je vytvoriť webovú aplikáciu, ktorá bude schopná automaticky vyhodnocovať testy vypracované študentmi (užívateľmi). Aplikácie musia obsahovať dve stránky, ktorými sú domovská stránka a detail úlohy.

Domovská stránka bude obsahovať zoznam úloh. Zobrazenie jednotlivých úloh musí obsahovať názov úlohy, jej krátky popis a dosiahnuté výsledky v podobe počtu splnených testov z celkového počtu testov v súbore.



Obrázok 27 Grafický návrh domovksej stránky webovej aplikácie

Po stlačení tlačidla *Otvoriť* dôjde k otvoreniu druhej stránky, stránka detailu úlohy. Táto stránka obsahuje mimo iné názov úlohy a jej popis spolu s ukážkovým vstupom a výstupom programu. Ďalej súbor testov, ktorý má program vypracovaný užívateľom spĺňať v rámci danej úlohy. Každý test zo súboru obsahuje vstup, očakávaný výstup a výstup z programu študenta. Test zo súboru bude považovaný za splnený, ak očakávaný výstup a reálny výstup z programu študenta sú hodnotovo zhodné. Posledná sekcia bude obsahovať editor s konzolovým výstupom a tlačidlami na spustenie kódu a uloženie úlohy.



Obrázok 28 Grafický návrh detailu úlohy

Plnohodnotná webová aplikácia na automatické vyhodnocovanie testov vypracovaných užívateľmi by mala, mimo opísaných stránok, ďalej obsahovať možnosť implementácie autentifikácie a autorizácie užívateľov, uloženia a odoslania úloh a CRUD operácií nad úlohami administrátorom. Implementácií týchto funkcionalít sa táto práca venovať nebude, pretože nie sú kritické pre demonštráciu využitia technológie Try .NET. Budú však zhodnotené možnosti ich nasadenia pre mód Hosted a pre webovú aplikáciu využívajúcu implementáciu módu Try.

7.2 Automatická kontrola testov pomocou Try .NET Hosted

Prvým krokom implementácie automatickej kontroly testov bolo vytvorenie potrebných zložiek a súborov. Boli vytvorené zložky dokumentácie a zložka projektu. V zložke projektu bol vytvorený .NET konzolový projekt a do zložky dokumentácie boli vytvorené detaily testov. Na vytvorenie súborov a zložiek boli použité príkazy spúšťané pomocou príkazového riadka uvedené nižšie.

```
C:\Projects>md AutomaticTestsHosted\Project
C:\Projects>type nul > AutomaticTestsHosted\testDetail1.md
C:\Projects>type nul > AutomaticTestsHosted\testDetail2.md
C:\Projects>type nul > AutomaticTestsHosted\testDetail3.md
C:\Projects>dotnet new console -o AutomaticTestsHosted
The template "Console Application" was created successfully.

Processing post-creation actions...
Running 'dotnet restore' on AutomaticTestsHosted\AutomaticTestsHosted.csproj...
  Determining projects to restore...
  Restored C:\Projects\AutomaticTestsHosted\AutomaticTestsHosted.csproj (in 127 ms).

Restore succeeded.

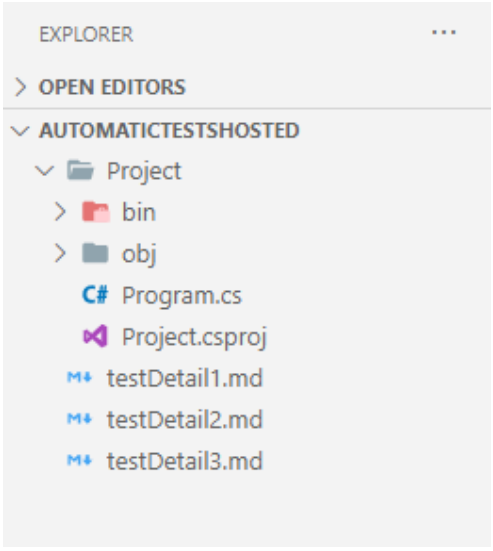
C:\Projects>dotnet new console -o AutomaticTestsHosted\Project
The template "Console Application" was created successfully.

Processing post-creation actions...
Running 'dotnet restore' on AutomaticTestsHosted\Project\Project.csproj...
  Determining projects to restore...
  Restored C:\Projects\AutomaticTestsHosted\Project\Project.csproj (in 136 ms).

Restore succeeded.
```

Ukážka kódu 5 Príkazy na vytvorenie súborov a zložiek – Hosted

Výsledná štruktúra je zobrazená na obrázku číslo 29.



Obrázok 29 Štruktúra projektu módu Hosted

Dokumentácia je spustená pomocou príkazu *dotnet-try* z koreňového priečinka.

```
C:\Projects\AutomaticTestsHosted> dotnet-try
```

Na obrázku číslo 30 je zobrazená domovská stránka dokumentácie, ktorá je spustená na lokálnom servery. Nástroj neposkytuje žiadnu možnosť editácie domovskej stránky, preto nie je možné po vizuálnej ani obsahovej stránke priblížiť sa grafickým návrhom, ktoré boli uvedené v cieľoch webových aplikácií.

Obrázok 30 Domovská stránka - Hosted

Po dokončení domovskej stránky bola vytvorená prvá úloha, ktorá sa venuje implementácií metódy súčtu dvoch čísel. Úloha je tvorená dvomi súbormi, ktorými sú *testDetail1.md* a *Program.cs*. Markdown súbor vytvára vizuálny výstup stránky, zatiaľ čo v C# súbore je implementovaná logika porovnania výsledkov testov.

V rámci implementácie prvej úlohy bol upravený Markdown súbor *testDetail1.md*. V súbore bol vytvorený nadpis úlohy (1), popis úlohy (2), ukážkové vstupy (3), súbor testov (4) a editor odkazujúci na súbor *Program.cs* (5). V súbore *Program.cs* bude neskôr aj región *userCode*, ktorého obsah bude zobrazený v editore po načítaní stránky. Opäť nie je možné priblížiť sa po grafickej stránke návrhom, pretože nie je možné vnoriť Markdown element do elementu HTML. Obsah súboru *testDetail.md* je uvedený nižšie. Označené časti kódu odpovedajú označeniam v texte vyššie.

```
<!-- 1 -->
# Úloha #1 - sčítanie
<!-- 2 -->
## Popis úlohy
Implementujte metódu Add tak aby jej výsledkom bolo sčítanie dvoch vstupných paramet-
rov 'a', 'b'.
<!-- 3 -->
**Ukážkový vstup**
```
100, 10
```
**Ukážkový výstup**
```
110
```
<!-- 4 -->
## Súbor testov
### Test číslo 1
**Vstup**
```
1, 2
```
**Očakávaný výstup**
```
3
```
### Test číslo 2
**Vstup**
```
3, 4
```
**Očakávaný výstup**
```
7
```
<!-- 5 -->
## Kód
```cs --source-file ./Project/Program.cs --project ./Project/Project.csproj
--region userCode
```
```

Ukážka kódu 6 Detail úlohy módu Hosted

V súbore *Program.cs* je implementovaná jednoduchá logiku na vyhodnotenie testov v metóde *CreateTestResultMessage*. Metóda vracia v prípade zhody očakávaného a reálneho výsledku hlášku *X prešiel.*, kde X je meno testu. V opačnom prípade vracia hlášku *X neprešiel. Očakávaný výstup Y. Reálny výstup Z.,* kde X je meno testu, Y očakávaný výstup a Z reálny výstup. Poslednou časťou súboru *Program.cs* je región *userCode*, obsahujúci kostru metódy, ktorú má v rámci úlohy dopísať užívateľ. Obsah regiónu bude zobrazený v editore pri načítaní stránky.

```
using System;

namespace Project
{
  class Program
  {
    static void Main(string[] args)
    {
      Console.WriteLine(CreateTestResultMessage(1 + 2, Add(1, 2), "Test case 1"));
      Console.WriteLine(CreateTestResultMessage(3 + 4, Add(3, 4), "Test case 2"));
    }

    #region userCode
    public static int Add(int a, int b)
    {
      // Tu implementujte súčet dvoch čísiel.
      return 0;
    }
    #endregion

    private static string CreateTestResultMessage(int expectedOutput,
       int output, string testName)
    {
      return $"{testName} {(expectedOutput == output ?
       "prešiel." :
       $"neprešiel. Očakávaný výstup {expectedOutput}. Reálny výstup {output}.")}";
    }
  }
}
```

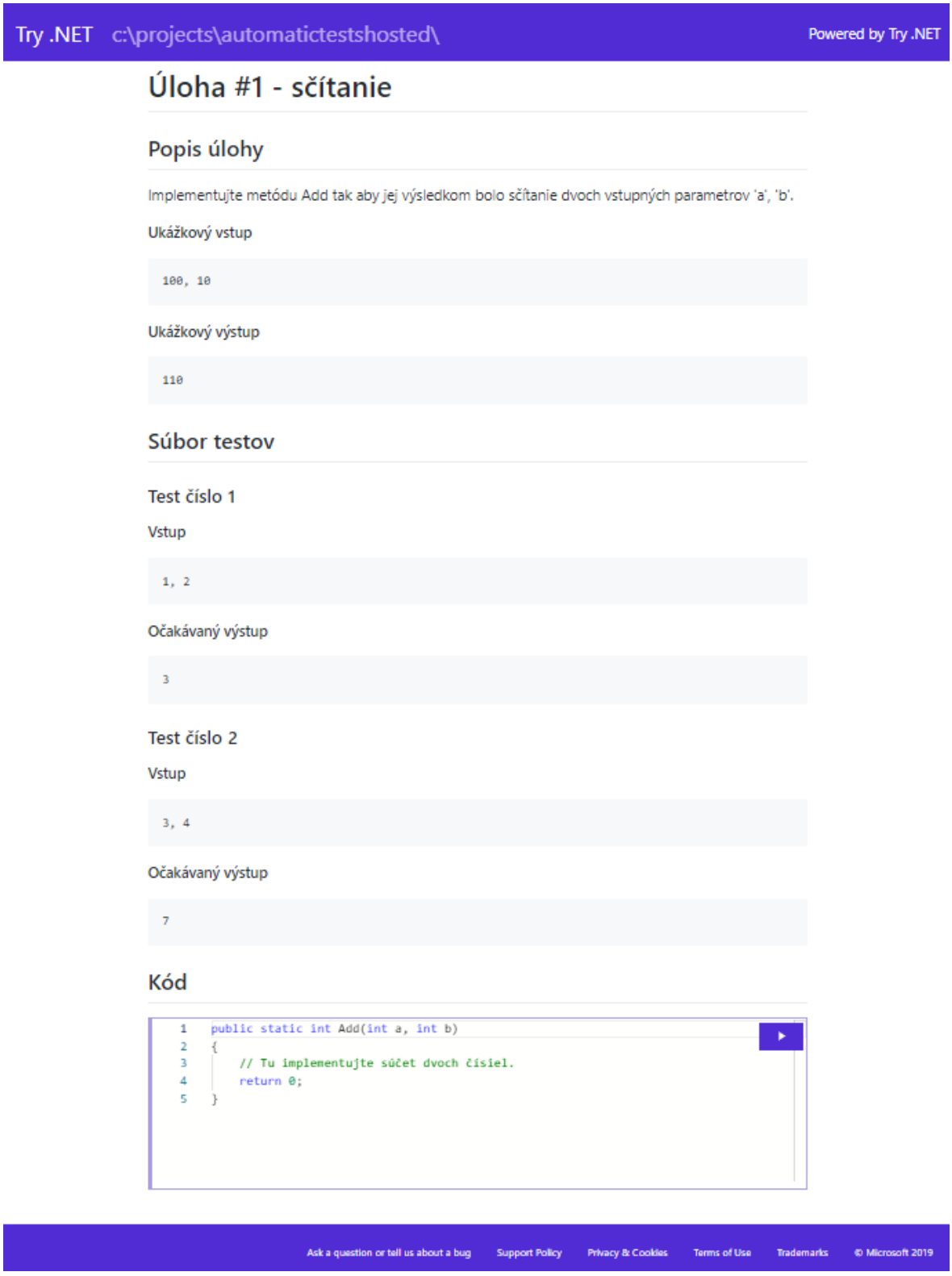
Ukážka kódu 7 Obsah súboru *Program.cs* - Hosted

Po vytvorení potrebných súborov sa overí platnosť obsahu týchto súborov pomocou príkazu *verify*.

```
PS C:\Projects\AutomaticTestsHosted> dotnet-try verify
C:\Projects\AutomaticTestsHosted\testDetail1.md
---------------------------------------------
  Checking Markdown...
    ✓  Line 36: C:\Projects\AutomaticTestsHosted\Project\Program.cs (in project
C:\Projects\AutomaticTestsHosted\Project\Project.csproj)
  Compiling samples for region "userCode"
    ✓  No errors found within samples for region "userCode"
C:\Projects\AutomaticTestsHosted\testDetail2.md
---------------------------------------------
C:\Projects\AutomaticTestsHosted\testDetail3.md
---------------------------------------------
dotnet try verify found 0 error(s)
```

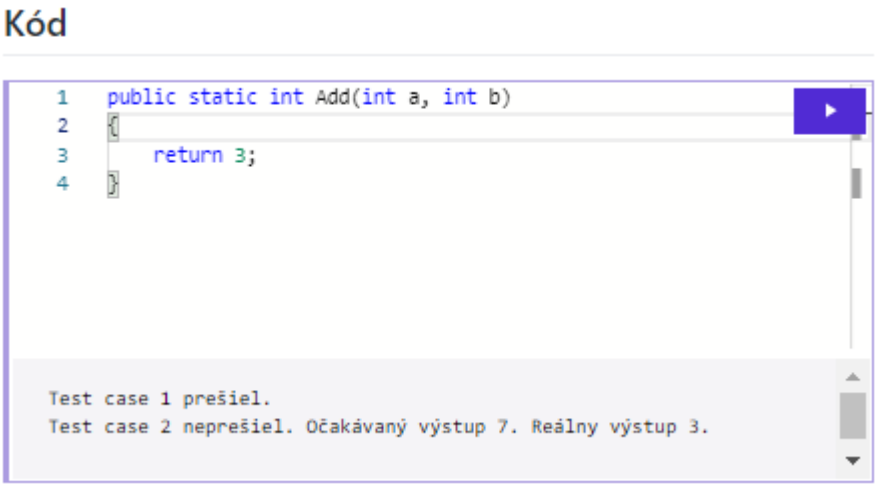
Ukážka kódu 8 Overenie platnosti obsahu súborov - Hosted

Z výpisu vyplýva, že súbory neobsahujú žiadnu chybu. Po spustení dokumentácie pomocou príkazu *dotnet-try* a prejdení na detail úlohy sa zobrazí stránka, ako na obrázku nižšie.



Obrázok 31 Detail úlohy - Hosted

Spustením kódu editora sa študentova implementácia metódy pošle na server, kde sa overí jej správnosť pomocou súboru vstupných a výstupných hodnôt. V prípade nesprávnej implementácie je zobrazená hláška, informujúca o očakávanej hodnote a reálnej hodnote vypočítanej pomocou metódy *Add*.



Obrázok 32 Nesprávna implementácia metódy *Add*

7.2.1 Zhodnotenie

Po vytvorení domovskej stránky a prvého detailu úlohy sú nevýhody tohto módu pre vytvorenie aplikácie na automatické vyhodnocovanie testov vypracovaných užívateľmi zrejmé. Vytváranie ďalších testov stráca zmysel. Tieto nevýhody sú:

1. Nie je možné zmeniť domovskú stránku po obsahovej ani vizuálnej stránke.
2. Vývojár nemá priamu kontrolu nad generovaním stránky.
3. Nie je možné vnoriť Markdown element do HTML elementov.
4. Nie je možné implementovať autentifikáciu ani autorizáciu užívateľov.
5. Nie je možné implementovať uloženie a odoslanie úloh.
6. Nie je možné implementovať CRUD operácie nad úlohami pre administrátorov.
7. Využitie tohto módu oproti iným riešeniam nemá prakticky žiadnu výhodu.

Väčšina nevýhod plynie zo špecifického zamerania módu Hosted na vytváranie interaktívnej dokumentácie. Tento mód nie je dostatočne flexibilný na iné využitie, ako bol jeho primárny účel.

7.3 Automatická kontrola testov pomocou vlastnej implementácie Try .NET Try

Cieľom implementácie módu Try nástroja Try .NET je vytvoriť aplikáciu, ktorá by slúžila ako ukážka využitia módu Try pri kontrole testov vypracovaných študentami. V rámci aplikácie bude integrovaný editor monaco. Webová aplikácia bude pri práci s editorom čo najbližšie kopírovať procesy použité v móde Try. Cieľom je zabezpečiť preklad užívateľského kódu na strane servera a následne jeho spustenie na strane webového prehliadača užívateľa. Na implementáciu budú použité rovnaké technológie, ktorými sú UI framework Blazor a prekladač Roslyn na preklad užívateľského kódu. Aplikácia bude schopná automaticky vyhodnotiť správnosť úlohy, ktorá bude vypracovaná užívateľom. Správnosť sa vyhodnotí pomocou súboru testov, ktoré budú obsahovať vstup, reálny výstup a očakávaný výstup.

V záujme zvýšenia prehľadnosti opisu aplikácie budú v texte tejto práce opísané iba kľúčové prvky implementácie. Za kľúčové prvky implementácie sú považované proces vytvorenia .NET riešenia, integrácia editora monaco, implementácia domovskej stránky a detail úlohy, implementácia kontroléru na preklad kódu a na prácu s úlohami, modely, komponenty a servisu na spustenie kódu.

7.3.1 Vytvorenie .NET riešenia *AutomaticTestTry*

Implementácia webovej aplikácie začala vytvorením Blazor WebAssembly projektu pre front-end a ASP.NET Core projektu pre back-end. Projekty sú vytvorené pomocou príkazového riadka. Príkaz na vytvorenie opísaných projektov v rámci .NET riešenia s názvom *AutomaticTestsTry* spolu s výsledkom tohoto príkazu je uvedený nižšie.

```
C:\Projects>dotnet new blazorwasm --hosted -o AutomaticTestsTry
The template "Blazor WebAssembly App" was created successfully.
Processing post-creation actions...
Running 'dotnet restore' on AutomaticTestsTry\AutomaticTestsTry.sln...
  Determining projects to restore...
  Restored C:\Projects\AutomaticTestsTry\Shared\AutomaticTestsTry.Shared.csproj
  (in 175 ms).
  Restored C:\Projects\AutomaticTestsTry\Client\AutomaticTestsTry.Client.csproj
  (in 288 ms).
  Restored C:\Projects\AutomaticTestsTry\Server\AutomaticTestsTry.Server.csproj
  (in 3.59 sec).
Restore succeeded.
```

Ukážka kódu 9 Vytvorenie .NET riešenia *AutomaticTestsTry*

7.3.1.1 Opis projektov .NET riešenia AutomaticTestTry

.NET riešenie *AutomaticTestsTry* pozostáva z projektov *Client*, *Server* a *Shared*.

Client je Blazor WebAssembly projekt, slúžiaci na vykresľovanie webových stránok pomocou Blazor UI frameworku. Novovytvorený projekt obsahuje zložky *Properties*, *wwwroot*, *Pages* a *Shared*. Pre implementáciu tejto webovej aplikácie sú dôležité priečinky:

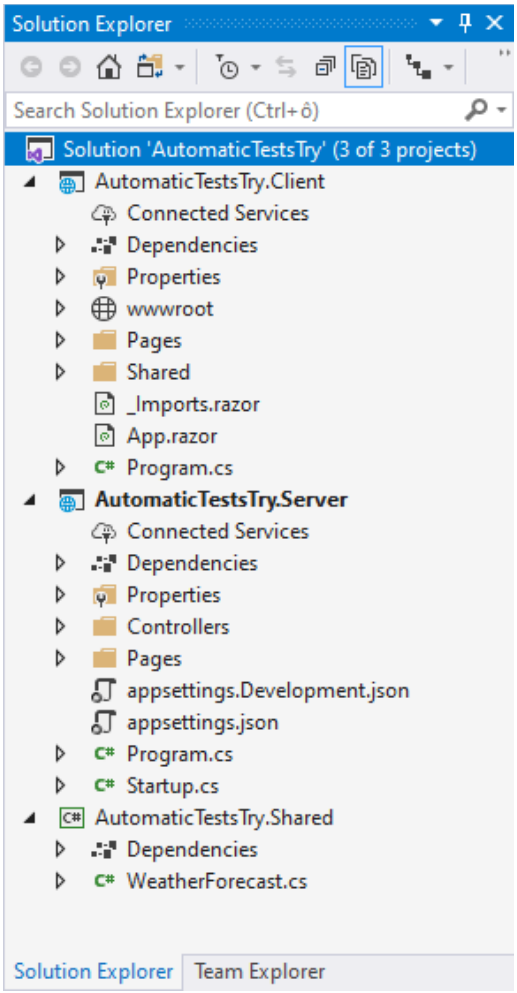
- *wwwroot* – Koreňová zložka webu aplikácie, ktorá obsahuje všetky verejne dostupné statické prostriedky aplikácie. Zložka obsahuje súbor index.html s JavaScript kódom, ktorý sa stará o stiahnutie .NET runtime-u.
- *Pages* – Zložka obsahuje razor stránky (komponenty).
- *Shared* – Zložka obsahuje zdieľané komponenty ako je napríklad layout stránky a navigačné okno.

Okrem priečinkov obsahuje projekt *Client* súbory *_Imports.razor*, *App.razor* a *Program.cs*:

- *_Imports.razor* – Súbor obsahuje globálne zdieľané direktívy pre import knižníc.
- *App.razor* – Koreňová komponenta aplikácie. Táto komponenta zabezpečuje smerovania na strane klienta.
- *Program.cs* – Vstupný bod aplikácie.

Server je ASP.NET Core projekt, slúžiaci na poskytovanie *Client* aplikácie užívateľom a na implementáciu časovo náročných a komplexných úloh, ako je napríklad spracovanie dát z databázy a preklad užívateľského kódu. Projekt sa skladá zo zložiek *Properties*, *Controllers* a *Pages*. Pre implementáciu webovej aplikácie je dôležitá zložka *Controllers*, ktorá obsahuje kontroléry na spracovanie jednotlivých endpointov API. Ďalej projekt obsahuje súbory *Program.cs* a *Startup.cs*, kde *Program.cs* je vstupný bod aplikácie a súbor *Startup.cs* slúži na definovanie logiky spúšťania aplikácie.

Posledným projektom vo vytvorenom .NET riešení je projekt *Shared*. Projekt slúži primárne na zdieľanie modelov a servís medzi projektami *Server* a *Client*.

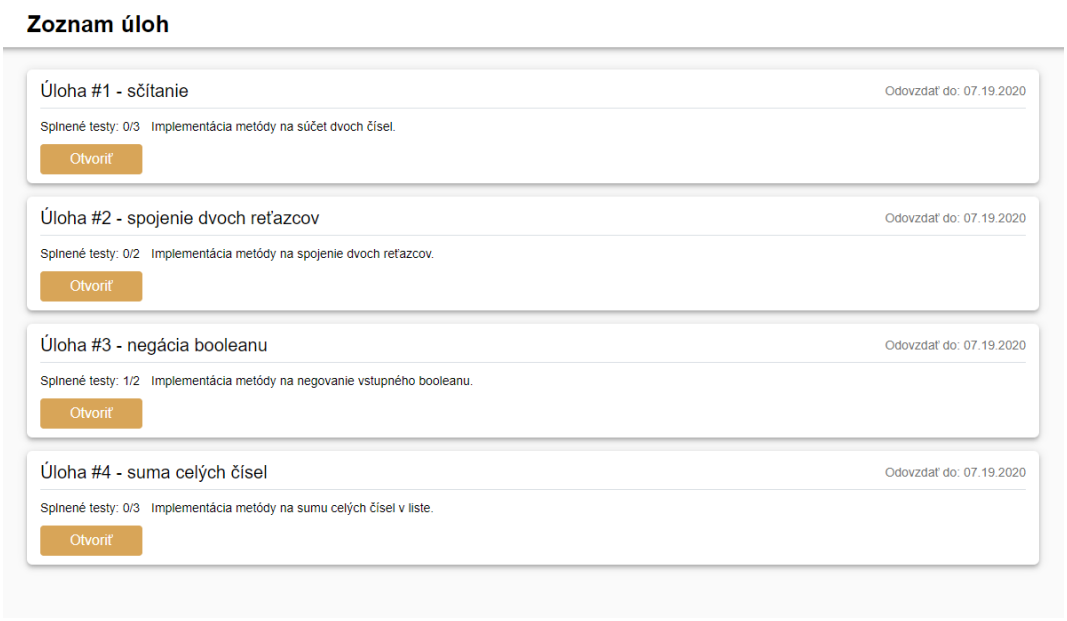


Obrázok 33 Štruktúra .NET riešenia *AutomaticTestsTry*

7.3.2 Domovská stránka

Domovská stránka sa skladá z hlavičky a zoznamu úloh na vypracovanie. Každá z úloh poskytuje informácie o teste, ako je názov, popis, dosiahnuté výsledky a dátum odovzdania.

Na detail jednotlivých úloh je možné prejsť pomocou tlačidla *Otvoriť*. Vykreslená domovská stránka je zobrazená na obrázku číslo 34.



Obrázok 34 Domovská stránka - Try

Domovská stránka je implementovaná v súbore *Pages/HomePage.razor*, v projekte *Client*. Smerovanie stránky je definované pomocou direktívy *page* a je nastavené na hodnotu / (index webovej aplikácie). Stránka využíva na zobrazenie komponenty, ktorými sú *LoaderComponent* a *AssignmentComponent*.

```
@page "/"
...
<div class="header">
    <div class="container">
        <div class="header-content">
            <h1>Zoznam úloh</h1>
        </div>
    </div>
</div>
<div class="container">
    <div class="box-list">
        @foreach (var assignment in Assignments)
        {
            <AssignmentComponent Assignment="assignment"></AssignmentComponent>
        }
    </div>
</div>
...
```

Ukážka kódu 10 Domovská stránka - Try

Po inicializácií kódu stránky je vyvolaná metóda *OnInitializedAsync*. V rámci tejto metódy je asynchrónne načítaný zoznam úloh z endpointu servera *Assignments/get-all*. Počas doby

spracovania tohoto načítania je užívateľovi pomocou komponenty *LoaderComponent* zobrazený indikátor zaneprázdnenosti aplikácie (ďalej len loader), ktorý užívateľovi zabraňuje interagovať so stránkou. Po asynchrónnom načítaní úloh z endpointu *Assignments/get-all* je loader skrytý a zobrazujú sa jednotlivé úlohy na vypracovanie.

```
protected override async Task OnInitializedAsync()
{
    var assignmentsJson = await Http.GetStringAsync($"Assignments/get-all");

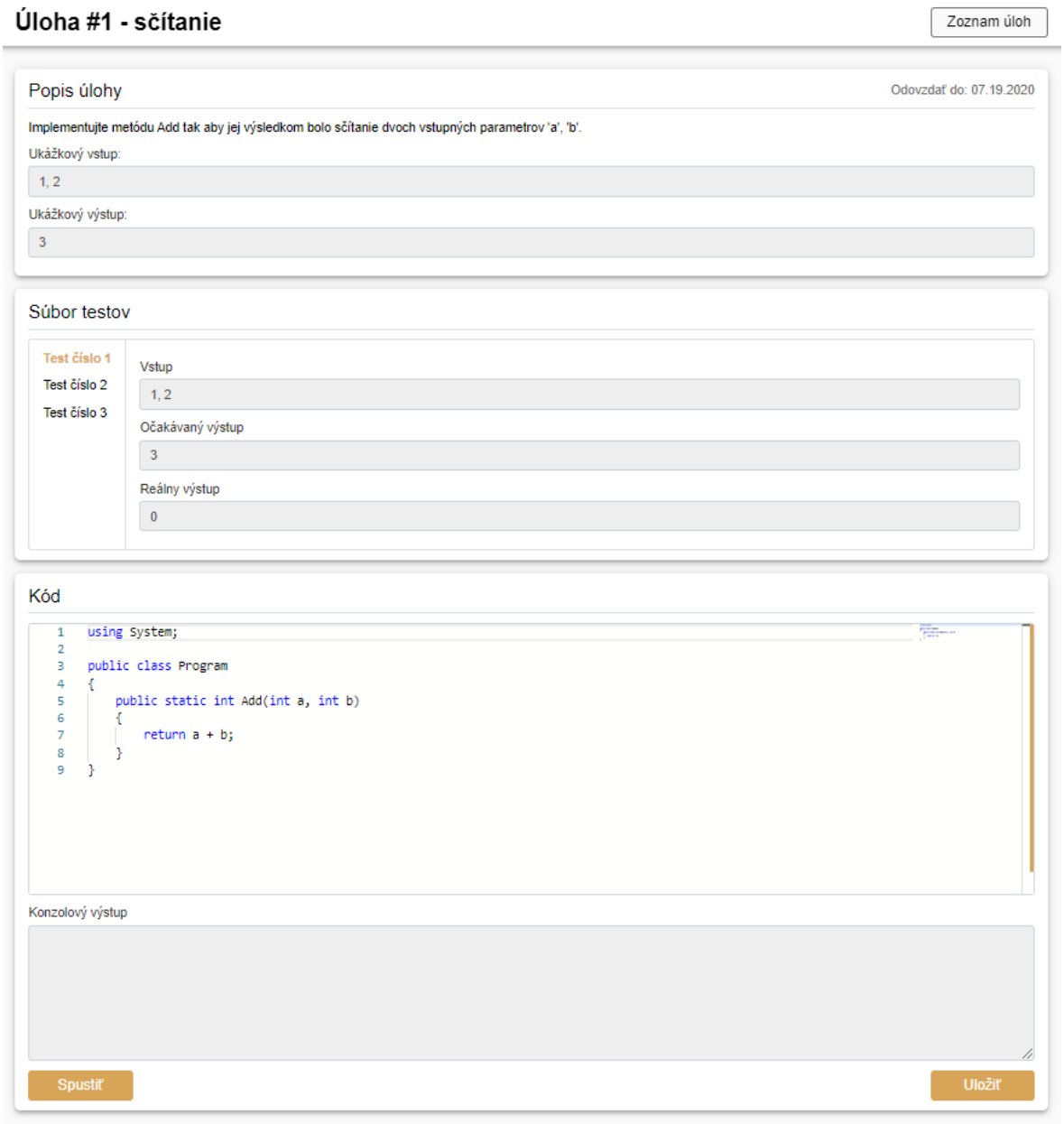
    Assignments = JsonConvert.DeserializeObject<List<Assignment>>(assignmentsJson,
      new JsonSerializerSettings
    {
        TypeNameHandling = TypeNameHandling.Objects
    });

    ShowLoader = false;
}
```

Ukážka kódu 11 Logika inicializácie domovskej stránky - Try

7.3.3 Detail úlohy

Detail úlohy sa skladá z názvu úlohy, presmerovania na zoznam úloh (domovskú stránku) vo forme tlačidla a troch sekcií. Prvá sekcia obsahuje popis úlohy spolu s ukážkovými vstupmi a výstupmi. Druhá sekcia zobrazuje súbor testov. Každý test má vstupnú hodnotu, očakávanú hodnotu a reálnu hodnotu z metódy implementovanou užívateľom. Posledná sekcia je zložená z editora monaco, konzolového výstupu z prekladu kódu užívateľa a tlačidiel na spustenie a uloženie kódu. Vykreslená stránka detailu úlohy je zobrazená na obrázku číslo 35.



Obrázok 35 Detail úlohy - Try

Detail úlohy je implementovaný v súbore *Pages/AssignmentPage.razor*, v projekte *Client*. Smerovanie stránky je definované pomocou direktívy *page* na cestu */assignment/{id:int}*. Kde *id:int* je zástupný symbol identifikačného čísla úlohy typu celé číslo. Toto identifikačné číslo je vstupným parametrom pre stránku detailu úlohy a je získané z URL.

```
@page "/assignment/{id:int}"
.
.
.
@code {
    [Parameter]
    public int ID { get; set; }
.
.
.
}
```

Ukážka kódu 12 Smerovania spolu so vstupnými parametrami detailu úlohy - Try

Po inicializácií kódu stránky, ktorá zahŕňa naplnenie vstupných parametrov, ako je napríklad *ID*, je vyvolaná metóda *OnInitializedAsync*. V rámci tejto metódy je asynchrónne načítaná úloha z endpointu servera *Assignments/get-by-id/{ID}*. Počas načítania úlohy je zobrazený loader, ktorý je následne po načítaní úlohy skrytý. Po načítaní úlohy a skrytí loaderu je manuálne vyvolané prekreslenie stránky pomocou metódy *StateHasChanged*.

```
protected override async Task OnInitializedAsync()
{
    var assignmentJson = await Http.GetStringAsync($"Assignments/get-by-id/{ID}");

    Assignment = JsonConvert.DeserializeObject<Assignment>(assignmentJson,
      new JsonSerializerSettings
    {
        TypeNameHandling = TypeNameHandling.Objects
    });

    SelectedTestCase = Assignment.TestCases.First();

    ShowLoader = false;

    StateHasChanged();
}
```

Ukážka kódu 13 Logika inicializácie stránky detailu úlohy - Try

Po vykreslení stránky je vyvolaná metóda *OnAfterRenderAsync*. Účelom tejto metódy je inicializovať editor monaco v prípade, že je načítaná úloha a editor nie je už inicializovaný. Inicializácia editora je implementovaná pomocou JavaScript funkcií *createEditor* a *setEditorValue*. Prvá funkcia editor vytvára, zatiaľ čo druhá nastavuje hodnotu zobrazenú editorom. Funkcie budú bližšie popísané v sekcií integrácie editora monaco.

```
protected override async Task OnAfterRenderAsync(bool firstRender)
{
    if (Assignment != null && !IsMonacoInitialized)
    {
        IsMonacoInitialized = true;

        await JSRuntime.InvokeVoidAsync("createEditor");
        await JSRuntime.InvokeVoidAsync("setEditorValue", Assignment.InitialCode);
    }
}
```

Ukážka kódu 14 Logika inicializácie editora monaco - Try

Udalosť stlačenia tlačidla *Spustiť* spracováva metóda *Run*. Metóda zobrazí loader a načíta užívateľský kód z editora monaco pomocou JavaScript funkcie *getEditorValue*, ktorá je volaná asynchrónne z C# kódu. Načítaný kód je následne zaobalený do requestu na preklad, ktorý je asynchrónne poslaný na endpoint serveru *Compile/compile*. V prípade úspešného prekladu kódu je kód spustený nad súborom testov pomocou triedy *CodeRunner*. Hodnoty získané spustením kódu nad súborom testov sú zobrazené v reálnych výstupoch. V prípade neúspešného prekladu užívateľského kódu sú chyby vypísané v konzolovom výstupe. Príklad tohto prípadu je zobrazený na obrázku číslo 37.

```
private async Task Run()
{
    ShowLoader = true;
    ConsoleOutput = "";

    var code = await JSRuntime.InvokeAsync<string>("getEditorValue");
    var compilationResult = await Http.PostJsonAsync<CompilationResult>(
       "Compile/compile", new CompilationRequest(code));

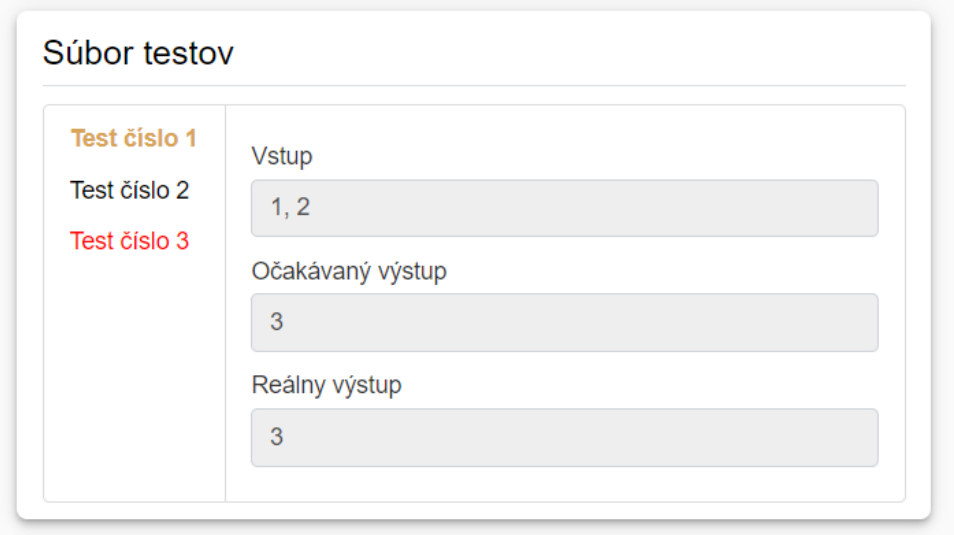
    if (compilationResult.Succeeded)
    {
        var codeRunner = new CodeRunner();
        var method = codeRunner.GetMethod(Assignment.EntryMethodNamespace,
              Assignment.EntryMethodName, compilationResult.Base64Assembly);
        var result = codeRunner.RunTestCases(method, Assignment.TestCases);
    }
    else
    {
        ConsoleOutput = string.Join(Environment.NewLine,
              compilationResult.CompilationErrors);

        foreach (var testCase in Assignment.TestCases)
        {
            testCase.TestOutput.ResetValue();
        }
    }

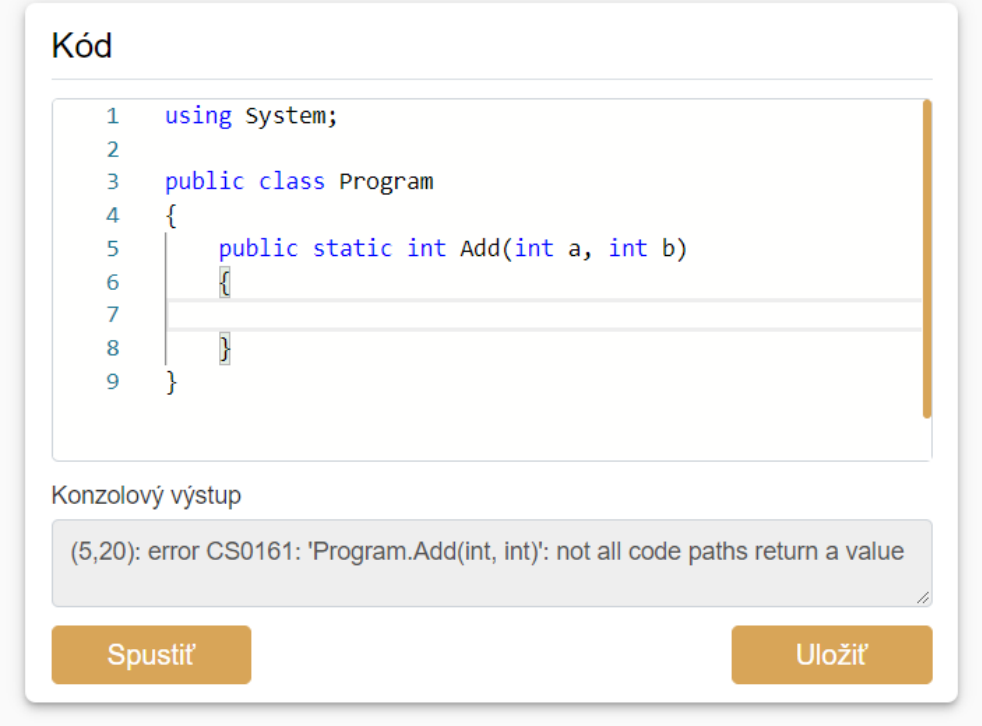
    HighlightFailedTests = true;
    ShowLoader = false;
}
```

Ukážka kódu 15 Logika spracovania spustenia užívateľského kódu - Try

Aktuálne zobrazený test je vyznačený zlatou farbou. Ak test splňuje podmienky testu (očakávaný výstup sa rovná reálnemu výstupu) je meno testu zlatej alebo čiernej farby. V opačnom prípade je meno testu vykreslené červenou farbou.



Obrázok 36 Vyhodnotený súbor testov – Try



Obrázok 37 Chyby užívateľského kódu vypísané v konzolovom výstupe

Udalosť stlačenia tlačidla *Uložiť* spracováva metóda *Save*. Metóda zobrazí loader a načíta užívateľský kód z editora monaco pomocou JavaScript funkcie *getEditorValue*, ktorá je volaná asynchrónne z C# kódu. Načítaný kód je následne zaobalený spolu s celou úlohou (*As-*

signment) do requestu na uloženie (*SaveAssignmentRequest*), ktorý je serializovaný a následne asynchrónne poslaný na endpoint serveru *Assignments/save*. Po spracovaní requestu serverom sa loader opäť skrýva.

```
private async Task Save()
{
    ShowLoader = true;

    var code = await JSRuntime.InvokeAsync<string>("getEditorValue");

    var request = JsonConvert.SerializeObject(new SaveAssignmentRequest(code,
      Assignment), new JsonSerializerSettings
    {
        TypeNameHandling = TypeNameHandling.Objects
    });

    await Http.PostAsync("Assignments/save", new StringContent(request,
      Encoding.UTF8, "application/json"));

    ShowLoader = false;
}
```

Ukážka kódu 16 Logika uloženia užívateľského kódu - Try

Celý zdrojový kód stránky detailu úlohy nie je uvedený z dôvodu jeho dĺžky. Celý kód je dostupný v prílohách práce.

7.3.4 Integrácia editora monaco

Editor monaco je integrovaný pomocou komunitného rozšírenia *Monaco Editor Webpack Loader Plugin*, ktoré zjednodušuje proces inicializácie editora. Integrácia editora začína inicializáciou projektu z pohľadu balíčkového manažéra *npm*. Inicializácia je spustená pomocou príkazového riadka v koreňovom priečinku projektu *Client*. Príkaz vytvorí súbor *package.json* so základnými údajmi, ako je meno, verzia a autor projektu. Primárny účel tohoto súboru je uchovanie metadát a závislostí projektu, ktoré budú doplnené neskôr.

```
PS C:\Projects\AutomaticTestsTry\Client> npm init –yes

Wrote to C:\Projects\test5\Client\package.json:

{
  "name": "Client",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

Ukážka kódu 17 Inicializácia balíčkového manažéra - Try

Ďalej boli pomocou balíčkového manažéra *npm* nainštalované balíčky webpack a webpack-cli. Webpack je nástroj slúžiaci na transformovanie JavaScript modulov spolu s ich závislosťami do statických súborov. Balíček webpack-cli umožňuje ovládať webpack pomocou príkazového riadka.

```
PS C:\Projects\AutomaticTestsTry\Client> npm install webpack webpack-cli

...

+ webpack-cli@3.3.12
+ webpack@4.43.0
added 391 packages from 218 contributors and audited 393 packages in 13.141s
found 0 vulnerabilities
```

Ukážka kódu 18 Inštalácia balíčkov webpack a webpack-cli - Try

Po inštalácií balíčkov boli nainštalované balíčky editora monaco a rozšírenie *Monaco Editor Webpack Loader Plugin*.

```
PS C:\Projects\AutomaticTestsTry\Client> npm install monaco-editor monaco-editor-
webpack-plugin

...

+ monaco-editor-webpack-plugin@1.9.0
+ monaco-editor@0.20.0
added 2 packages from 1 contributor and audited 395 packages in 3.889s
found 0 vulnerabilities
```

Ukážka kódu 19 Inštalácia balíčkov monaco-editor monaco-editor-webpack-plugin – Try

Posledné balíčky nutné na inštaláciu sú balíčky *style-loader*, *css-loader* a *file-loader*.

```
PS C:\Projects\AutomaticTestsTry\Client> npm install style-loader css-loader file-
loader

...

+ style-loader@1.2.1
+ css-loader@3.6.0
+ file-loader@6.0.0
added 24 packages from 56 contributors and audited 419 packages in 2.924s
found 0 vulnerabilities
```

Ukážka kódu 20 Inštalácia balíčkov style-loader css-loader file-loader - Try

Po inštalácií potrebných balíčkov je vytvorený súbor *index.js* v priečinku *wwwroot/src*. V súbore *index.js* sú v globálnom objekte *window* vytvorené funkcie na vytvorenie editora monaco, nastavenie kódu editora a jeho získanie. Editor sa vytvára pomocou funkcie *createEditor*. Táto funkcia nájde HTML elemente v DOM-e webovej stránky s identifikátorom *container* a inicializuje v ňom editor. Základným nastavením editora je jazyk *csharp* a prázdna reťazová hodnota, ktorá bude zobrazená v editore. Tieto funkcie sú následné volané z .NET kódu na zabezpečenie komunikácie s editorom.

```
import * as monaco from 'monaco-editor/esm/vs/editor/editor.api';

window.createEditor = () => {
    window.editor = monaco.editor.create(document.getElementById('container'), {
        value: '',
        language: 'csharp'
    });
};

window.setEditorValue = (code) => {
    return window.editor.setValue(code);
}

window.getEditorValue = () => {
    return window.editor.getValue();
}
```

Ukážka kódu 21 Obsah súboru *index.js* - Try

Ďalším krokom integrácie je vytvorenie konfiguračného súboru *webpack.config.js* pre nástroj *webpack*. Súbor bol vytvorený v koreňovom priečinku projektu *Client* s obsahom uvedeným nižšie.

```
const path = require("path");
const MonacoWebpackPlugin = require("monaco-editor-webpack-plugin");

module.exports = {
    mode: 'development',
    entry: './wwwroot/src/index.js',
    context: __dirname,
    output: {
        path: path.resolve(__dirname, 'wwwroot/dist'),
        filename: 'monaco.js',
        chunkFilename: '[id].js',
        publicPath: './dist/'
    },
    module: {
        rules: [{
            test: /\.css$/,
            use: ['style-loader', 'css-loader']
        }, {
            test: /\.ttf$/,
            use: ['file-loader']
        }]
    },
    plugins: [
        new MonacoWebpackPlugin()
    ]
};
```

Ukážka kódu 22 Konfiguračný súbor *webpack.config.js* - Try

Konfiguračný súbor bol následne využitý v sekcií skriptov (*scripts*) v súboru *package.json.* V tejto sekcií bol vytvorený príkaz *dev*, ktorý spúšťa nástroj *webpack* s konfiguračným súborom. Výsledný obsah súboru *package.json* je uvedený nižšie. V sekcií závislostí (*dependencies*) sú uvedené balíčky, na ktorých je projekt závislý.

```
{
  "name": "Client",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "dev": "webpack --config webpack.config.js"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "css-loader": "^3.6.0",
    "file-loader": "^6.0.0",
    "monaco-editor": "^0.20.0",
    "monaco-editor-webpack-plugin": "^1.9.0",
    "style-loader": "^1.2.1",
    "webpack": "^4.43.0",
    "webpack-cli": "^3.3.12"
  }
}
```

Ukážka kódu 23 Konfiguračný súbor *package.json* - Try

Poslednou úpravou je pridanie skriptov *monaco* do indexu (*wwwroot/index.html*) aplikácie a spustenie skriptu *dev*.

```
<!DOCTYPE html>
<html>

<head>
      ...
</head>

<body>
      ...
    <script type="text/javascript" src="dist/monaco.js"></script>
    <script src="_framework/blazor.webassembly.js"></script>
</body>

</html>
```

Ukážka kódu 24 Pridanie skriptov *monaco* do súboru *index.html* - Try

Skript *dev* pomocou konfiguračného súboru vytvorí skript *monaco.js*. Tento skript obsahuje všetku potrebnú logiku na vytvorenie editora monaco.

```
PS C:\Projects\test4\Client> npm run dev

> client@1.0.0 dev C:\Projects\test4\Client
> webpack --config webpack.config.js
Hash: b9ac555b04adc03c3d82
Version: webpack 4.43.0
Time: 5068ms
...
```

Ukážka kódu 25 Spustenie príkazu *dev* – Try

7.3.5 Kontrolér AssignmentsController

Kontrolér *AssignmentsController* je implementovaný v priečinku *Controllers,* v projekte *Server* a obsahuje tri endpointy. Endpoint *get-all* je spracovaný metódou *GetAll*, ktorá vracia zoznam úloh zo statickej premennej *Assignments*. Endpoint *get-by-id/{id}* je spracovaný metódou *GetByID*. Metóda má vstupný argument id, ktorého typ je celé číslo. Id je číselný identifikátor úloh, na základe ktorého je získaná požadovaná úloha zo zoznamu úloh. Posledným endpointom je endpoint *save.* Spracovaný je metódou *Save*, ktorej vstupný argument je request *SaveAssignmentRequest*. Účel metódy je uložiť užívateľský kód a výsledky testov do úlohy, identifikovanej pomocou id. Skrátený kód kontroléra je uvedený nižšie.

```
[ApiController]
[Route("[controller]")]
public class AssignmentsController : Controller
{
    [HttpGet("get-all")]
    public ActionResult<IList<Assignment>> GetAll() => Assignments;

    [HttpGet("get-by-id/{id}")]
    public async Task<Assignment> GetByID(int id) =>
        Assignments.First(e => e.ID == id);

    [HttpPost("save")]
    public async Task Save(SaveAssignmentRequest request)
    {
        var assignment = Assignments.First(e => e.ID == request.Assignment.ID);

        assignment.InitialCode = request.Code;
        assignment.TestCases = request.Assignment.TestCases;
    }

    private static readonly List<Assignment> Assignments = new List<Assignment>(
    {
        new Assignment()...
        new Assignment()...
        new Assignment()...
        new Assignment()...

    private static string AddCode =>...
    private static string AddStringsCode =>...
    private static string NegateCode =>...
    private static string SumCode =>...
}
```

Ukážka kódu 26 Kontrolér *AssignmentsController* - Try

7.3.6 Kontrolér *CompileController*

Kontrolér *AssignmentsController* je implementovaný v priečinku *Controllers,* v projekte *Server*. Tento kontrolér obsahuje jeden endpoint s názvom *compile*, ktorý je spracovaný pomocou metódy *Compile*. Metóda má vstupný parameter *CompilationRequest* a jej účel je vytvoriť IL kód z užívateľského kódu, ktorý je následne spúšťaný vo webovom prehliadači.

Za účelom prekladu je vytvorená kompilácia pomocou metódy *Create* statickej triedy *CShaprCompilation*. Do kompilácie musia byť poslané všetky dynamické knižnice, na ktoré odkazuje užívateľský kód. V rámci tejto aplikácie musia byť knižnice kompatibilné s runtimom mono, pretože výsledný kód je spúšťaný práve týmto runtimemom. Z tohoto dôvodu je možné použiť iba knižnice .NET Frameworku, ktoré sú lokálne uložené v priečinku *WasmDlls/4.8* vo verzií 4.8. Knižnice sú posielané do kompilácie vo forme metadát.

```
var compilation = CSharpCompilation.Create(
    Path.GetRandomFileName(),
    syntaxTrees: new[] { CSharpSyntaxTree.ParseText(request.Code) },
    references: _dlls.Select(file => MetadataReference.CreateFromFile(file)),
    options: new CSharpCompilationOptions(OutputKind.DynamicallyLinkedLibrary));
```

Ukážka kódu 27 Vytvorenie kompilácie - Try

Pomocou metódy *Emit* je získaný IL kód preloženého užívateľského kódu. Na základe výsledku metódy *Emit* je IL kód zakódovaný do formátu *base64* alebo v prípade chyby sú spracované chyby prekladu užívateľského kódu. Po spracovaní je výsledok obalený do triedy *CompilationResult* a odoslaný klientovi.

```
[ApiController]
[Route("[controller]")]
public class CompileController : Controller
{
    private static readonly string[] _dlls =
      Directory.GetFiles("C:\\Program Files (x86)\\WasmDlls\\v4.8\\", "*.dll");

    [HttpPost("compile")]
    public async Task<IActionResult> Compile(CompilationRequest request)
    {
        var compilation = CSharpCompilation.Create(
            Path.GetRandomFileName(),
            syntaxTrees: new[] { CSharpSyntaxTree.ParseText(request.Code) },
            references: _dlls.Select(file => MetadataReference.CreateFromFile(file)),
            options: new CSharpCompilationOptions(OutputKind.DynamicallyLinkedLibrary
        ));

        using var stream = new MemoryStream();
        var emitResult = compilation.Emit(stream);

        var result = new CompilationResult()
        {
            Succeeded = emitResult.Success
        };

        if (emitResult.Success)
        {
            stream.Seek(0, SeekOrigin.Begin);

            result.Base64Assembly = Convert.ToBase64String(stream.ToArray());
        }
        else
        {
            result.CompilationErrors = emitResult.Diagnostics
                .Where(diagnostic => diagnostic.IsWarningAsError ||
                   diagnostic.Severity == DiagnosticSeverity.Error)
                .Select(diagnostic => diagnostic.ToString())
                .ToList();
        }

        return Ok(result);
    }
}
```

Ukážka kódu 28 Kontrolér *CompileController* – Try

7.3.7 CodeRunner

Trieda *CodeRunner* slúži na zapuzdrenie logiky práce s IL kódom užívateľského kódu. Trieda sa skladá z dvoch metód. Prvou metódou je metóda s názvom *GetMethod*. Metóda dekóduje a načíta IL kód. Zo vzniknutej zostavy sa následne pomocou vstupných parametrov *entryMethodNamespace* a *entryMethodName* načíta požadovaná metóda.

```
public MethodInfo GetMethod(string entryMethodNamespace, string entryMethodName,
       string base64Assembly)
{
    var bytes = Convert.FromBase64String(base64Assembly);
    var assembly = Assembly.Load(bytes);

    var type = assembly.GetType(entryMethodNamespace);
    var method = type.GetMethod(entryMethodName);

    return method;
}
```

Ukážka kódu 29 Získanie metódy zo zakódovanej zostavy - Try

Druhou metódou je metóda na vyhodnotenie užívateľského kódu oproti súboru testov. Prvý parameter je metóda, ktorá sa spustí so vstupnými parametrami jednotlivých testov. Výsledok spustenia je následne porovnávaný s očakávanou hodnotou testu. Ak je hodnota zo vstupnej metódy zhodná s očakávanou, tak sa test považuje za splnený. Porovnanie hodnôt je zapuzdrené do metódy *Passed*.

```
public IList<TestCaseResult> RunTestCases(MethodInfo method,
       IList<TestCase> testCases)
{
    return testCases.Select(testCase =>
    {
        var parameters = testCase.TestInput
            ?.Select(e => e?.GetValue())
            .ToArray();

        var result = method.Invoke(null, parameters);

        testCase.TestOutput.SetValue(result);

        return new TestCaseResult()
        {
            TestCaseID = testCase.ID,
            Passed = testCase.Passed()
        };
    }).ToList();
}
```

Ukážka kódu 30 Vyhodnotenie súboru testov - Try

7.3.8 Komponenty

V rámci aplikácie sú využité dve komponenty, ktorými sú *LoaderComponent* a *AssignmentComponent*. Prvá komponenta slúži na zobrazenie a skrytie loaderu pomocou vstupného argumentu *ShowLoader*.

```
@if (ShowLoader)
{
    <div class="loader">
        <div></div>
        <img src="loader.gif" />
    </div>
}

@code {
    [Parameter]
    public bool ShowLoader { get; set; }
}
```

Ukážka kódu 31 Komponenta *LoaderComponent* - Try

Komponenta *AssignmentComponent* slúži na zobrazenie triedy *Assignment* na domovskej stránke webovej aplikácie. Jej vstupným parametrom je objekt typu *Assignment*.

```
@using MasterThesis.Shared.Models;
@using MasterThesis.Shared.Helpers;

@inject NavigationManager NavigationManager

<div class="box">
    <div class="box-header">
        <h2>@Assignment.Name</h2>
        <div class="date gray">Odovzdať do: @Assignment.DueDate.ToString("MM.dd.yyyy")</div>
    </div>
    <div class="box-content">
        <div class="info">
          <h3>Splnené testy: @PassedTestCasesCount()/@Assignment.TestCases.Count</h3>
          <button @onclick="Solve" class="btn btn-primary">Otvoriť</button>
        </div>
        <div class="text">
          @Assignment.ShortDescription
        </div>
    </div>
</div>

@code {
   [Parameter]
   public Assignment Assignment { get; set; }

   private void Solve() =>
       NavigationManager.NavigateTo($"/assignment/{Assignment.ID}");

   private int PassedTestCasesCount() => Assignment.TestCases.Count(e => e.Passed());
}
```

Ukážka kódu 32 Komponenta *AssignmentComponent* - Try

7.3.9 Modely

V tejto podkapitole sú opísané hlavné modely aplikácie, ktorými sú model úlohy *Assignment* a model testu *TestCase*.

Model *Assignment* slúži na uchovávania a prenos informácií o jednotlivých úlohách. Model obsahuje identifikačné číslo úlohy, meno, krátky a dlhý popis úlohy, dátum odovzdania, ukážkový vstup a výstup, súbor testov a inicializačný kód zobrazený po načítaní stránky v editore monaco. Tieto polia sú zobrazené na front-ende webovej aplikácia. Polia EntryMethodNamespace a EntryMethodName slúžia na identifikáciu menného priestoru a mena metódy v kóde užívateľa. Tieto parametre slúžia ako vstupné parametre pre metódu *CodeRunner.GetMethod*.

```
public class Assignment
{
    public int ID { get; set; }

    public string Name { get; set; }

    public string ShortDescription { get; set; }

    public string Description { get; set; }

    public DateTime DueDate { get; set; }

    public string SampleInput { get; set; }

    public string SampleOutput { get; set; }

    public IList<TestCase> TestCases { get; set; }

    public string InitialCode { get; set; }

    public string EntryMethodNamespace { get; set; }

    public string EntryMethodName { get; set; }
}
```

Ukážka kódu 33 Model *Assignment* – Try

Dôležitou súčasťou modelu *Assignment* je list objektov typu *TestCase*. Tento dátový typ obsahuje identifikačné číslo testu, list vstupných parametrov, reálny výstup a očakávaný výstup.

```
public class TestCase
{
    public int ID { get; set; }

    public IList<Parameter> TestInput { get; set; }

    public Parameter TestOutput { get; set; }

    public Parameter ExpectedTestOutput { get; set; }
}
```

Ukážka kódu 34 Model *TestCase* - Try

Parameter je abstraktná trieda poskytujúca rozhranie pre ostatné parametre, ktoré sú využité pri implementácií rôznych testov s rôznymi typmi vstupných a výstupných parametrov.

```
public abstract class Parameter
{
    public abstract object GetValue();

    public abstract void SetValue(object value);

    public abstract void ResetValue();
}
```

Ukážka kódu 35 Model *Parameter* - Try

Triedu *Parameter* dedia triedy *BoolParameter*, *IntParameter*, *StringParameter* a *ListParameter*. Triedy musia implementovať metódy *GetValue*, *SetValue*, *ResetValue* a *ToString*.

```
public class BoolParameter : Parameter
{
    public BoolParameter() { }

    public BoolParameter(bool value) => Value = value;

    public bool Value { get; set; }

    public override object GetValue() => Value;

    public override void SetValue(object value) => Value = (bool)value;

    public override void ResetValue() => Value = false;

    public override string ToString() => Value.ToString().ToLower();
}
```

Ukážka kódu 36 Model *BoolParameter* - Try

```
public class IntParameter : Parameter
{
    public IntParameter() { }

    public IntParameter(int value) => Value = value;

    public int Value { get; set; }

    public override object GetValue() => Value;

    public override void SetValue(object value) => Value = Convert.ToInt32(value);

    public override void ResetValue() => Value = 0;

    public override string ToString() => Value.ToString();
}
```

Ukážka kódu 37 Model *IntParameter* - Try

```
public class StringParameter : Parameter
{
    public StringParameter() { }

    public StringParameter(string value) => Value = value;

    public string Value { get; set; } = "";

    public override object GetValue() => Value;

    public override void SetValue(object value) => Value = (string)value;

    public override void ResetValue() => Value = "";

    public override string ToString() => Value;
}
```

Ukážka kódu 38 Model *StringParameter* - Try

```
public class ListParameter : Parameter
{
    public ListParameter() { }

    public ListParameter(IList<int> value) => Value = value;

    public IList<int> Value { get; set; } = new List<int>();

    public override object GetValue() => Value;

    public override void SetValue(object value) => Value = (IList<int>)value;

    public override void ResetValue() => Value = new List<int>();

    public override string ToString() => $"[{string.Join(", ", Value)}]";
}
```

Ukážka kódu 39 Model *ListParameter* - Try

7.3.10 Zhodnotenie

Webová aplikácia obsahuje vlastnú implementáciu módu Try nástroja Try .NET. Pomocou tejto webovej aplikácie bola demonštrovaná možnosť využitia módu Try v aplikácií na vyhodnocovanie testov vypracovaných študentami. V rámci implementácie sú dodržané procesy originálnej implementácie. To znamená, že užívateľský kód z editora monaco je prekladaný na strane servera a je spúšťaný na strane klienta vo webovom prehliadači. Spúšťanie užívateľského kódu v užívateľovom webovom prehliadači má výhodu zníženia záťaže servera. Ďalšou výhodou z bezpečnostného hľadiska je to, že užívateľský kód nemusí byť analyzovaný pretože beží mimo behové prostredie serverov.

V rámci webovej aplikácie je možné implementovať všetky potrebné funkcionality na vytvorenie plnohodnotnej aplikácie slúžiacej na vyhodnotenie testov vypracovaných študentmi.

ZÁVER

Táto práca sa venovala opisu technológie Try .NET od spoločnosti Microsoft. V rámci práce sú popísané jednotlivé módy, ktoré technológia poskytuje. Následne došlo k popísaniu procesov prekladu a spustenia užívateľského kódu jednotlivých módov spolu s príkladmi použitia.

V teoretickej časti je obecne popísaný webový štandard WebAssembly (kapitola číslo 1). Popis tohto štandardu zahrnuje obecné informácie o webovom štandarde, spôsoby tvorenia WebAssembly kódu, výkon a jeho použitie. Ďalej sú popísané technológie priamo využívané technológiou Try .NET, ktoré sú .NET, Blazor a Roslyn. Popis Try .NET-u (kapitola číslo 5) obsahuje obecné informácie, informácie o módoch tejto technológie. Popis je zakončený porovnaním módov Hosted a Try.

Praktická časť sa zaoberá možnosťami využitia technológie Try .NET v rámci výuky .NET technológií (kapitola číslo 6). Možnosť využitia je demonštrovaná vytvorením dokumentácie na výuku programovacieho jazyka C# pomocou módu Hosted. V čase písania práce nebol jeden z módov (mód Try) technológie Try .NET verejne dostupný. Preto bola vytvorená vlastná implementácia tohto módu, ktorá kopírovala procesy použité v originálnej implementácií. Vlastná implementácia a verejne dostupný mód Hosted sú následne použité na vytvorenie dvoch webových aplikácií, slúžiacich na demonštráciu možného využitia technológie Try .NET na automatickú kontrolu testov vypracovaných študentami (kapitola číslo 7.2.). Časti jednotlivých webových aplikácií sú taktiež popísané v rámci praktickej časti (kapitola číslo 7.2. a 7.3.).

Webové aplikácie demonštrujúce automatickú kontrolu testov implementujú minimálne potrebnú funkcionalitu na overenie konceptu využitia technológie Try .NET na daný účel. Na ich potencionálne využitia v produkčnom prostredí je nutné doplniť funkcionality, ako je autentifikácia / autorizácia užívateľov alebo uchovávanie dát v databázovom systéme.

ZOZNAM POUŽITEJ LITERATÚRY

[1] WebAssembly. *WebAssembly.* [Online] [Dátum: 27. 7 2020.] https://webassembly.org/.

[2] WebAssembly Concepts - WebAssembly | MDN. *MDN Web Docs.* [Online] [Dátum: 27. 7 2020.] https://developer.mozilla.org/en-US/docs/WebAssembly/Concepts.

[3] WebAssembly Community Group. *WebAssembly Community Group.* [Online] [Dátum: 27. 7 2020.] https://www.w3.org/community/webassembly/.

[4] GitHub - WebAssembly/wabt: The WebAssembly Binary Toolkit. *GitHub.* [Online] [Dátum: 27. 7 2020.] https://github.com/webassembly/wabt.

[5] Christophe Alladoum. Understanding WebAssembly. *Sophos.* [Online] 2018. [Dátum: 27. 7 2020.] https://www.sophos.com/en-us/medialibrary/PDFs/technical-papers/understanding-web-assembly.pdf.

[6] Introduction | The AssemblyScript Book. *The AssemblyScript Book.* [Online] [Dátum: 27. 7 2020.] https://www.assemblyscript.org/introduction.html.

[7] GitHub - WebAssembly/binaryen: Compiler infrastructure and toolchain library for WebAssembly. *GitHub.* [Online] [Dátum: 27. 7 2020.] https://github.com/WebAssembly/binaryen.

[8] David, Herrera, a iní. WebAssembly and JavaScript Challenge. *Sable McGill.* [Online] 14. 3 2018. [Dátum: 27. 7 2020.] http://www.sable.mcgill.ca/publications/techreports/2018-2/techrep.pdf.

[9] Asanovic, Krste, a iní. The Landscape of Parallel Computing Research. [Online] 18. 12 2006. [Dátum: 27. 7 2020.] https://www2.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.pdf.

[10] Use Cases - WebAssembly. *WebAssembly.* [Online] [Dátum: 27. 7 2020.] https://webassembly.org/docs/use-cases/.

[11] Musch, Marius, a iní. New Kid on the Web: A Study on the Prevalence of WebAssembly in the Wild. *Institute of System Security - TU Braunschweig.* [Online] 2019. [Dátum: 27. 7 2020.] https://www.sec.cs.tu-bs.de/pubs/2019a-dimva.pdf.

[12] .NET architectural components | Microsoft Docs. *Microsoft Docs.* [Online] 23. 8 2017. [Dátum: 27. 7 2020.] https://docs.microsoft.com/en-us/dotnet/standard/components.

[13] .NET Core official support policy. *Microsoft .NET.* [Online] [Dátum: 27. 7 2020.] https://dotnet.microsoft.com/platform/support/policy/dotnet-core.

[14] .NET Framework official support policy. *Microsoft .NET.* [Online] [Dátum: 27. 7 2020.] https://dotnet.microsoft.com/platform/support/policy/dotnet-framework.

[15] Mono Releases | Mono. *Mono.* [Online] 2020. [Dátum: 27. 7 2020.] https://www.mono-project.com/docs/about-mono/releases/.

[16] Lander, Richard. Introducing .NET 5 | .NET Blog. *Microsoft .NET Blog.* [Online] 6. 5 2019. [Dátum: 27. 7 2020.] https://devblogs.microsoft.com/dotnet/introducing-net-5/.

[17] .NET. *Wikipedia.* [Online] [Dátum: 27. 7 2020.] https://cs.wikipedia.org/wiki/.NET.

[18] ALBAHARI, Joseph a Ben ALBAHARI. *C# 7.0 in a nutshell. 7th edition.* Sebastopol : O´Reilly, 2018. ISBN 978-1-491-98765-0.

[19] NAGEL, Christian. *C# 2008: programujeme profesionálně.* Brno : Computer Press, 2009. ISBN 978-80-251-2401-7.

[20] Nagel, Christian. *Professional C# 7 and .Net Core 2.0.* Indianapolis, Indiana : Wrox, a Wiley brand, 2018. ISBN 978-1-119-44927-0.

[21] Overview of the .NET Framework | Microsoft Docs. *Microsoft Docs.* [Online] 30. 3 2017. [Dátum: 27. 7 2020.] https://docs.microsoft.com/en-us/dotnet/framework/get-started/overview.

[22] .NET Core. *Wikipedia.* [Online] [Dátum: 27. 7 2020.] https://en.wikipedia.org/wiki/.NET_Core.

[23] Home | Mono. *Mono.* [Online] 2020. [Dátum: 27. 7 2020.] https://www.mono-project.com/.

[24] Mono (software). *Wikipedia.* [Online] [Dátum: 27. 7 2020.] https://en.wikipedia.org/wiki/Mono_(software).

[25] About Mono | Mono. *Mono.* [Online] 2020. [Dátum: 27. 7 2020.] https://www.mono-project.com/docs/about-mono/.

[26] Himschoot, Peter. *Blazor revealed : building web applications in .NET.* Berkeley, CA : Apress L.P, 2019. ISBN 978-1-4842-4343-5.

[27] Roth, Daniel. Blazor WebAssembly 3.2.0 now available | ASP.NET Blog. *Microsoft ASP.NET Blog.* [Online] 19. 5 2020. [Dátum: 27. 7 2020.] https://devblogs.microsoft.com/aspnet/blazor-webassembly-3-2-0-now-available/.

[28] ASP.NET Core Blazor modelech hostování | Microsoft Docs. *Microsoft Docs.* [Online] 19. 5 2020. [Dátum: 27. 7 2020.] https://docs.microsoft.com/cs-cz/aspnet/core/blazor/hosting-models?view=aspnetcore-3.1.

[29] Blazor. *Wikipedia.* [Online] Wikimedia Foundation. [Dátum: 27. 7 2020.] https://en.wikipedia.org/wiki/Blazor.

[30] Roslyn (compiler). *Wikipedia.* [Online] [Dátum: 27. 7 2020.] https://en.wikipedia.org/wiki/Roslyn_(compiler).

[31] GitHub - dotnet/roslyn: The Roslyn .NET compiler provides C# and Visual Basic languages with rich code analysis APIs. *GitHub.* [Online] [Dátum: 27. 7 2020.] https://github.com/dotnet/roslyn.

[32] The .NET Compiler Platform SDK (Roslyn APIs) | Microsoft Docs. *Microsoft Docs.* [Online] 10. 10 2017. [Dátum: 27. 7 2020.] https://docs.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/.

[33] Roslyn Overview · dotnet/roslyn Wiki · GitHub. *GitHub.* [Online] [Dátum: 27. 7 2020.] https://github.com/dotnet/roslyn/wiki/Roslyn-Overview.

[34] Sole, Alessandro Del . *Roslyn Succinctly.* Morrisville, NC : Syncfusion, Inc., 2016.

[35] GitHub - dotnet/try: Try .NET provides developers and content authors with tools to create interactive experiences. *GitHub.* [Online] [Dátum: 27. 7 2020.] https://github.com/dotnet/try.

[36] Cannot embed using the provided iframe on https://dotnet.microsoft.com/platform/try-dotnet · Issue #298 · dotnet/try · GitHub. *GitHub.* [Online] 24. 6 2019. [Dátum: 27. 7 2020.] https://github.com/dotnet/try/issues/298.

[37] Provide documentation for using wasm "mode" · Issue #382 · dotnet/try · GitHub. *GitHub.* [Online] 19. 8 2019. [Dátum: 27. 7 2020.] https://github.com/dotnet/try/issues/382.

[38] Support Bloggers and Code Samples · Issue #561 · dotnet/try · GitHub. *GitHub.* [Online] 26. 10 2019. [Dátum: 27. 7 2020.] https://github.com/dotnet/try/issues/561.

[39] Monaco Editor. *Microsoft.* [Online] 2020. [Dátum: 27. 7 2020.] https://microsoft.github.io/monaco-editor/.

[40] Visual Studio Code. *Wikipedia.* [Online] [Dátum: 27. 7 2020.] https://en.wikipedia.org/wiki/Visual_Studio_Code.

[41] GitHub - microsoft/monaco-editor: A browser based code editor. *GitHub.* [Online] [Dátum: 27. 7 2020.] https://github.com/microsoft/monaco-editor.

[42] .NET In-Browser Tutorial - Step 1: Intro. *Microsoft .NET.* [Online] [Dátum: 27. 7 2020.] https://dotnet.microsoft.com/learn/dotnet/in-browser-tutorial/1.

[43] Electron (software framework). *Wikipedia.* [Online] [Dátum: 27. 7 2020.] https://en.wikipedia.org/wiki/Electron_(software_framework).

ZOZNAM POUŽITÝCH SYMBOLOV A SKRATIEK

API	Application Programming Interface
CRUD	Create, Read, Update and Delete
DLL	Dynamic-Link Library
DOM	Document Object Model
GUI	Graphical User Interface
IoT	Internet of Things.
SPA	Single-Page Application
UI	User Interface
URL	Uniform Resource Locator
VM	Virtual Machine
VS	Visual Studio
WYSIWIG	What You See Is What You Get

ZOZNAM OBRÁZKOV

ZOZNAM TABULIEK

ZOZNAM UKÁŽOK KÓDU

ZOZNAM PRÍLOH

Príloha P I: CD

PRÍLOHA P I: CD

CD obsahuje:

- Zdrojové kódy dokumentácie určenej na výuku jazyka C#: priloha.zip / docSrc
- Zdrojové kódy webovej aplikácie (mód Hosted): priloha.zip / srcHosted
- Zdrojové kódy webovej aplikácie (mód Try): priloha.zip / srcTry
- Diplomová práca vo formáte pdf: fulltext.pdf