Univerzita Hradec Králové

Fakulta Informatiky a management

Webová aplikace pro sledování chyb

Bakalářská práce

Autor: Tomáš Malík

Studijní obor: Aplikovaná informatika

Jilemnice březen 2020

Prohlášení:

Prohlašuji, že jsem bakalářskou práci zpracoval samostatně a s použitím uvedených zdrojů.

.....

V Jilemnici, dne 31.3.2020 Tomáš Malík

Poděkování:

Tímto bych rád poděkoval svému vedoucímu bakalářské práce doc. Mgr. Tomáši Kozlovi, Ph.D. za metodické vedení, odborné připomínky a cenné rady ke zpracování této bakalářské práce.

Anotace

Cílem práce je popsat problematiku tvorby webové aplikace pro sledování chyb a vytvořit ukázkovou aplikaci s využitím platformy .NET. Práce je rozdělena do několika tematických celků, kde první se věnuje již existujícím řešením pro sledování chyb. Druhý celek se zaměřuje na analýzu a návrh aplikace. Cílem třetího celku je představit platformu .NET a její součásti, které mohou být následně využity při vývoji. Poslední část práce se zabývá popisem implementace ukázkové aplikace, její databáze a následné i otestováním toho, zda aplikace funguje správně.

Annotation

Title: Web application for crash reporting

The goal of this Bachelor Thesis is to describe the problem of creating a web application for bug reporting and to create a sample application using the .NET platform. The Bachelor Thesis is divided into several thematic sections, where the first one deals with already existing solutions for bug reporting. The second section focuses on the analysis and design of the application. The aim of the third section is to introduce the .NET platform and its components, which can be then used in development of sample application for bug reporting. The last part of the Bachelor Thesis describes the implementation of the application itself, its database and testing of whether the application works correctly.

Obsah

Seznam obrázků

Seznam grafů

Seznam zdrojových kódů

1. Úvod

Nejdůležitějším pilířem každého softwaru je jeho funkcionalita. Ta by měla řešit nějaký problém či potřebu uživatele. V současné době je také neméně důležitá vizuální a estetická stránka aplikace, která určuje pohodlnost při jejím používání. Další velice podstatnou součástí softwaru je jeho zabezpečení. To dnes představuje pro řadu uživatelů daleko významnější faktor než vizuální stránka. Všechny výše zmíněné pilíře však nejsou důležité, pokud daná aplikace není stabilní a obsahuje chyby, které uživateli brání v jejím aktivním používání. Pokud bude uživatel takovýmto chybám vystaven příliš často, tak to může vést k tomu, že nebude aplikaci důvěřovat a raději zvolí konkurenční řešení. Takováto špatná zkušenost může mít za důsledek např. negativní hodnocení v obchodech s aplikacemi. To může následně odradit další potencionální uživatele od toho, aby aplikaci vyzkoušeli sami.

Proto platí, že před vypuštěním na veřejnost je nutné aplikaci náležitě otestovat. Právě v procesu testování mohou pomoci různá automatizovaná řešení, která budou informace o chybách shromažďovat na jednom centrálním místě. Vývojáři softwaru se tak budou moci věnovat přímo opravám daných chyb, nikoliv jejich shromažďování a třídění. Drtivou většinu chyb testování pravděpodobně odhalí, ale může se také stát, že nějaká chyba projde testovacím procesem bez odhalení a narazí na ní až uživatel při používání aplikace. Z tohoto důvodu je vhodné sledování chyb v aplikaci provádět i v produkčním prostředí a zachytit tak zbylé chyby.

Cílem práce je popsat problematiku tvorby webové aplikace pro sledování chyb a vytvořit ukázkovou aplikaci s využitím platformy .NET. Nejdříve by mělo dojít k popsání metod, které lze ke sběru reportů o chybách využít, a uvedení příkladů již existujících řešení, které je využívají. Dále by měl být proveden návrh a analýza budoucí aplikace pro sledování chyb. Důležitou částí je také představení platformy .NET, která bude použita k implementaci ukázkové aplikace. Závěr práce by měl popsat právě samotnou implementaci aplikace a otestování její funkčnosti.

2. Nástroje pro sledování chyb

Existuje celá řada nástrojů, které je možné k záznamu chyb použít. Celkově je lze rozdělit do dvou kategorii. První představuje takovou skupinu, kde uživatel sledované aplikace musí sám přistoupit do systému pro sledování chyb a manuálně nahlásit, že se v aplikaci vyskytla chyba. Druhou kategorií jsou automatizované systémy, kde v případě chyby dojde k automatickému odeslání hlášení do systému pro sledování chyb.

2.1. Manuální řešení

Manuální řešení je možno využít především v situacích, kde lze předpokládat zkušeného uživatele, který alespoň rámcově tuší, jak program uvnitř funguje. Důležitým předpokladem je, aby uživatel dokázal detailně popsat, jak se aplikace v případě chyby zachovala a pokud možno jak tuto chybu replikovat. Tedy popsat krok za krokem, co v aplikaci dělal, když chyba nastala. Toto řešení je často využíváno open-source aplikacemi, kde může být např. z finančních důvodu problém řádně otestovat software. Díky tomu je tato část přenechána odborné veřejnosti, která software vyzkouší a případné chyby řádně nahlásí. Ty pak mohou být odstraněny před tím, než je software distribuován k běžným uživatelům.

2.1.1 GitHub Issues

Představitelem nástrojů pro manuální sběr chyb je např. služba GitHub Issues, která je součástí aplikace pro verzování zdrojového kódu GitHub. Jednotlivá hlášení chyb jsou v systému zobrazena jako seznam. Ke každé nahlášené chybě mohou přihlášení uživatelé přidávat komentáře a vést tak diskuzi o tom, jakým způsobem by bylo možné daný problém vyřešit.

K jednotlivým problémům lze přiřadit štítky, díky kterým lze problémy rozdělit do různých skupin. Tyto štítky mohou např. uvádět, jaké části aplikace se daný problém týká. Jednotlivé chyby lze přiřadit k milníkům, které představují klíčové body při vývoji aplikace. Díky tomu je možné sledovat, v které verzi aplikace bude daná chyba vyřešena. Další významnou funkcionalitou je možnost k dané chybě přiřadit osobu či více osob, které budou mít za úkol daný problém vyřešit. [1]

Protože je GitHub určený především pro open-source projekty, tak v rámci GitHub Issues nabízí možnost využití tzv. pull-requestů. Ty fungují tak, že další vývojáři mohou nabídnout změnu přímo ve zdrojovém kódu aplikace a vyřešit tak daný problém. Tvůrce aplikace pak již pouze ověří, že navrhovaná změna skutečně řeší problém a zda je řešení přijatelné v rámci dané aplikace. Pokud jsou obě tyto podmínky splněny, tak jen změnu potvrdí a její kód je automaticky zařazen do zdrojového kódu aplikace. [1]

2.2. Automatizovaná řešení

Automatizace procesu hlášení chyb přináší řadu výhod oproti manuálnímu sběru, ale zároveň sebou nese i nežádoucí efekty. Hlavní výhodou je to, že není nutná přímá interakce uživatele se systémem pro sběr chybových hlášení. To především znamená, že není nutné, aby byl uživatel pro nahlášení chyby odborníkem. V určitých situacích ani nemusí mít tušení, že došlo k odeslání hlášení. Zjednodušeně funguje celý proces tak, že jsou sebírána data o aplikaci, jejím pádu a další informace (např. operační systém na kterém byla aplikace spuštěna). Největší nevýhodu oproti manuálnímu sběru představuje nutnost mít systém, který bude na straně klienta zaznamenávat chyby a následně zajistí přenos na server do aplikace pro jejich sběr. V ideálním případě by měl přenos na server probíhat pomocí nějakého standardizovaného komunikačního protokolu. Díky tomu je následně možné tvořit klientskou část systému v jakékoliv technologii, která bude pro danou sledovanou aplikaci vhodná.

2.2.1 Crashlytics

Asi nejznámějším řešením pro automatizovaný sběr chybových hlášení je aplikace Crashlytics. Původně byla vyvinuta společností Fabric, která je vlastněná firmou stojící za sociální sítí Twitter. Později byla však aplikace odkoupena softwarovým gigantem Google a integrována do platformy Firebase. Ta představuje systém pro podporu vývoje mobilních aplikací. [2]

Bohužel Crashlytics trpí řadou slabin. Tou největší je nemožnost vyhledání konkrétního reportu o chybě. Při velkém množství zobrazených dat také trpí pomalým uživatelským rozhraním, a to především ve webových prohlížečích, které nejsou založené na vykreslovacím jádře Chromium. Poslední výraznou slabinou Crashlytics oproti dalším podobným řešením je fakt, že obsahuje pouze omezené množství grafů a statistik. Ty by v ideálním případě měly zobrazovat více užitečných informací. [2]

2.2.2 AppCenter

Představuje cloudové[1] řešení od společnosti Microsoft. AppCenter se neomezuje pouze na sledování chyb, ale nabízí navíc celou řadu nástrojů pro podporu vývoje především mobilních aplikací. Pomocí AppCenter lze aplikace sestavovat[2], testovat, provádět analýzy, zvolit způsob jejich distribuce ke klientům a především sbírat chybová hlášení. Celý proces diagnostiky AppCenter funguje tak, že při pádu aplikace je na klientském zařízení vytvořen log soubor. Ten je při dalším spuštění aplikace odeslán spolu s oznámením o chybě do AppCenter. [3]

Je nutné poznamenat, že celý AppCenter lze propojit pouze s aplikacemi vytvořenými pomocí nějaké technologie platformy .NET (např. WPF, UWP, Xamarin atd.). Velkou výhodou AppCenter je fakt, že veškerá funkcionalita kolem sběru chybových hlášení a spoustu dalších funkcí je nabízeno kompletně zdarma. [3]

2.2.3 Bugsee

Další alternativou Crashlytics je aplikace Bugsee. Firma na svých webových stránkách uvádí [4], že jsou jedinou platformou pro sledování chyb, která podporuje možnost nahrávání videozáznamu používání aplikace uživatelem s frekvencí 10 snímků za sekundu. K nahrávání dění na obrazovce využívají proprietární řešení, které podle prohlášení nezpomaluje běh uživatelského rozhraní a dalších částí aplikace.

Bugsee také zaznamenává logy při pádu aplikace. Ty následně dokáže zkombinovat a synchronizovat s pořízeným záznamem obrazovky, díky čemuž lze přesně zjistit, kdy během používání aplikace daný problém nastal. Další zajímavou funkcí je schopnost zaznamenávat všechny HTTP dotazy a odpovědi aplikace. To umožňuje vývojáři diagnostikovat, kde v průběhu síťové komunikaci mohlo dojít k chybě. Tyto dotazy též lze synchronizovat s pořízeným videem pádu aplikace. Při vytvoření chybového hlášení Bugsee také zaznamená stav systému, jako např. zda byl zapnutý Bluetooth čip. Díky tomu lze lépe diagnostikovat, co mohlo vést k pádu aplikace. [4]

[1] Aplikace provádějící část své logiky na logické rozsáhlé síti serverů
[2] Vytvoření spustitelné verze aplikace ze zdrojového kódu

3. Analýza a návrh aplikace

Požadavky na aplikaci rozdělujeme do dvou druhů. První druh je nazýván funkční a patří do něj popis všeho, co by aplikace měla umět. Jako příklady takovýchto požadavků lze uvést: „Aplikace musí uživateli umožnit hodnotit jednotlivé výrobky.“ nebo „Aplikace musí uživateli zobrazit shrnutí o pohybu peněz na jeho bankovním účtu.“

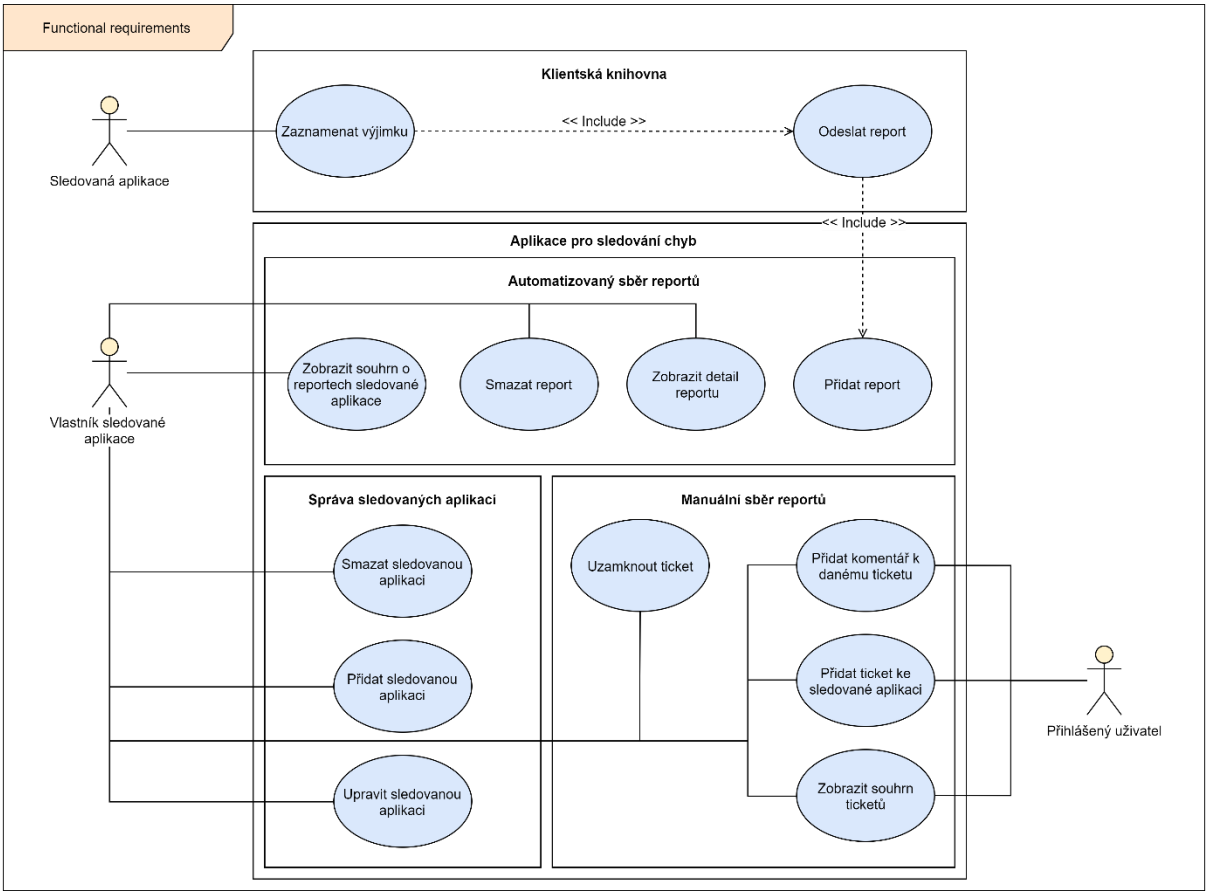
Druhý druh požadavků nazýváme non-funkční a jejich cílem je definovat vlastnosti aplikace. Požadavky mohou být různé jako např. členění zdrojového kódu, výkon, bezpečnost, spolehlivost, dostupnost nebo rozšiřitelnost aplikace. Konkrétní příklady takovýchto požadavků mohou být následující: „Aplikace musí být dostupná i v případě, že by na ní v jeden moment chtělo přistupovat o 100 % více uživatelů, než je běžné.“ nebo „Aplikace musí být připravena v případě potřeby jednoduše migrovat na jinou databázi.“

3.1. Funkční požadavky na aplikaci

Aplikace pro sledování chyb by vývojáři měla poskytovat centrální místo pro sběr hlášení o chybách, které nastaly v jeho sledovaných aplikacích. Jednotlivé funkční požadavky na aplikaci jsou rozděleny do čtyř následujících druhů služeb:

- Správa sledovaných aplikací
- Klientská knihovna
- Automatizovaný sběr reportů
- Manuální sběr reportů

Funkční požadavky na aplikaci lze reprezentovat mnoha různými způsoby. Nejjednodušší je textová podoba, kdy jednotlivé funkční požadavky představují řádky v seznamu. Dalším a mnohem lepším způsobem je tvorba diagramu případů užití neboli use-case. Jednotlivé případy užití popisují sekvenci interakcí mezi systémem a externím aktérem, které vyústí v takový výsledek, jenž má aktérovi poskytnout nějakou hodnotu. [5] Aktérem může být například osoba, zařízení nebo software, který komunikuje s daným systémem. Diagramy případů užití jsou velmi užitečné ve vizualizaci toho, se kterými systémy aktéři komunikují a jaké akce na těchto systémech provádí. Případ užití pak reprezentuje samostatnou aktivitu, kterou aktér provede, aby dosáhl výsledku.



Obr. 1 Diagram případů užití aplikace; Zdroj: autor

3.1.1 Správa sledovaných aplikací

Tento druh služeb obsahuje všechny případy užití, které slouží k manipulaci sledovaných aplikací. Aktér (vlastník sledované aplikace) si zde může do systému přidat aplikaci, kterou plánuje sledovat. Při vytváření si vlastník nastaví základní informace o aplikaci. Následně je vygenerován nový záznam do databáze systému a dojde k nastavení sledované aplikace tak, aby k ní bylo možné přiřazovat nová chybová hlášení. Vlastník také může upravovat informace o jednotlivých sledovaných aplikací. Posledním případem užití tohoto modulu je možnost smazat sledovanou aplikaci.

3.1.2 Klientská knihovna

Tuto knihovnu si může vývojář připojit do své sledované aplikace. Ta následně provádí sběr chybových hlášení automaticky. V případě, že ve sledované aplikaci dojde k výjimce, tak je touto knihovnou zaznamenána a odeslána do aplikace pro sledování chyb, kde je následně zpracována tak, aby byla přiřazena ke správné aplikaci. Vývojář sledované aplikace je tedy nucen provést prvotní nastavení knihovny, aby celý systém

věděl, ke které aplikaci dané hlášení přiřadit. K tomuto rozlišení slouží API[3] klíč, který je každé sledované aplikaci přiřazen při jejím vytvoření. Knihovna pak sama naváže spojení s aplikačním serverem a odešle hlášení. Pokud pro danou vývojovou platformu neexistuje klientská knihovna, tak je možné využít i webové API, přes které je možné přijmout správně formátovaný požadavek určitého typu souboru. V tomto případě si ale vývojář musí implementovat zaznamenávání a odesílání hlášení sám.

3.1.3 Automatizovaný sběr reportů

Tato skupina služeb obsahuje případy užití související s automatizovaně odeslanými hlášeními. Případ užití pojmenovaný „Přidat report“ souvisí s předchozí skupinou služeb klientské knihovny. Je poslední fází přidání reportu, kdy je přijatý report zpracován a v případě, že je nalezena příslušná sledovaná aplikace, tak je přidán záznam o chybovém hlášení do aplikace pro sledování chyb.

Další případ užití umožňuje vývojáři zobrazit souhrn o chybových hlášení jednotlivých sledovaných aplikací. Vývojář zde vidí, kolik bylo přijato chybových hlášení v posledních 24 hodinách a kolik již bylo přijato hlášení celkem. Tato skupina služeb také obsahuje případ užití pro odstranění konkrétního reportu, díky čemuž má vývojář možnost odstranit hlášení, které nejsou relevantní vzhledem k dalšímu vývoji sledované aplikace. Posledním případem užití je zobrazení detailního popisu přijatých hlášení jednotlivých sledovaných aplikací. Tento detailní popis obsahuje celou řadu informací od data, kdy bylo hlášení přidáno přes verzi operačního systému, na kterém k chybě došlo, až po samotný záznam výpisu chyby.

3.1.4 Manuální sběr reportů

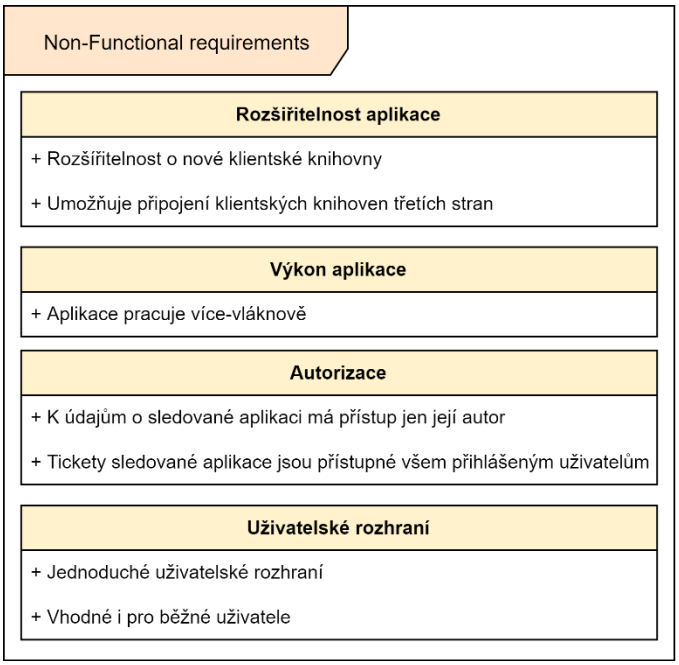
Poslední kategorie služeb aplikace se zabývá případy užití, které souvisí s manuálním sběrem reportů. Přihlášení uživatele mají možnost přidávat nové tickety v případě, že během práce s aplikací narazili na nějaký problém, či mají návrh na přidání nějaké funkce do aplikace. Přihlášení uživatelé si mohou též zobrazit všechny tickety, které byly k dané sledované aplikaci přidány ostatními uživateli. Toho lze využít, pokud vývojář potřebuje otestovat novou verzi své aplikace v rámci betatestování. Může tak ke komunikaci o jednotlivých problémech, které nastaly, využít právě tyto služby.

[3] Programové rozhraní umožňující dalším aplikacím přistupovat k funkcím dané aplikace

Ke každému ticketu mohou přihlášení uživatelé přidávat komentáře. A to v případech, kdy chtějí o daném problému diskutovat, případně pokud narazili na stejný problém a sledovaná aplikace se chová jinak, než je v ticketu popisováno, či naopak jiná chyba způsobuje stejný problém. Pokud vývojář problém popsaný v daném ticketu vyřešil, nebo došel k závěru, že navrhovaná funkcionalita nebude do aplikace přidána, tak uzavře ticket. Tím vzniká situace, při které se tento ticket změní do stavu pouze pro čtení a již k němu nelze přidávat nové komentáře.

3.2. Non-funkční požadavky na aplikaci

Nejdůležitějším z non-funkčních požadavků na aplikaci je ten, aby bylo možné jednoduše vytvářet klientské knihovny pro další vývojové platformy. Aplikace by měla být dokonce připravena na situaci, kdy budou klientské knihovny vytvářet vývojáři třetích stran pro účely svých vlastních systémů. Další non-funkční požadavek zahrnuje nutnost, aby celá aplikace fungovala více vláknově a dokázala tak obsloužit velké množství uživatelů současně. Aplikace je koncipována tak, aby ke všem údajům o svých sledovaných aplikacích měl přístup pouze jejich vývojář. Výjimku tvoří tickety, které jsou přístupné všem přihlášeným uživatelům. Grafické rozhraní aplikace by mělo být navrhnuto tak, aby bylo přístupné i uživatelům, kteří běžně nepracují s nástroji pro podporu vývoje aplikací.



Obr. 2 Diagram non-funkčních požadavků na aplikaci; Zdroj: autor

4. Platforma .NET

.NET je framework[4] sloužící k tvorbě softwarových aplikací a je vyvíjen společností Microsoft. Tento framework se skládá z nástrojů, programovacích jazyků a knihoven. Všechny zmíněné součásti může vývojář využít k urychlení své práce. První verze frameworku se objevila v únoru 2002 [6] a byla určená pouze k tvorbě aplikací pro operační systém Microsoft Windows. Později se však začaly objevovat neoficiální a následně i oficiální implementace frameworku pro operační systémy mimo ekosystém společnosti Microsoft.

.NET se nezaměřuje na tvorbu softwaru pro konkrétní oblast, ale jedná se o univerzální řešení, které lze použít pro tvorbu celé řady různých aplikací podobně jako jazyk Java od společnosti Oracle. Mezi typy aplikací, které lze pomocí frameworku .NET vyvíjet patří desktopové aplikace pro Windows, multiplatformní mobilní aplikace, webové aplikace, software pro IoT[5], aplikace využívající umělou inteligenci a videohry v enginech[6] Unity3D, Godot a CryEngine. [7]

4.1. CIL a CLR

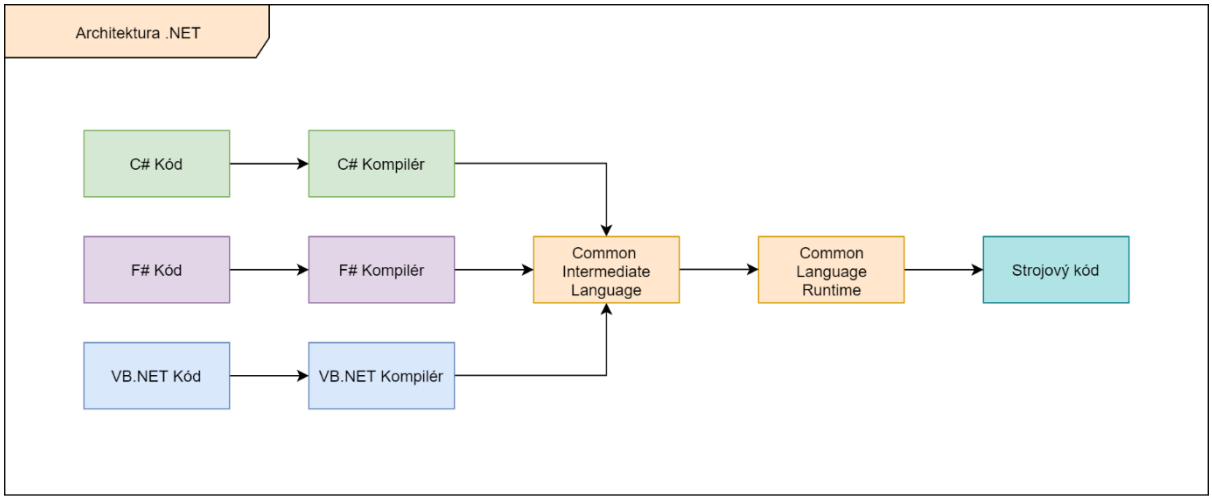
Kompilátor je počítačový program, který funguje tak, že vezme celý lidsky čitelný kód programu a převede ho do podoby čitelné pro počítač dané platformy. Tento proces nazýváme kompilace. Zkompilovaný kód nazýváme strojový kód. Program je následně distribuován ke koncovým uživatelům právě v podobě tohoto kódu. Díky tomu může program fungovat opravdu rychle. Tento postup je však vykoupen jednou zásadní nevýhodou, kterou představuje fakt, že aplikace musí být zkompilována pro každou platformu, kde má být později spustitelná. Kompilátor využívají např. jazyky C a Rust.

Interpretr oproti kompilátoru zastává zcela odlišnou filozofii. Zjednodušeně pracuje tak, že vezme řádek lidsky čitelného zdrojového kódu, provede ho a pokud neskončí chybou, tak se přesune na následující řádek. Program je tedy nutné šířit v podobě lidsky čitelného kódu. To má výhodu v tom, že je spustitelný na každé platformě, kde existuje interpretr, ale zároveň je kód jednoduše odcizitelný a interpretr

[4] Sada nástrojů pro zjednodušení vývoje softwaru
[5] Množina fyzických zařízení, jako např. domácí spotřebiče propojená přes počítačovou síť
[6] Komplexní software zahrnující mnoho nástrojů pro tvorbu videoher

nemůže provádět optimalizace podobně jako kompilátor, který zná celý program. Interpretr využívají např. jazyky Python a JavaScript.

.NET patří mezi platformy vyžívající tzv. virtuální stroj. Ten kombinuje výhody kompilátoru a interpretru. Pojmem bytekód označujeme jazyk specifický pro daný virtuální stroj. Tento jazyk je svojí strukturou velmi podobný assembleru[7] a vzniká zkompilováním lidsky čitelného kódu, díky čemuž má výhody kompilátoru jako mikro-optimalizace. Virtuální stroj je pak podobný interpretru, ale místo překladu lidsky čitelného kódu překládá bytekód. Díky této vlastnosti lze program spustit na jakékoliv platformě, pro kterou existuje daný virtuální stroj. Výsledný program je pak šířen v podobě bytekódu. Na platformě .NET se bytekód nazývá CIL (Common Intermediate Language) a virtuální stroj CLR (Common Language Runtime). [8]



Obr. 3 Diagram architektury .NET; Zdroj: autor; Zpracováno dle: [8]

4.2. Programovací jazyky pro .NET

V rámci platformy .NET lze využít hned několik programovacích jazyků, které mají rozdílnou syntaxi a zastávají odlišná paradigmata[8]. Těmito jazyky jsou C#, F# a VisualBasic.NET. Velkou výhodu platformy .NET je to, že jednotlivé projekty mohou být vytvořeny v jakémkoliv z výše uvedených jazyků a následně pak využity v projektu používajícím jiný jazyk z rodiny .NET. A to vše díky tomu, že se jednotlivé jazyky kompilují do společného bytekódu.

[7] Nízko-úrovňový programovací jazyk, který ovládá počítač skrz jednoduché instrukce
[8] Udává způsob, jakým bude daný program vytvořen

4.2.1 C#

Primárním programovacím jazykem platformy .NET je C#, což má za následek to, že jeho vývoji se Microsoft věnuje nejvíce. C# je především objektově orientovaným[9] jazykem, ale v posledních verzích implementoval i prvky spojené s funkcionálním programováním[10]. Svou syntaxí je velmi podobný jazykům Java a C. Oproti těmto jazykům má však celou řadu syntaktických prvků, které umožňují jednodušší psaní přístupových metod[11] veřejných atributů nebo tzv. rozšiřujících metod, které se volají na proměnné, upraví její hodnotu a výsledek mohou ulož zpět do dané proměnné. Tyto rozšiřující metody, lze jednoduše řetězit za sebou.

Dále je C# striktně typovaným jazykem, což znamená, že každá proměnná musí mít konkrétní typ. Pro představu je možné uvést opačný přístup v podobě dynamického typování, které odhaduje typ proměnné na základě toho, co je v ní uložené. Představitelem dynamického typování je např. jazyk JavaScript. I když je striktní typování časově náročnější proces, tak přináší výhodu v tom, že velké množství chyb je možné odhalit už při kompilaci zdrojového kódu do bytekódu.

Ve výchozím nastavení má C# automatickou správu operační paměti, díky čemuž se programátor nemusí obávat problémů způsobených nevhodnou správou paměti. Lze povolit tzv. nebezpečný kód v případech, kdy je nutné pracovat s nízko-úrovňovými knihovnami, avšak s tím programátor přichází o výhody spojené s automatickou správou paměti.

```
public class Robot
{
    private readonly Beacon mBeacon;

    public Coordinates Position { get; set; }

    public Robot(Coordinates position)
    {
        mBeacon = new Beacon(Color.Blue);
        Position = position;
    }
}
```

Ukázka kódu 1 Jazyk C#; Zdroj: autor;

[9] Přístup, který se snaží z dat vytvářet objekty, které mají určité vlastnosti a schopnosti
[10] Přístup založený na vyhodnocování funkcí a snažící se vyhýbat globálnímu stavu aplikace
[11] Metody pro nastavování a získávání hodnoty proměnné daného objektu

4.2.2 F#

F# je především funkcionální programovací jazyk, ale obsahuje i řadu prvků objektově orientovaného programování. Sdílí velké množství vlastností s jazykem C# jako např. striktní typování a automatickou správu paměti. Ač je F# striktně typovaný, tak používá tzv. implicitní typování. To znamená, že programátor nemusí nutně určovat typ dané proměnné, ale její typ se pevně určí podle první přiřazené hodnoty.

Zvláštností jazyka F# je, že ve výchozím stavu jsou všechny typy tzv. immutable. To značí, že jejich hodnotu nelze změnit. Lze však definovat proměnnou i jako mutable, díky čemuž lze její hodnot měnit, jako v případě objektově orientovaných jazyků. Další zvláštností jazyka je, že není nutné kompilovat celý projekt, ale klidně samostatné řádky kódu ve stylu skriptovacích jazyků jako Python.

```
let names = [ "Ana"; "Felipe"; "Emillia"]

names
|> List.iter (fun name -> printfn "Hello %s" name)
```

Ukázka kódu 2 Jazyk F#; Zdroj: autor; Zpracováno dle: [9]

4.2.3 VisualBasic.NET

Třetím jazykem, který je možné v rámci vývoje na platformě .NET využít je VisualBasic.NET. Ten vznikl jako přímý pokračovatel staršího jazyka VisualBasic, ale není s ním kompatibilní. Významným specifikem tohoto jazyka je jeho syntaxe, která se má co nejvíce podobat lidsky čitelnému textu. Příkladem může být zápis větvení programu, kde jsou jednotlivé větve oddělené výrazy jako *If-Then*, *Else*, *End If* místo složených závorek jako je to běžné v jazycích podobných C#.

```
Dim names As New List(Of String)({
    "Ana",
    "Felipe",
    "Emilia"
})

For Each name In names
    Console.WriteLine($"Hello {name}")
Next
```

Ukázka kódu 3 Jazyk VB.NET; Zdroj: autor; Zpracováno dle: [9]

4.3. Běhová prostředí .NET

Za dobu existence .NET platformy vznikla celá řada implementací, od oficiální verze s uzavřeným zdrojovým kódem pro operační systém Windows až po komunitní verze s otevřeným zdrojovým kódem, které fungují na rozličných operačních systémech. Tyto rozdílené implementace byly vzájemně nekompatibilní, což vedlo k tříštění celé platformy. V současné době se však objevují snahy o standardizaci celé platformy.

4.3.1 .NET Framework

Je nejstarší implementací platformy .NET. Má uzavřený zdrojový kód a aplikace napsané v .NET Frameworku lze spustit pouze na operačním systému Windows. První verze byla vydána v roce 2002. Jejím cílem bylo usnadnit tvorbu webových aplikací a aplikací pro osobní počítače. Technologie WindowsForms a ASP.NET WebForms, které byly obsaženy v prvních verzích frameworku se při návrhu uživatelského rozhraní spoléhaly na WYSIWYG[12] editor. Tento přístup však vývojáře značně omezoval v tom, jak budou jejich aplikace vypadat po vizuální stránce, a tak pozdější verze frameworku přinesly technologie WPF, UWP a ASP.NET MVC, které umožňují definovat uživatelské prostředí pomocí XML nebo HTML. To vede k téměř neomezeným možnostem v upravování vzhledu aplikace. Další verze frameworku přinesly podporu např. asynchronního programování[13] a dotazovacího jazyka LINQ, který usnadňuje práci s kolekcemi[14]. [10]

4.3.2 Mono

Jedná se o open-source[15] implementaci platformy .NET, která začala vznikat téměř hned po oznámení .NET Frameworku a první vývojová verze byla uvolněna již v roce 2001. [11] Mono bylo už od počátku vyvíjeno jako multiplatformní alternativa, která měla .NET vývojářům umožnit tvořit aplikace pomocí jejich oblíbených technologii. Hlavní premisa Mona je být 1:1 implementací .NET Frameworku. To znamená, že všechny knihovny a funkce obsažené v .NET Frameworku jsou obsažené i v Monu a zároveň by měly vykazovat úplně stejné chování, a to i v případě chybových hlášení.

[12] Editor, ve kterém se tvoří uživatelské rozhraní vizuálně pomocí myši a umisťování jednotlivých ovládacích prvků do prostředí okna aplikace
[13] Programování aplikace tak, že dokáže fungovat na více procesorových vláken zároveň
[14] Složitá datová struktura, která uchovává více data stejného typu
[15] Program s veřejně dostupným zdrojovým kódem

Mono se neomezuje pouze na desktopové operační systémy, jako Windows, macOS a Linux, ale funguje mobilních systémech Android, iOS a dokonce i dalších platformách jako tvOS, watchOS, Sony Playstation 4, XboxOne a BSD. [12] Xamarin společnost stojící za vývojem Mona vytvořila stejnojmennou platformu Xamarin, která umožňuje jednoduše vytvářet multiplatformní mobilní aplikace.

Kromě mobilních aplikací našlo Mono také uplatnění ve videoherním průmyslu. C# je v kombinaci s Monem využíván celou řadou herních enginů jako skriptovací jazyk pro tvorbu herní logiky. Jedná se např. o enginy Unity3D [13], Godot [14] a CryEngine [15].

Ve chvíli, když začala společnost Xamarin a Mono získávat větší popularitu mezi korporátními klienty, tak se Microsoft rozhodl společnost koupit. S ní přirozeně koupil i Mono a nástroje pro tvorbu mobilních aplikací. [16]

4.3.3 .NET Core a .NET 5

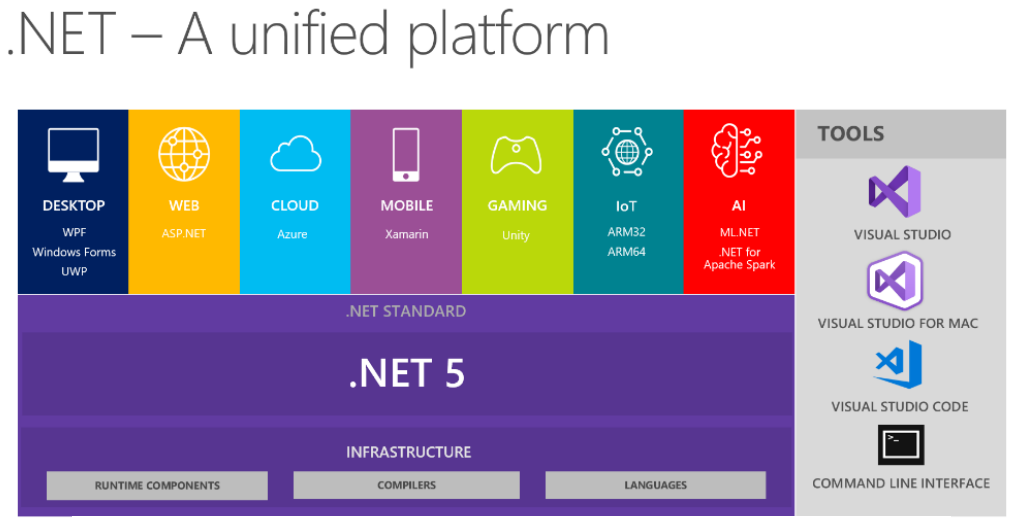
Ač se po odkoupení společnosti Xamarin dostalo do rukou firmy Microsoft multiplatformní Mono, tak se firma rozhodla zvolit jinou strategii a vytvořit úplně novou implementaci .NET platformy. Tuto novou multiplatformní implementaci pojmenovali .NET Core. V raných verzích umožňovala tvořit pouze jednoduché webové a konzolové aplikace[16], ale v následujících vydáních došlo k implementaci řady dalších technologii. V době psaní nejnovější verze .NET Core 3.0 umožňuje vytvářet také desktopové aplikace pro Windows pomocí WindowsForms, WPF a pokrývá tak velkou většinu případů užití .NET Frameworku a Mona. V tuto chvíli však .NET Core nelze použít pro tvorbu mobilních aplikací.

Microsoft se aktuálně snaží veškerý vývoj směřovat k .NET Core a doporučuje ho použít pro všechny nové projekty. Tato situace vede ke stavu, kdy .NET Framework a Mono přestávají dostávat nové funkce. Příkladem může být osmá verze jazyka C#, která má na .NET Core plnou podporu, ale v .NET frameworku jen částečnou a Mono nemá dokonce ani plnou podporu předchozí sedmé verze C#. [17] [18]

Celý přesun od starších implementací platformy .NET směrem k .NET Core má vyvrcholit v roce 2020 projektem nazvaným .NET 5. Ten představuje vizi Microsoftu pro další hlavní verze .NET Core, jejichž nejpodstatnější cílem je pokrýt všechny případy užití

[16] Aplikace běžící v příkazové řádku operačního systému

.NET Frameworku a Mona, které v současné době dávají smysl. Zároveň by měl odstranit všechny nedostatky předchozích implementací. .NET 5 má být v budoucnosti jedinou implementací platformy .NET. Bude jí možné použít pro tvorbu desktopových aplikací, webových aplikací, cloudových aplikací, mobilních aplikací, IoT aplikací, AI aplikací[17] a videoher. [19]



Obr. 4 Vize .NET 5; zdroj: [19]

4.3.4 .NET Standard

Aktuálním řešením problému přenositelnosti kódu mezi jednotlivými implementacemi je specifikace .NET Standard. Ta udává, jaké knihovny musí implementace obsahovat, aby byl kód označený danou verzí specifikace .NET Standard spustitelný. Nejnovější verzi specifikace 2.1 v době psaní podporuje .NET Core a Mono. Poslední verzi, která je podporována .NET Frameworkem je verze 2.0. [20]

Příkladem využití této specifikace může být případ, kdy vývojář tvoří mobilní aplikaci pomocí Mona a webový backend[18] v .NET Core. Většina business logiky[19] je stejná pro obě aplikace. Díky .NET Standardu vytvoří vývojář jednu knihovnu obsahující tuto logiku a tu pak připojí jak k mobilní aplikaci, tak k backendu. Není tedy nutné, aby daný kód psal dvakrát pro dvě různé implementace platformy .NET. Význam specifikace .NET Standard pravděpodobně zanikne spolu s vydáním .NET 5.

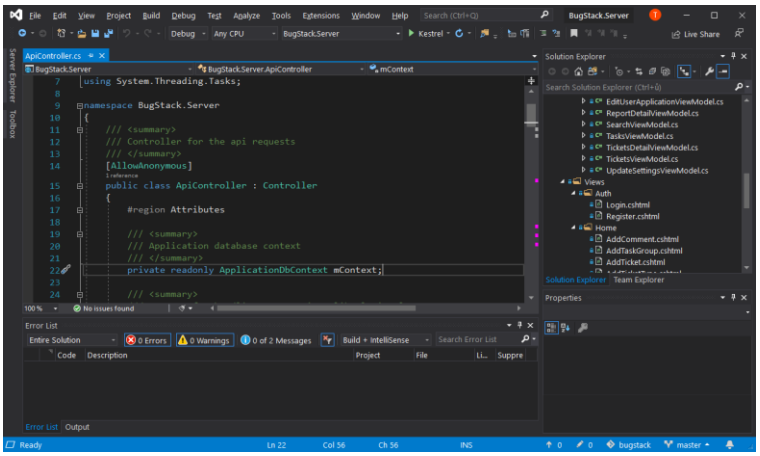
[17] Aplikace využívající umělou inteligenci
[18] Část aplikace běžící na serveru a starající se o logiku aplikaceí
[19] Poskytuje funkce aplikace a stará se o zpracování dat

5. .NET IDE a nástroje

Do doby před příchodem .NET Core existovala pouze dvě hlavní prostředí pro vývoj .NET aplikací. Pro plnohodnotný .NET Framework to bylo Visual Studio a pro Mono to byl MonoDevelop (Xamarin Studio). S první verzí .NET Core byly vydané nástroje pro CLI[20], díky čemuž je možné sestavovat .NET aplikace bez závislosti na jakémkoliv IDE[21] nebo textovém editoru. Vydání CLI nástrojů vedlo také ke vzniku nových IDE a rozšíření pro textové editory.

5.1. Visual Studio

Hlavní představitelem IDE v .NET světě je Visual Studio, které je vyvíjené přímo společností Microsoft a určité časti jeho kódu jsou vytvořené právě za pomoci platformy .NET. [21] Kromě základních nástrojů pro vývoj, ladění a testování aplikací obsahuje také funkce, které nejsou v jiných IDE příliš běžné. Jednou z takových je umělou inteligencí poháněná technologie IntelliCode, která se snaží vývojáři chytře napovídat části kódu během jeho psaní. Další užitečnou součástí je funkce LiveShare, která umožní dalším vývojářům hromadně editovat kód v reálném čase. Kromě aplikací pro .NET, lze do Visual Studia doinstalovat oficiálně podporované součástí pro vývoj aplikací v jazycích jako Python, C++, Node.js, R a Javascript/Typescript. Kromě oficiálních součástí lze doinstalovat i celou řadu rozšíření třetích stran. Visual Studio bohužel funguje jen na operačním systému Windows. Nedávno však vznikla i macOS verze Visual Studia, která je však pokračovatelem Xamarin Studia nikoliv Visual Studia z Windows. [22]



Obr. 5 Ukázka prostředí IDE Visual Studio; Zdroj: autor

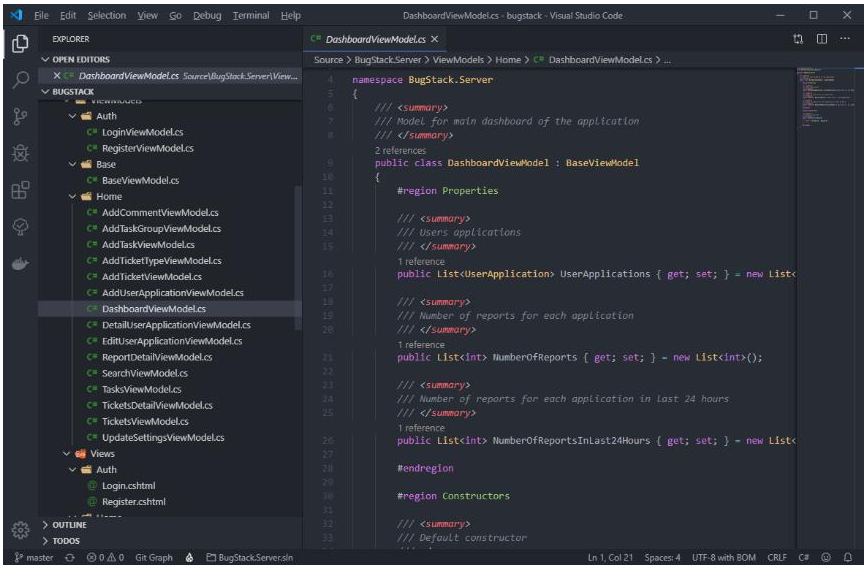
[20] Nástroje pro příkazovou řádku (Command Line Interface)
[21] Vývojové prostředí zahrnující nástroje pro tvorbu a testování aplikací.

5.2. Visual Studio Code

Je textový editor zdrojových kódu, který taktéž pochází od společnosti Microsoft. Oproti IDE má méně integrovaných součástí a funkce jako ladění programů je nutné do textového editoru doplnit pomocí rozšíření.

Zajímavostí je, že se jedná o fork[22] dalšího oblíbeného textového editoru Atom. Visual Studio Code je vytvořené jako front-end webová aplikace založená na frameworku Electron (Představuje odlehčenou verzi webového prohlížeče Chromium bez uživatelského rozhraní, který zobrazuje zvolenou webovou stránku, tak aby vypadala jako nativní aplikace[23] pro daný operační systém).

To, že je Visual Studio Code vytvořené za pomocí webových technologii má za důsledek, že i doplňky pro něj se tvoří právě za pomocí stejných nástrojů. Díky tomu, že webové technologie ovládá velké množství vývojářů, vznikla kolem editoru velká komunita a v současné době se jedná se o jeden z nejoblíbenějších programů pro vývoj softwaru. Microsoft oficiálně poskytuje rozšíření pro integraci jazyků jako C#, Python a Java, ale komunita postupem času vytvořila integraci pro téměř jakýkoliv programovací jazyk. Kromě integrací nových programovacích jazyků existují i rozšíření přidávající různou funkcionalitu, jako např. IntelliCode [23] a LiveShare [24] známé z plnohodnotného Visual Studia.

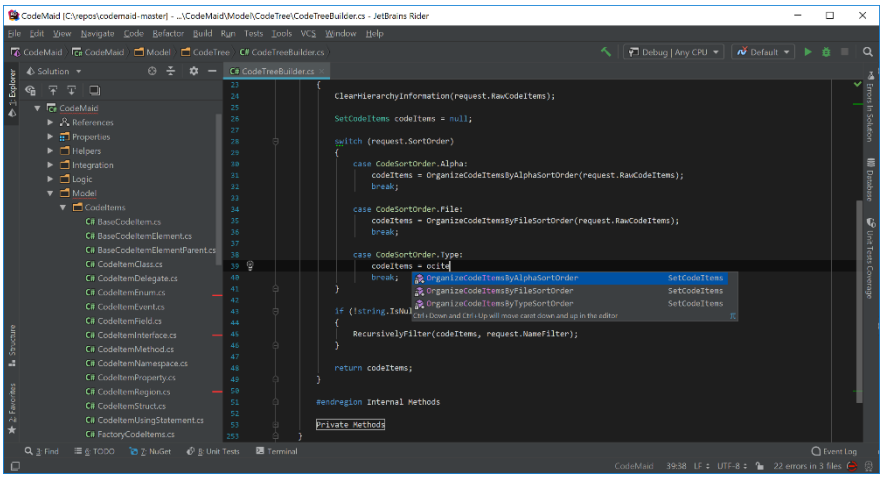


Obr. 6 Ukázka prostředí Visual Studio Code; Zdroj: autor

[22] Program založený na zdrojovém kódu jiného programu
[23] Aplikace vytvořená pro konkrétní operační systém

5.3. Jetbrains Rider

Jedním z IDE, které vzniklo na základě vydání .NET CLI nástrojů je JetBrains Rider od autorů známých IDE jako Intelij IDEA, PyCharm, WebStorm nebo CLion. V podstatě se jedná o upravenou verzi Intelij IDEA, která nahrazuje nástroje pro vývoj v programovacím jazyce Java nástroji pro vývoj .NET Core aplikací. Obsahuje také oblíbené funkce z rozšíření ReSharper pro Visual Studio, které je též vyvíjeno společností JetBrains. Nevýhodou tohoto IDE je fakt, že JetBrains nenabízí žádnou verzi zdarma, jako tomu je v případě některých dalších nástrojů této společnosti.



Obr. 7 Ukázka prostředí JetBrains Rider; Zdroj: [25]

5.4. NuGet

NuGet je balíčkovací systém sloužící k vytváření a sdílení balíčků kódů. To je užitečné především, pokud projekt využívá kód třetích stran. Vývojář tak nemusí veškerou správu těchto externích součástí provádět sám, ale přenechá to balíčkovacímu systému, který se postará o instalaci, aktualizaci či případnou odinstalaci balíčků.

Balíčky lze distribuovat lokálně jako „.nupkg“ soubory, ale v praxi se nejčastěji využívá repositářů[24]. Microsoft poskytuje repositář nuget.org, kam mohou vývojáři umisťovat balíčky, které budou dostupné všem. Pokud by vývojář nechtěl balíčky umisťovat na servery třetích stran, tak existují implementace NuGet serveru s otevřeným zdrojovým kódem. Příkladem může být BaGet [26] nebo Sleet [27], které může vývojář nainstalovat a provozovat na vlastním serveru.

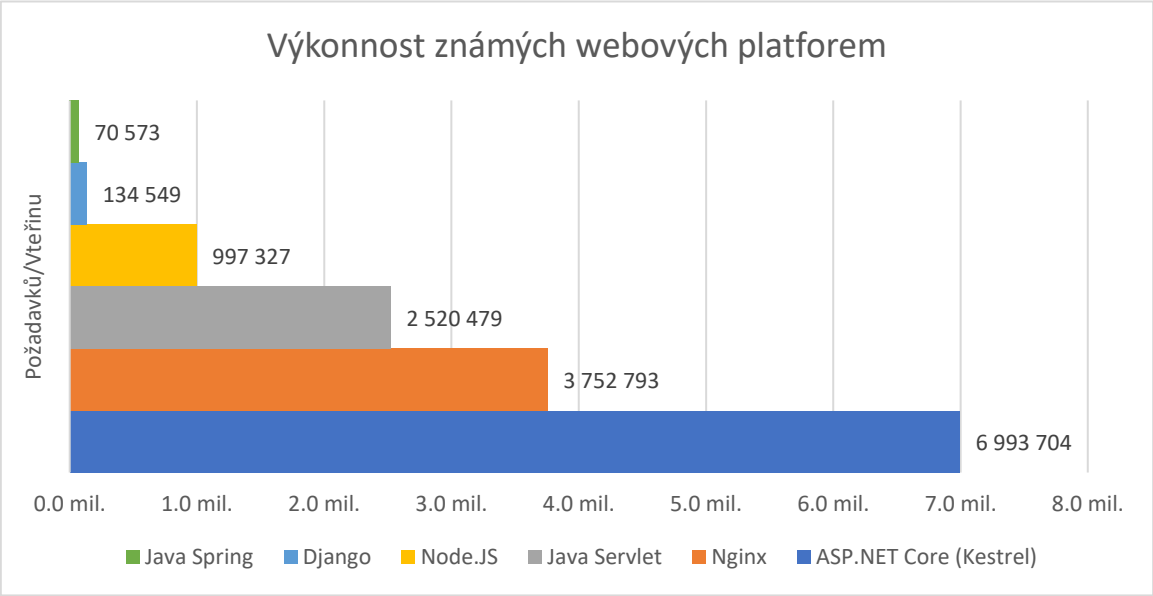
[24] Místo na vzdáleném serveru, kde je daný balíček kódu uložen

6. ASP.NET Core

Oficiálním řešením pro tvorbu webových aplikací na platformě .NET je framework nazvaný ASP.NET. Podporuje různé typy aplikací od MVC[25] až po jednoduchá webová API[26]. V aktuální verzi ASP.NET Core umožňuje hostovat vytvořené aplikace i na operačním systému Linux. Původní verze postavená na .NET Frameworku fungovala pouze na operačním systému Windows Server.

6.1. Kestrel

Spolu s ASP.NET Core byl vydán i nový multiplatformní webový server[27] nazvaný Kestrel. Původní webový server určený pouze pro operační systém Windows Server se nazýval IIS (Internet Information Services). I když byl ASP.NET Core vyvinut pro Kestrel, tak jsou tyto aplikace kompatibilní i s IIS.



Graf 1 Výkonnost známých webových platforem; Zdroj: autor; Zpracováno dle: [28]

V červenci 2019 proběhl test, jehož úkolem bylo změřit, kolik je webový server schopen zpracovat požadavků dané kategorie za vteřinu. Nejjednodušší z testovaných kategorii byly požadavky, kde server na žádost klienta odeslal zprávu obsahující text „Hello World“. ASP.NET Core aplikace běžící v Kestrelu na Linuxu se umístila na 10. místě z 357 testovaných s 6,99 milióny zpracovaných požadavků za vteřinu. [28]

[25] Architektura webové aplikace rozdělující její součásti do tří typů komponent (modely, pohledy a kontroléry)
[26] Rozhraní, které umožňuje ostatním aplikacím jednoduše využívat funkcionalitu dané aplikace
[27] Server zpracovávající požadavky HTTP protokolu (Hyper Text Transfer Protocol)

6.2. Typy šablon projektů

ASP.NET Core nabízí hned celou řadu šablon projektů. Ty se liší podle toho, jakou architekturu bude vyvíjená aplikace používat. Může se jednat o aplikaci se kterou bude uživatel komunikovat pomocí webového rozhraní, ale může také jít pouze službu, která bude poskytovat strojově zpracovatelná data dalším aplikacím.

6.2.1 MVC

Nejpoužívanější šablonou projektu v ASP.NET je MVC. Ta podle stejnojmenné architektury rozděluje součásti aplikace do tří typů komponent. Prvním jsou modely, které slouží jako třídy reprezentující jednotlivé entity v rámci aplikace. Často jsou vytvářeny tak, aby sloužily jako předpis databázové tabulky pro danou entitu. Druhým typem komponent MVC jsou pohledy, které představují uživatelské rozhraní, pomocí kterého bude uživatel s aplikací pracovat. Jsou vytvářeny tak, aby neměly tušení, že se v aplikaci nachází nějaké modely. Díky tomu, pokud dojde ke změně pohledu není nutné měnit model. Nejdůležitějším typem komponent v celé aplikaci jsou kontroléry, které obstarávají aplikační logiku a slouží jako most mezi modely a pohledy.

6.2.2 MVC WebAPI

Je odnož šablony ASP.NET MVC. Funguje v podstatě stejně, ale s tím rozdílem, že místo pohledů v podobě HTML stránek vrací data ve strojově zpracovatelném formátu. Nejčastěji používanými formáty jsou JSON a XML. JSON však v poslední době získává čím dál větší popularitu, protože při jeho použití může být velikost výsledného souboru dat až poloviční. Zároveň je jeho strojové zpracování ve většině případů značně rychlejší než zpracování XML.

Takovýto přístup nachází využití především při tvorbě mobilních a webových front-endových aplikací. Ty pak nemusí obsahovat žádnou složitou business logiku. Nemusí také např. řešit synchronizaci databází napříč různými zařízeními, ale všechna data přijímají a odesílají právě serveru s API rozhraním, který se stará o správu dat za ně. Přijatá data front-endové aplikace pouze zpracují a zobrazí uživateli. K tvorbě webových front-endových aplikací konzumující serverová API se často používají technologie jako Vue, Svelte a React.

6.2.3 Razor Pages

Jednou z nových šablon ASP.NET Core aplikací je Razor Pages. Její hlavní výhoda spočívá v jednoduchosti. Oproti ASP.NET MVC není nutné rozdělovat části aplikace do několika různých komponent, ale je možné mít seskupenou logiku dané webové stránky v jednom souboru a její vzhled v druhém souboru. Logika stránky je tvořena jako běžná C# třída a vzhled je zhotoven jako HTML soubor rozšířený o speciální znaky jazyka Razor, které umožňují dynamicky dosadit obsah do webové stránky. Každá dvojice souborů pak představuje jednu stránku ve výsledné aplikaci. V případě nutnosti je samozřejmě možné části dané stránky rozdělit do více různých souborů. Výsledná struktura projektu pak může vypadat tak, že obsahuje pouze soubory Program.cs, Startup.cs, které vytvoří a nastaví ASP.NET aplikaci a soubory daných webových stránek ve složce Pages. [29]

6.2.4 Endpoint-Routing

Endpoint-Routing je další nová šablona pro vytváření webových aplikací v ASP.NET Core. Nachází využití především v případech, kde není nutné používat ostatní plnohodnotné typy aplikací dostupné v ASP.NET Core. Příkladem užití může být testovací aplikace, která slouží jako dočasné webové API pro vývoj mobilní aplikace předtím, než bude vytvořen plnohodnotný backend v ASP.NET Core MVC API. Celý proces funguje tak, že se k danému URL vzoru přiřadí jednouchá metoda, která zpracuje HTTP požadavek a odešle odpověď. S tím, jak se funkcionalita ASP.NET Core postupně rozděluje do samostatných modulů, tak lze do Endpoint-Routing projektu doplnit i součásti z MVC či dalších typů ASP.NET projektů.

7. Popis implementace aplikace

Před vytvořením projektu je nutné mít nainstalované .NET Core SDK[28]. Projekt byl vytvářen ve verzi 2.2, ale je možné použít i v době psaní nejnovější verzi 3.0. S SDK se zároveň nainstalují i .NET CLI nástroje. Pokud bude vývoj probíhat ve Visual Studiu nebo Jetbrains Rider, tak je nutné mít nainstalované i tyto IDE. Visual Studio v nejnovější verzi 2019 nabízí možnost nainstalovat .NET Core SDK během jeho instalace.

7.1. Struktura aplikace a založení projektů

První krok při tvorbě aplikace pro sledování chyb představuje vytvoření samotného projektu ze šablony a její následná úprava pro potřeby projektu.

7.1.1 Struktura aplikace

Největší celky kódu na platformě .NET se nazývají projekty. Tyto projekty jsou vytvářeny tak, že se buď jedná o spustitelnou aplikaci nebo o knihovnu kódu, která muže být použita spustitelnými aplikacemi, ale samostatně ji nelze spustit.

Aplikace pro sledování chyb se bude dohromady skládat ze tří projektů. Dvou knihoven a jednoho spustitelného projektu. Spustitelný projekt bude představovat samotná MVC aplikace. První knihovna bude zastupovat klientskou knihovnu, kterou mohou vývojáři použít ve své aplikaci. Druhá knihovna bude sloužit ke sdílení kódu, který bude potřebovat jak projekt aplikace, tak projekt klientské knihovny.

7.1.2 Založení projektů aplikace

Založení nového .NET Core projektu může probíhat dvěma způsoby, a to graficky nebo pomocí CLI nástrojů. Grafický průvodce vytvoření projektu je nabízen pouze v prostředí Visual Studia a Jetbrains Rider, pokud bude projekt vytvářen bez použití jednoho z těchto prostředí, tak je nutné použití CLI nástrojů.

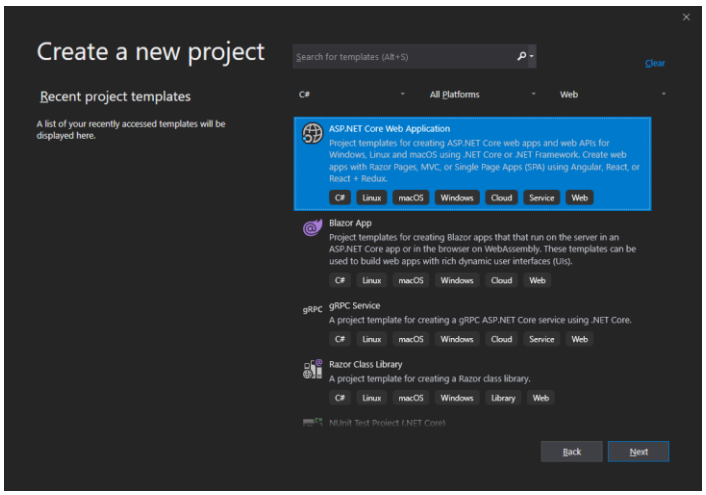
V případě Visual Studia je po spuštění nutné zvolit položku „Create a new project“, následně vybrat šablonu „ASP.NET Core Web Application“ se štítkem „C#“, protože aplikace bude vyvíjena v jazyce C#. V dalším kroku se vyplní název a umístění projektu. Jméno projektu bude: „BugStack.Server“. V posledním kroku dává Visual Studio na výběr typ ASP.NET projektu. Aplikace bude využívat architekturu MVC, proto je nezbytné zvolit

[28] Sada nástrojů pro vývoj aplikací v daném programovacím prostředí

šablonu „Web Application (Model-View-Controller)“. Po potvrzení formuláře Visual Studio vytvoří tzv. „Solution“, což je soubor, který umožňuje otevřít několik projektů zároveň. V něm vytvoří nový projekt podle zvolených kritérii v předchozích krocích a otevře ho.

Následné přidání knihoven kódu do „Solution“ probíhá tak, že se v okně pro zobrazení souborů klikne pravý tlačítkem myši na „Solution“ a zvolí se položka „Add“ a v ní „New project“, což vyvolá stejný dialog, jako při vytváření hlavního projektu. Jedinou změnou je šablona, kterou bude tentokrát „Class Library (.NET Standard)“. Visual Studio pak automaticky do Solution přidá nově vytvořený projekt. Tímto způsobem je nutné vytvořit knihovnu pro klientskou aplikaci, která bude pojmenována „BugStack.Client“ a následně i knihovnu sdíleného kódu nazvanou „BugStack.Core“.

Poslední věc, která zbývá je propojení projektů tak, aby daný projekt mohl využívat funckionality dalšího projektu. V okně pro zobrazení souborů stačí otevřít projekt aplikace a pravým tlačítkem myši kliknout na „Dependencies“. Následně ze zobrazeného menu vybrat „Add reference“. Poté se objeví okno s výběrem projektů, kde je potřeba zvolit projekt knihovny, a nakonec potvrdit tlačítkem OK. Takto musí dojít k připojení knihovny „BugStack.Core“ k projektům „BugStack.Server“ a „BugStack.Client“.



Obr. 8 Ukázka tvorby ASP.NET Core projektu v prostředí Visual Studio; Zdroj: autor

Druhým způsobem, jak vytvořit projekt je použití CLI nástrojů. Tento postup se týká především vývojářů používajících Visual Studio Code, ale samozřejmě je možné vygenerovaný projekt spustit i ve Visual Studiu nebo Jetbrains Rider. Vytvoření probíhá velmi podobně, ale místo grafického průvodce se vše píše v podobě příkazů, díky čemuž

je celý proces o mnoho rychlejší a přehlednější. Stačí se v terminálu navigovat do složky, kde se projekt bude nacházet, a napsat následující sadu příkazů:

```
dotnet new solution --name BugStack

dotnet new mvc --name BugStack.Server
dotnet new classlib --name BugStack.Core
dotnet new classlib --name BugStack.Client

dotnet sln add BugStack.Server/BugStack.Server.csproj
dotnet sln add BugStack.Core/BugStack.Core.csproj
dotnet sln add BugStack.Client/BugStack.Client.csproj

dotnet add BugStack.Server/BugStack.Server.csproj reference BugStack.Core/BugStack.Core.csproj
dotnet add BugStack.Client/BugStack.Client.csproj reference BugStack.Core/BugStack.Core.csproj
```

Ukázka kódu 4 Tvorba ASP.NET Core projektu pomocí CLI; Zdroj: autor

Jednotlivé bloky příkazů pak provedou následující akce. Nejdříve je vytvořeno řešení neboli tzv. „Solution“, který je možné otevřít ve Visual Studiu nebo Jetbrains Rider. Druhý blok vytvoří ASP.NET Core MVC aplikaci „BugStack.Server“, sdílenou knihovnu kódu „BuhStack.Core“ a klientskou knihovnu „BugStack.Client“. Třetí blok slouží k přidání jednotlivých projektů do „Solution“. V poslední bloku dojde k propojení projektů tak, aby aplikace a klientská knihovna mohli využívat kód definovaný ve sdílené knihovně.

7.1.3 Základní struktura projektů

Protože projekt „BugStack.Server“ byl vygenerován jako MVC projekt, tak obsahuje celou řadu souborů, kdežto projekty knihoven „BugStack.Core“ a „BugStack.Client“ po vygenerování obsahují pouze jeden soubor s prázdnou třídou.

Nejdůležitějším souborem projektu „BugStack.Server“ je „Program.cs“, obsahující hlavní metodu, která je spuštěna po zapnutí aplikace. V této metodě dojde k inicializaci ASP.NET Core aplikace podle zvoleného nastavení. Ve výchozím stavu aplikace převezme nastavení ze souboru „Startup.cs“, který je taktéž vytvořen během generování nového projektu.

„Startup.cs“ obsahuje výchozí konstruktor, jeden veřejný read-only[29] atribut „Configuration“ a dvě veřejné metody „ConfigureServices“ a „Configure“. V konstruktoru

[29] Atribut jehož hodnot lze nastavit pouze v konstruktoru třídy

dojde k nastavení atributu „Configuration“, který představuje kolekci jednoduché struktury klíč/hodnota. Jednotlivé hodnoty nastavení lze načíst např. z konfiguračního souboru JSON nebo XML uloženého na úložišti zařízení. Metoda „ConfigurationServices“ slouží k inicializaci všech závislostí, které bude aplikace používat podle techniky Dependency-Injection[30]. V metodě „Configure“ dochází k nastavení samotné ASP.NET Core aplikace. Lze zde např. nastavit, že aplikace bude využívat MVC komponenty nebo jak bude vypadat výchozí směrovací tabulka aplikace.

Dále projekt obsahuje tři adresáře a to „Models“, „Views“ a „Controllers“. První adresář by měl sloužit pro bussiness logiku aplikace. Druhý adresář je určený pro soubory (tzv. „pohledy“) související s uživatelským rozhraním aplikace. Třetí adresář slouží pro kód řídící jednotlivé dotazy, které aplikace obdrží.

Poslední významný prvek, který je obsažen ve vygenerovaném projektu představuje adresář „wwwroot“, který slouží pro statické soubory jako např. kaskádové styly, soubory JavaScriptu, fonty atd. Na tyto soubory se pak lze odkazovat přímo v pohledech aplikace.

7.2. Řídící logika v aplikaci

Nejdůležitějším prvkem v rámci MVC aplikace jsou kontroléry, které řídí příjem dotazů od uživatele. Celý proces funguje tak, že se uživatel ve svém webovém prohlížeči dotáže na nějakou konkrétní stránku v aplikaci. Aplikace po přijetí dotazu podle vzoru URL vybere správný kontrolér, který obstarává požadovanou množinu funkcí. Z té je následně vybrána přesně ta, na kterou se uživatel dotázal a následně dojde k jejímu spuštění. Funkce kontroléru využije data z modelů, které zpracuje. Poté vybere vhodný pohled, do kterého zpracovaná data dosadí. Výsledek celého procesu je předán zpět uživateli.

V rámci aplikace mohou být data z kontrolérů do pohledů předávána několika různými způsoby. Proto je nutné pochopit, jak přesně předávání funguje. První a nejdůležitější ze způsobů předávání dat je pomocí modelů (někdy také nazývaných „viewModely“). Jedná se o konkrétní objekt, díky čemuž pohled ví, jaké všechny atributy a metody obsahuje a může tak vývojáři napovídat při psaní kódu. Mezi další způsoby patří

[30] Technika umožňující vkládání závislostí do dané komponenty tak, že komponenta není závislá na konkrétní implementaci závislosti, díky čemuž ji není potřeba upravovat, pokud se závislost změní

konstrukty „ViewData“ a „ViewBag“. Oba fungují stejně, liší se jen ve způsobu zápisu. V podstatě jde o jednoduchý slovník, kde je k unikátnímu klíči přiřazena konkrétní hodnota. Bohužel IDE předem neví, jaké hodnoty budou do těchto slovníků přiřazeny, a tak nemůže vývojáři během psaní pohledu pomáhat. Z těchto důvodů by měly být využity pouze pro dodatečné informace pohledu. Příkladem může být titulek stránky. Posledním způsobem, jak předat data z kontroléru do pohledu, je pomocí konstruktu „TempData“. Funguje stejně jako „ViewData“ s tím rozdílem, že jeho stav přetrvává pouze do konce zpracování daného dotazu. Jedinou situaci, kde dává smysl ho využít je při přesměrování na jinou stránku.

7.2.1 Kontrolér pro autentifikaci

Úkolem kontroléru s názvem „Auth.cs“ je zpracovávat dotazy spojené s autentifikací[31] uživatele. V rámci konstruktoru jsou do něj injektovány služby „UserManager“, „SignInManager“ a „IMessageService“. První dvě zmiňované jsou třídy obsažené ASP.NET Core a jsou do Dependency-Injection kontejneru vloženy přímo samotným vývojovým frameworkem. „UserManager“ slouží ke správě uživatelů. Tedy jejich vytváření, mazání, úpravě a hledání. „SignInManager“ slouží k přihlášení a odhlášení uživatele z aplikace. Poslední injektovaná služba typu programového rozhraní „IMessageService“ je vytvořená v rámci aplikace pro sledování chyb a slouží jako obecný předpis pro službu, která bude zasílat textové zprávy uživateli. Může se jednat např. o emaily pro potvrzení registrace nebo o SMS s jednorázovým kódem pro dvoufázové ověření.[32]

Kontrolér obsahuje dvakrát metodu „Login“. První bude sloužit pro zobrazení přihlašovacího formuláře a je vyvolána, pokud je příchozí dotaz typu HttpGet[33]. Druhá metoda je vyvolána v případě, že se jedná o dotaz typu HttpPost[34] a slouží k přijmutí dat z formuláře. Ty zpracuje a pokusí se podle nich přihlásit uživatele. K procesu přihlášení tyto metody využívají injektované služby „UserManager“ a „SignInManager“.

[31] Proces ověření, že uživatel je ten, za koho se vydává, a to například pomocí hesla nebo biometrických údajů jako otisk prstu či obraz oční sítnice

[32] Bezpečnostní mechanismus, který pro přihlášení uživatele kromě hesla vyžaduje ještě alespoň jeden další bezpečnostní prvek, např. Zadání jednorázového kódu z příchozí SMS

[33] Metoda protokolu HTTP, které by měla sloužit pouze pro získávání dat

[34] Metoda protokolu HTTP, která by měla sloužit k odesílání dat na server

Obdobným způsobem jsou definovány i metody pro registraci uživatele s názvem „Register“, kde jedna slouží pro zobrazení registračního formuláře a druhá opět pro přijmutí dat z něj. Oproti metodám pro přihlášení uživatele je zde využito injektované služby „IMessageService“, která zde slouží k odeslání emailu pro ověření uživatele po jeho registraci.

Na metody spojené s registrací navazuje metoda „VerifyEmail“, která zpracovává příchozí dotaz na ověření emailové adresy po registraci a je typu HttpGet. Parametry metody obsahují ID uživatele[35], který chce svůj email ověřit a unikátní token, který byl vygenerován při registraci a slouží k samotnému ověření. Pokud ověření proběhne úspěšně, tak je uživateli dovoleno se přihlásit do aplikace.

Poslední metodou v rámci autentifikačního kontroléru je metoda „Logout“, která za pomoci služby „SignInManager“ odhlásí uživatele z aktuální relace a následně přesměruje na stránku s přihlašovacím formulářem.

7.2.2 Kontrolér pro hlavní funkce aplikace

Ústředním kontrolérem celé aplikace je „HomeController“. Ten poskytuje veškerou funkcionalitu pro ovládání aplikace skrz webové rozhraní. Pro využívání jeho metod musí být uživatel autentifikován.

Konstruktor kontroléru má injektované služby „UserManager“ a „ApplicationDbContext“. První z nich stejně jako autentifikačního kontroléru obstarává informace o uživateli. Druhá služba slouží k manipulaci s daty v databázi aplikace.

V rámci správy sledovaných aplikací kontrolér obsahuje hned několik metod. První z nich se jmenuje „Index“ a slouží k zobrazení pohledu se seznamem všech sledovaných aplikací daného uživatele. První z dvojce metod „AddUserApplication“ zobrazuje formulář pro přidání nové sledované aplikace. Druhá přijme data z tohoto formuláře, zpracuje je a pokud jsou validní, tak v databázi vytvoří nový záznam s danou aplikací. „Detail“ je neméně důležitou metodou, protože vrací pohled s podrobnými informacemi o konkrétní sledované aplikaci. Metoda „DeleteUserApplication“ je určena ke smazání zvolené sledované aplikace z databáze. Další dvě metody označené „EditUserApplication“ slouží k zobrazení formuláře pro

[35] Unikátní identifikátor daného uživatele

změnu informací o sledované aplikaci a následné zpracování dat z tohoto formuláře. Další podstatnou metodou je „ReportDetail“. Ta zobrazuje podrobný popis konkrétního reportu zvolené sledované aplikace. Metoda „DeleteReport“ umožňuje smazat konkrétní report z databáze.

Následující velkou skupinu metod představují ty, které spravují manuální reporty sledované aplikace. „TicketsList“ je metoda, která zobrazuje seznam manuálních reportů dané aplikace. Dvojce metod „AddTicketType“ vrací formulář pro přidání typu manuálního reportu a následně ho zpracuje. Úkolem metod s názvem „AddTicket“ je zobrazit formulář pro přidání nového manuálního reportu a následně daný report přidat do databáze aplikace. „TicketDetails“ vrací detailní výpis daného manuálního reportu. Dvojce metod „AddComment“ umožňují jakémukoliv přihlášenému uživateli přidat ke konkrétnímu manuálnímu reportu komentáře. Metoda „CloseTicket“ umožňuje vlastníkovi sledované aplikace daný manuální report uzamknou tak, že k němu již nelze přidávat další komentáře.

Poslední metoda kontroléru se nazývá „SearchAsync“ a vrací pohled zobrazující nalezené výsledky vyhledávání podle zadaného textového řetězce. Vyhledávácí algoritmus bere v potaz uživatelské aplikace a jejich reporty.

7.2.3 API kontrolér

Posledním kontrolérem v rámci aplikace pro sledování chyb je API kontrolér. Ten obsahuje jedinou metodu s názvem „AddReport“, která zpracovává příchozí HttpPost dotazy. Tato metoda bere jako vstupní parametr objekt z těla dotazu, který odpovídá typu „AddBugReportApiModel“. Jedná se o jednoduchou třídu, která obsahuje atributy spojené s reportem chybového hlášení. Jako odpověď vrací objekt „ApiResponse“, který obsahuje informaci o tom, zda byl dotaz úspěšný či nikoliv, případnou chybovou hlášku a také dobrovolný atribut s daty odpovědi. Ten není v rámci aplikace použit, avšak při dalším rozšiřování aplikace by mohl být zužitkován.

```
[HttpPost]
[Route(ApiRoutes.AddReport)]
public async Task<ApiResponse> AddReportAsync([FromBody]AddBugReportApiModel model)
{
    if (string.IsNullOrWhiteSpace(model.ApplicationApiKey))
    {
        return new ApiResponse()
        {
            ErrorMessage = "Please provide all data needed for bug report model creation"
        };
```

```
    }

    var application = mContext
        .UserApplications
        .FirstOrDefault(x => x.ApiKey.ToString() == model.ApplicationApiKey);

    if (application == null)
    {
        return new ApiResponse()
        {
            ErrorMessage = "No application with provided API key wasn't found"
        };
    }

    var report = new BugReport()
    {
        UserApplication = application,
        UploadDate = DateTime.Now
    };

    if (!string.IsNullOrWhiteSpace(model.ShortDescription))
        report.ShortDesctiption = model.ShortDescription;

    if (!string.IsNullOrWhiteSpace(model.ContentLanguage))
        report.ContentLanguage = model.ContentLanguage;

    if (!string.IsNullOrWhiteSpace(model.Package))
        report.Package = model.Package;

    if (!string.IsNullOrWhiteSpace(model.BugDate))
    {
        DateTime convertedDate;

        if (!DateTime.TryParse(model.BugDate, out convertedDate))
        {
            return new ApiResponse()
            {
                ErrorMessage = "BugDate is not in correct format"
            };
        }

        report.BugDate = convertedDate;
    }

    if (!string.IsNullOrWhiteSpace(model.Version))
        report.Version = model.Version;

    if (!string.IsNullOrWhiteSpace(model.OSVersion))
        report.OSVersion = model.OSVersion;

    if (!string.IsNullOrWhiteSpace(model.Details))
        report.Details = model.Details;

    await mContext.Reports.AddAsync(report);
    await mContext.SaveChangesAsync();

    return new ApiResponse();
}
```

Ukázka kódu 5 Kód metody pro zpracování API dotazu na přidání reportu

7.3. Tvorba uživatelského rozhraní

Tvorba uživatelského rozhraní v rámci ASP.NET Core MVC aplikacích probíhá pomocí pohledů vytvořených v souborech „.cshtml“. Jedná se o běžné HTML[36] soubory, které je ale možné rozšířit pomocí speciálních znaků jazyka Razor. Ten umožňuje v HTML šablonách využívat prvky jazyka C#. Jednoduchým příkladem může být vygenerování tabulky, kde dojde ke spuštění běžného C# cyklu, který dynamicky generuje HTML značky pro jednotlivé řádky tabulky.

```
<table>
    <tr>
        <th>Id</th>
        <th>Description</th>
        <th>OS</th>
    </tr>
    @foreach (var bug in Model.Bugs)
    {
        <tr>
            <td>@bug.Id</td>
            <td>@bug.Description</td>
            <td>@bug.Os</td>
        </tr>
    }
</table>
```

Ukázka kódu 6 Jazyk Razor; Zdroj: autor

Speciálními pohledy jsou tzv. „layouty“. Ty slouží jako obecný předek pro další konkrétní pohledy. Layout může vypadat například tak, že obsahuje záhlaví, zápatí a obsah mezi nimi. Tento obsah je pak představován dynamicky doplňovanými pohledy, kdežto záhlaví a zápatí je vždy stejné. Mezi důležité prvky jazyka Razor, které jsou používány k tomuto dynamickému doplňování dat do layoutů patří volání metod „@RenderBody()“ a „@RenderSection()“. První metoda vykreslí na místo, kde je umístěna, konkrétní pohled. Druhá metoda vykreslí zvolenou sekci v závislosti na pohledu. Může se jednat například o postranní menu s doplňujícími informacemi k obsahu samotného pohledu. To jaký layout bude použit se dá definovat přímo ve zvoleném pohledu pomocí direktivy „@{ Layout = cesta_k_souboru }“.

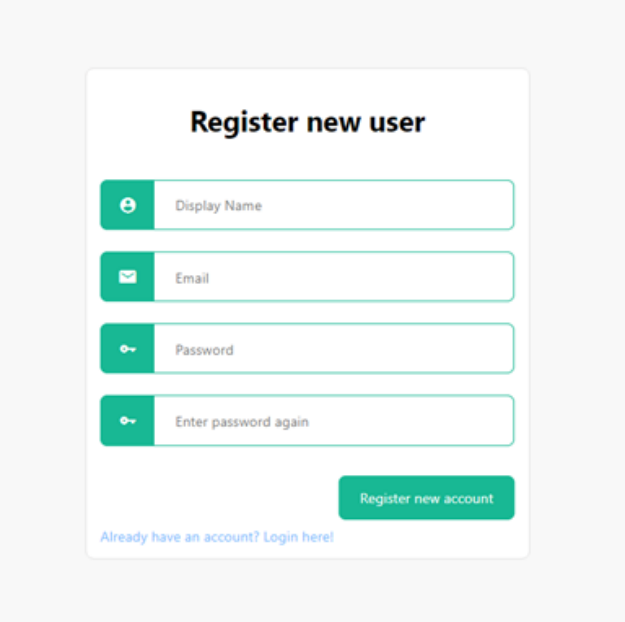
[36] Standardní formát určený pro tvorbu uživatelského rozhraní webové stránky

7.3.1 Pohledy v aplikaci

Z pohledu aplikace pro sledování chyb jsou definovány dva layouty. První layout je využit pro pohledy umožňující přihlášení a registraci uživatele. Jednotlivé pohledy tohoto layoutu pak obsahují HTML formulář skládající se např. z textových polí a potvrzovacího tlačítka. Součástí layoutu jsou také odkazy na JavaScriptové knihovny JQuery a JQuery.Validation.Unobtrusive. Ty jsou v pohledech využity pro ověření, zda mají dané formuláře náležitě a správně vyplněná všechna pole předtím, než je jejich obsah odeslán na server. Tento přístup umožňuje snížit počet dotazů odeslaných na server. Samozřejmě je však nutné na serveru ověřovat všechny příchozí formuláře, protože uživatel může manipulovat s lokálním obsahem stránky a skripty tak vyřadit či změnit. To, jaká jsou pravidla pro ověřování jednotlivých prvků formuláře je definováno v modelu pohledu.

```
@model BugStack.Server.RegisterViewModel;

@{
    Layout = "~/Views/Shared/_LoginLayout.cshtml";
    ViewBag.Title = Model.Title;
}

<div class="vcenter-helper login-body">
    <section class="login-form-component">
        <h1>Register new user</h1>
        <form asp-controller="Auth" asp-action="Register" method="post" role="form">
            <div class="input-control input-text-container">
                <i class="input-text-icon icon ion-md-contact"></i>
                <input asp-for="DisplayName" type="text" placeholder="Display Name" />
            </div>
            <span asp-validation-for="DisplayName"></span>
            <div class="input-control input-text-container">
                <i class="input-text-icon icon ion-md-mail"></i>
                <input asp-for="Email" type="email" placeholder="Email" />
            </div>
            <span asp-validation-for="Email"></span>
            <div class="input-control input-text-container">
                <i class="input-text-icon icon ion-md-key"></i>
                <input asp-for="Password" type="password" placeholder="Password" />
            </div>
            <span asp-validation-for="Password"></span>
            <div class="input-control input-text-container">
                <i class="input-text-icon icon ion-md-key"></i>
                <input asp-for="PasswordVerification" type="password" placeholder="Enter password again" />
            </div>
            <span asp-validation-for="PasswordVerification"></span>
            <div asp-validation-summary="All"></div>
            <div>
                <input type="submit" value="Register new account" class="input-control align-right" />
            </div>
        </form>
        <a asp-controller="Auth" asp-action="Login">Already have an account? Login here!</a>
    </section>
</div>
```

Ukázka kódu 7 Kód pohledu Register.cshtml; Zdroj: autor



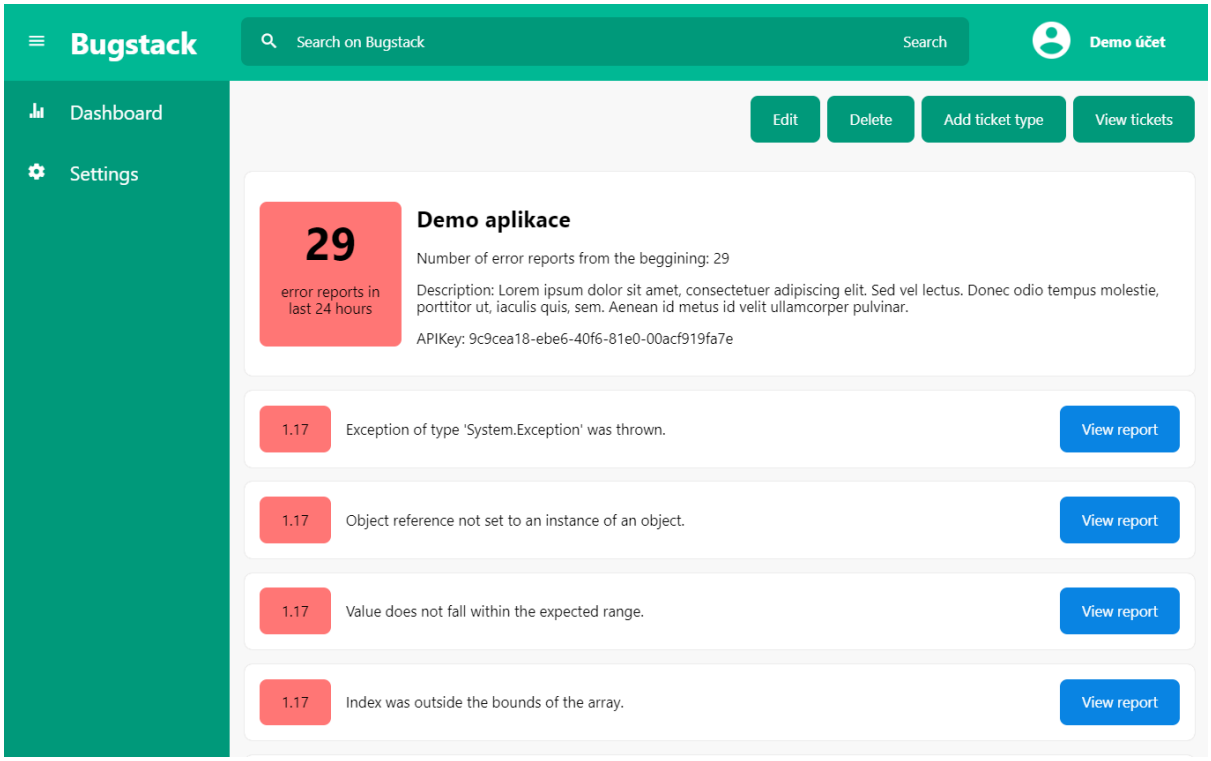
Obr. 9 Uživatelské rozhraní registračního formuláře; Zdroj: autor

V případě druhého layoutu je počet na něj navázaných pohledů značně větší. Prvním pohledem, který uživatel uvidí po přihlášení do systému je tzv. „Dashboard“. Na něm je znázorněn seznam všech aplikací, jejichž chyby sleduje spolu s informacemi, o jakou aplikaci jde, kolik chyb bylo zaznamenáno za posledních 24 hodin a kolik za dobu od počátku sledování. Pak také obsahuje odkazy na pohledy s podrobnými informacemi o jednotlivých sledovaných aplikacích a na pohled s formulářem pro přidání nové sledované aplikace.

Pohled s podrobnými informacemi sledované aplikace zobrazuje unikátní API klíč, který je sledované aplikaci přiřazen při jejím vytvoření a slouží k jednoznačné identifikaci toho, které aplikaci příchozí report náleží. Dále zobrazuje seznam příchozích reportů s krátkým popisem dané chyby, verzí aplikace, ve které nastala, a odkazem na podrobný popis reportu. Pohled má také odkazy na stránky pro upravení informací o aplikaci, odstranění sledované aplikace a na seznam manuálních reportů.

Pohled s informacemi o daném reportu obsahuje tabulku s jednotlivými položkami jako např. verze operačního systému, verze aplikace a detailní výpis chyby (též nazývaný jako „stack-trace“). Tento pohled má také tlačítko pro smazání daného reportu.

Poslední skupina pohledů používající druhý layout slouží ke správě manuálních reportů. Jde o pohledy pro přidání nových reportů, jejich typů, zobrazení detailního popisu a přidání komentářů.



Obr. 10 Uživatelské rozhraní pohledu „Dashboard“; Zdroj: autor

7.4. Databáze

Jako úložiště persistentních dat[37] je v projektu použita SQL databáze PostgreSQL. Pro usnadnění práce s touto databází je použit ORM framework EntityFramework Core. Ten umožňuje pracovat s databázovými tabulkami stejně jako s běžnými objekty v objektově orientovaném programování a přináší například integrovanou ochranu proti útokům typu „SQL injection“.[38] Bohužel jako téměř každá abstrakce sebou nese použití EntityFrameworku nevýhodu v podobě větší náročnosti na systémové prostředky. Tato vyšší náročnost však v drtivé většině případů nepředstavuje zásadní problém vzhledem k tomu, jak velké množství práce dokáže vývojáři ušetřit.

[37] Data, která jsou uchována i po ukončení aplikace
[38] Útok na databázi, kde přes neošetřený vstup projde závadný SQL kód, který může poškodit danou databázi či kompromitovat její data.

7.4.1 IdentityServer

Aplikace pro správu uživatelů (tj. registrace, přihlášení a odhlášení) využívá integrované knihovny IdentityServer. Díky ní není nutné tvořit vlastní mechanismy pro bezpečné ukládání hesel, správu oprávnění, proces autentifikace a dvoufázové ověřování. Knihovna například standardně šifruje hesla uživatelů. Součástí knihovny je i třída „IdentityUser“ představující předpis tabulky uživatelů, která bude následně vytvořena ORM frameworkem. Tuto třídu lze rozšířit o vlastní atributy. To se provede vytvořením nové třídy, která z třídy „IdentityUser“ dědí[39]. V rámci aplikace pro sledování chyb se tato nová třída nazývá „ApplicationUser“. Obsahuje navíc například atribut „DisplayName“ představující jméno uživatele, které bude v rámci aplikace zobrazováno. Třída „IdentityUser“ již obsahuje podobný atribut „UserName“. Ten však musí mít u každého uživatele unikátní hodnotu, kdežto „DisplayName“ ne.

7.4.2 Připojení databáze do projektu

Proto, aby mohl EntityFramework komunikovat s PostgreSQL databází je nutné nainstalovat NuGet balíček, který to umožňuje. Toho lze v případě Visual Studia docílit pomocí grafického průvodce. Stačí kliknout pravým tlačítkem myši na projekt, do kterého má být balíček nainstalován a vybrat možnost „Manage Nuget Packages“. Objeví se vizuální průvodce, kde je nutné ve vyhledávacím poli nalézt balíček s názvem Npgsql.EntityFrameworkCore.PostgreSQL a nainstalovat ho. V případě použití CLI nástrojů bude postup instalace probíhat následovně. Nejdříve je nutné v příkazové řádce přejít do adresáře projektu (v tomto případě „./BugStack.Server/“) a pak již stačí spustit příkaz k instalaci samotného balíčku.

```
dotnet add Npgsql.EntityFrameworkCore.PostgreSQL
```

Ukázka kódu 8 Příkaz pro přidání Npgsql NuGet balíčku do projektu; Zdroj: autor

Dalším krokem k připojení databáze do projektu je vytvořit třídu s názvem „ApplicationDbContext“. Ta představuje předpis tabulek, ze kterých je následně vytvořena fyzická struktura databáze. Tato třída musí dědit z třídy „IdentityDbContext“, protože je v projektu použita knihovna IdentityServer. Pokud by nebyla použita, tak by stačilo dědit z třídy „DbContext“. „IdentityDbContext“ má jeden generický parametr[40].

[39] Koncept objektově orientovaného programování, kdy potomek dědí vlastnosti a schopnosti předka
[40] Proměnná, jež může být definována s různými datovými typy

Datový typ tohoto parametru však musí být potomkem třídy „IdentityUser“. V aplikaci je definovaná výše zmíněná vlastní třída „ApplicationUser,“ která z „IdentityUser“ dědí, a proto bude nastavena jako datový typ generického parametru. Následně bude vyžadováno, aby byl v třídě „ApplicationDbContext“ definován konstruktor, který volá konstruktor svého rodiče. Jeho obsah však může zůstat prázdný.

Jednotlivé tabulky jsou pak definovány jako atributy typu „DbSet“. Ten má jeden generický parametr. Pro třídu definovanou v generickém parametru bude následně vygenerována tabulka v databázi a zároveň budou sloužit jako kolekce prvků daného typu v aplikaci. Veškeré změny provedené nad touto kolekcí budou následně propagovány do samotné databáze. V rámci aplikace pro sledování chyb jsou definovány kolekce pro sledované aplikace a jejich reporty chybách.

```
public class ApplicationDbContext : IdentityDbContext<ApplicationUser>
{
    public DbSet<UserApplication> UserApplications { get; set; }

    public DbSet<BugReport> Reports { get; set; }

    public DbSet<Ticket> Tickets { get; set; }

    public DbSet<TicketType> TicketTypes { get; set; }

    public DbSet<Comment> Comments { get; set; }

    public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
        : base(options) {}
}
```

Ukázka kódu 9 Třída „ApplicationDbContext“ definující databázový kontext; Zdroj: autor

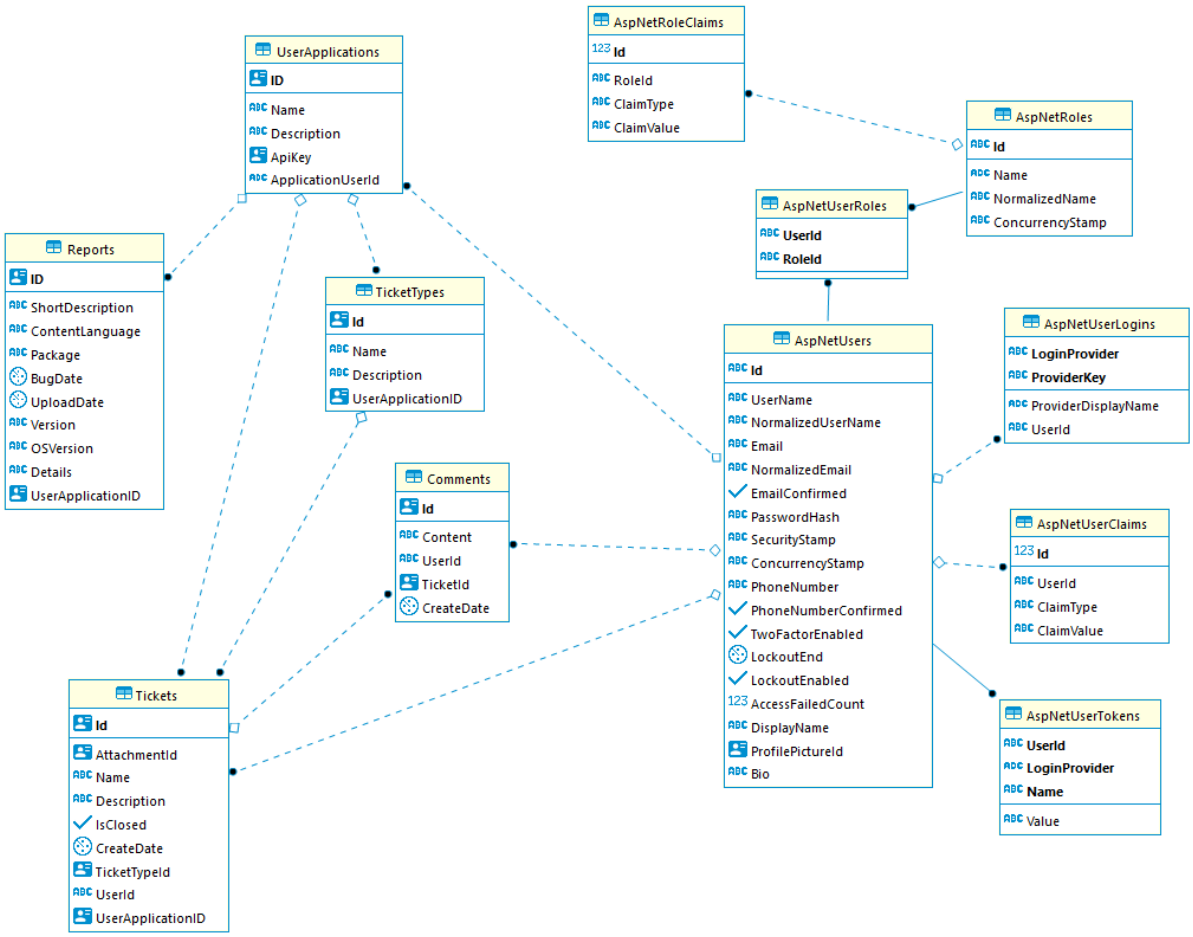
Poté už je nutné jen nastavit Dependency-Injection službu pro samotný databázový kontext. To se provede zavoláním rozšiřující metody „AddDbContext“ nad kolekcí „IServiceCollection“ v metodě „ConfigureServices“ v souboru „Startup.cs“. „AddDbContext“ má jako vstupní parametr nastavení „DbContextOptionsBuilder“ nad nímž lze volat rozšiřující metodu „UseNpgsql“. Ta pochází z doinstalovaného NuGet balíčku a její vstupní parametr představuje textový řetězec s udají k připojení do databáze. Tyto údaje byly přečteny z JSON souboru s nastavením aplikace.

```
public void ConfigureServices(IServiceCollection services)
{
    ...

    services.AddDbContext<ApplicationDbContext>(options =>
    {
```

```
        options.UseNpgsql(Configuration["ConnectionStrings:DatabaseConnection"]);
    });

    ...
}
```

Ukázka kódu 10 Přidání DI služby databázového kontextu do aplikace; Zdroj: autor

7.4.3 Fyzická struktura databáze



Obr. 11 Struktura databáze aplikace pro sledování chyb; Zdroj: autor

Struktura databáze aplikace se skládá ze skupiny tabulek souvisejících s interními částmi ASP.NET Core a skupiny tabulek vytvořených přímo pro aplikaci. Tabulky „AspNetUserClaims“, „AspNetRoleClaims“, „AspNetRoles“ a „AspNetUserRoles“ jsou interními tabulkami a obsahují informace o rolích a právech jednotlivých uživatelů v aplikaci. Interní tabulka „AspNetUserLogins“ obsahuje informace o přihlášení pomocí poskytovatelů třetích stran, jako jsou např. Google, Microsoft, Twitter. „AspNetUsers“ je interní tabulka, která byla ale v rámci aplikace rozšířena o několik vlastních sloupců a vychází z programové třídy „ApplicationUser“ popsané v kapitola 7.4.2.

Vlastní tabulka „UserApplications“ uchovává data o sledovaných aplikacích jako jsou jméno, popis a unikátní API klíč. Součástí tabulky je také cizí klíč odkazující na uživatele, který sledovanou aplikaci vlastní.

Druhá tabulka vytvořená v rámci aplikace pro sledování chyb se nazývá „Reports“. Jejím úkolem je ukládat data o jednotlivých automatizovaných reportech. Obsahuje řadu textových sloupců jako např. popis chyby, jazyk aplikace, balíček aplikace, verzi aplikace, verzi operačního systému a detailní popis (např. ve formě výpisu „Stack-Trace“). Obsahuje také dva sloupce, které jsou datového typu „Date“. Jsou to otisky doby, kdy byl report o chybě zaznamenán a kdy byl nahrán do aplikace pro sledování chyb. Tabulka také obsahuje jeden cizí klíč, který určuje, ke které sledované aplikaci daný report patří.

Tabulka „Tickets“ slouží pro ukládání dat o manuálních reportech jako je např. název, popis, stav a typ. Typ manuálního reportu je cizí klíč odkazující na tabulku „TicketTypes“. Ta obsahuje všechny definované typy manuálních reportů. Poslední vlastní tabulka se nazývá „Comments“ a představuje komentáře, které mohou být dalšími uživateli přidány k manuálním reportům.

Je důležité podotknout, že u všech vlastních tabulek jsou unikátní klíče datového typu GUID. Ten představuje jedinečný řetězec znaků, který není jednoduše odhadnutelný. Takovýto přístup je daleko lepší z pohledu bezpečnosti než situace, kdy jsou unikátní klíče představovány celým číslem a vždy při požadavku na nový klíč se inkrementují o hodnotu 1. Slabinu celých čísel lze demonstrovat na následujícím příkladu. Útočník si nejprve zkusí vytvořit novou sledovanou aplikaci a zjistí, že její API klíč je např. 378. Díky tomu může odhadnout, že teoreticky všechna čísla menší než 378 představují API klíče dalších sledovaných aplikací. Pak mu již stačí vytvořit aplikaci, která bude odesílat falešné dotazy buď na konkrétní aplikaci nebo náhodně na všechny. Použití GUID pak v praxi znemožní, aby útočník mohl tímto způsobem kompromitovat aplikaci.

7.5. Tvorba klientské knihovny

Poslední částí aplikace pro sledování chyb je klientská knihovna. Její hlavní úkol je zajistit sběr a odeslání reportu o chybě ze sledované aplikace do aplikace pro sledování chyb. Ta nabízí standartní HTTP rozhraní, které je platformě nezávislé a díky tomu ho lze

konzumovat v jakémkoliv programovacím jazyce, který podporuje síťovou komunikaci. Ideální je takový jazyk, který dokáže pracovat s protokoly aplikační vrstvy[41].

7.5.1 Implementace na platformě .NET

Ukázková implementace klientské knihovny je vytvořena na platformě .NET a exportovaná ve formě NuGet balíčků, které může kdokoliv přidat do své aplikace a následně tak využívat služeb aplikace pro sledování chyb.

Prvním krokem, který je nutný před tvorbou klientské knihovny udělat je nainstalovat NuGet balíček s názvem „Newston.Json“. Ten slouží pro práci s JSON objekty. Je také nutné ověřit, že je přidána reference na projekt „BugStack.Core“.

```
public class BugStackClient
{
    private readonly HttpClient mHttpClient;

    private readonly string mAppVersion;

    private readonly string mApiKey;

    private readonly string mAppLanguage;

    private readonly string mBaseUrl;

    public BugStackClient(string appVersion, string apiKey, string appLanguage, string baseUrl)
    {
        mAppVersion = appVersion;
        mApiKey = apiKey;
        mAppLanguage = appLanguage;
        mBaseUrl = baseUrl;

        mHttpClient = new HttpClient();

        AppDomain.CurrentDomain.UnhandledException += new UnhandledExceptionEventHandler(HandleException);
    }

    Private void HandleException(object sender, UnhandledExceptionEventArgs e)
    {
        Exception ex = (e.ExceptionObject as Exception);

        if(ex == null)
            return;

        AddBugReportApiModel bugReport = new AddBugReportApiModel()
        {
            ApplicationApiKey = mApiKey,
            BugDate = DateTime.UtcNow.ToString(),
            Version = mAppVersion,
            ContentLanguage = mAppLanguage,
            Details = ex.StackTrace,
            ShortDescription = ex.Message,
            OSVersion = RuntimeInformation.OSDescription + " " + RuntimeInformation.OSArchitecture,
            Package = ex.Source,

        };

        string payload = JsonConvert.SerializeObject(bugReport);
        var content = new StringContent(payload, Encoding.UTF8, "application/json");
```

[41] Vrstva síťového modelu ISO/OSI, která představuje abstraktní protokoly nad nižšími vrstvami síťového modelu. Jsou určeny k implementaci přímo v klientských aplikací.

```
            _ = mHttpClient.PostAsync(mBaseUrl + ApiRoutes.AddReport, content).Result;
        }
    }
```

Ukázka kódu 11 Implementace klientské knihovny v C#; Zdroj: autor

Klientská knihovna je představována třídou nazvanou „BugStackClient“. Ta obsahuje pět privátních atributů, které jsou nastavené jako readonly[42]. „mHttpClient“ představuje instanci třídy „HttpClient“, která umožňuje posílat HTTP požadavky a přijímat na ně odpovědi. „mAppVersion“ reprezentuje verzi sledované aplikace. V proměnné „mApiKey“ je uložen API klíč sledované aplikace. „mAppLanguage“ uchovává jazyk ve kterém byla sledovaná aplikace spuštěna. Poslední proměnná „mBaseUrl“ slouží pro uložení základní adresy aplikace pro sledování chyb. Může být např. ve formátu „https://adresaaplikace.cz/“ nebo „http://localhost:5000/“.

Konstruktor třídy má jako parametry hodnoty pro jednotlivé atributy, do kterých následně dané hodnoty dosadí. Jedinou výjimku tvoří „HttpClient“, jehož instance je vytvořena právě v těle konstruktoru. „AppDomain.CurrentDomain.UnhandledException“ představuje událost[43], která je vyvolána v případě, že nastane výjimka, která není nikde jinde v aplikaci ošetřena. K této události je přihlášena nová instance třídy „UnhandledExceptionEventHandler“, která má jako svůj vstupní parametr metodu. Ta má podle struktury delegátu[44] dva vstupní parametry. První je typu „object“ a má název „sender“. Jedná se o referenci na objekt, který neošetřenou výjimku vyvolal. Druhým parametrem je instance třídy „UnhandledExceptionEventArgs“ s názvem „e“. Ta obsahuje informace o nastalé výjimce. Právě tato metoda definovaná ve vstupním parametru nové instance třídy „UnhandledExceptionEventArgs“ je následně zavolána v případě, že nastane událost neošetřené výjimky.

Vytvořená metoda „HandleException“ splňuje požadavky definovaného delegátu, a proto je využita jako vstupní parametr nové instance třídy „UnhandledExceptionEventHandler“. Úkolem této metody je starat se o záznam

[42] Hodnotu do takto označené proměnné lze dosadit pouze v konstruktoru třídy a dále její obsah již nelze měnit
[43] Na platformě .NET představuje implementaci návrhového vzoru pozorovatel. Událost má seznam objektů, které ji pozorují a pokud tato událost nastane, tak jsou všichni pozorovatelé informováni o tom, že tato událost nastala
[44] Delegát představuje předpis metody. Určuje tedy návratový typ metody a také jaké vstupní parametry obsahuje

a následné odeslání reportu do aplikace pro sledování chyb. Z důvodu, že na .NET platformě lze vyvolat výjimku různých datlových typů, tak vstupní parametr typu „UnhandledExceptionEventArgs“ obsahuje objekt obecného datového typu „ExceptionObject“. Tento objekt je nejprve nutné tzv. přetypovat[45] na typ „Exception“. Z něj lze již jednoduše získat informace o dané výjimce. .NET nabízí možnost bezpečného přetypování pomocí klíčového slova „as“. Takto provedený pokus o přetypování skončí úspěchem pouze tehdy, když objekt, který má být přetypován může být daného datového typu. Pokud by nebyl, tak je výsledkem procesu „null“. Tedy stav, kdy daná proměnná neodkazuje na žádnou instanci objektu. Tento postup prakticky představuje prvních několik řádku metody „HandleException“, kdy nejdříve dojde k pokusu o přetypování a následně se vyzkouší, zda se přetypování povedlo. Pokud by se nepovedlo, tak zde metoda skončí a dále již daný „ExceptionObject“ nebude zpracovávat.

Pokud se vše povede, tak je následně vytvořena nová instance třídy „AddBugReportApiModel“, která pochází z projektu „BugStack.Core“. Do této instance jsou dosazeny informace o chybě. Z přetypovaného objektu chyby jsou využity atributy „StackTrace“, „Message“ a „Source“, které představují dlouhý výpis chyby, krátký popis chyby a název komponenty aplikace, která chybu způsobila. Za zmínku stojí i atributy „OSDescription“ a „OSArchitectureů statické třídy „RuntimeInformation“, které obsahují informace o operačním systému a jeho architektuře.

Dále je nutné vytvořenou instanci třídy „AddBugReportApiModel“ tzv. serializovat[46] do textové podobny JSON objektu, aby mohl být poslán pomocí HTTP požadavku do aplikace pro sledování chyb. Dalším krokem je vytvoření nové instance třídy „StringContent“, která má jako vstupní parametry textový obsah (zde bude dosazen serializovaný JSON kód), typ kódování (bude zvolen UTF8) a typ obsahu (Textový řetězec s informací o tom, že se jedná o JSON kód). Nakonec už lze jednoduše poslat požadavek pomocí metody „PostAsync“ na instanci třídy „HttpClient“ na URL adresu aplikace pro sledování chyb spolu s obsahem v objektu „StringContent“.

Teď již stačí vytvořit NuGet balíček klientské aplikace. Před jeho vytvořením je však nutné nastavit několik parametrů v konfiguračním souboru projektu klientské knihovny s názvem „BugStack.Client.csproj“. Jedná se o jednoduchý XML soubor, kde

[45] Změna datového typu daného objektu na jiný
[46] Rozložení .NET objektu do takové podoby, aby mohl být např. poslán za pomoci síťové komunikace

jednotlivé položky představují zvolená nastavení projektu. Nejprve je nutné vyhledat počáteční a konečný tag „PropertyGroup“. Mezi nimi se nachází tag „TargetFramework“ specifikující implementaci .NET platformy, která bude použita pro sestavení projektu. Pro nastavení NuGet balíčku je nutné přidat ještě několik dalších tagů. Prvním je „PackageId“, jehož hodnota představuje unikátní název balíčku, který bude vidět v dané NuGet galerii. Druhý tag, který je nutné přidat nese název „Version“ a udává verzi balíčku. Tag „Authors“ udává název autora balíčku. Poslední přidaný tag nese název „Company“ a jeho hodnota je název skupiny projektů, do které daný balíček patří.

```
<PropertyGroup>
  <TargetFramework>netstandard2.0</TargetFramework>
  <PackageId>BugStack.Client</PackageId>
  <Version>1.0.0</Version>
  <Authors>Tomas Malik</Authors>
  <Company>BugStack</Company>
</PropertyGroup>
```

Ukázka kódu 12 Nastavení hodnot NuGet balíčku klientské knihovny; Zdroj: autor

Nyní je již pouze potřeba balíček zabalit. To lze v případě použití .NET CLI udělat otevřením adresáře projektu klientské knihovny „BugStack.Client“ v příkazové řádce a následným zavoláním příkazu „dotnet pack“. Ten vytvoří ve složce „bin“ soubor s balíčkem, který má příponu „.nupkg“. Při použití Visual Studia stačí v prohlížeči souborů projektu otevřít kontextové menu stisknutím pravého tlačítka myši na konfiguračním souboru projektu. Následně je nutné z menu zvolit možnost „Pack“. Výsledný balíček se stejně jako v případě použití .NET CLI nachází ve složce „bin“.

Stejný způsobem je nutné vytvořit i balíček pro projekt „BugStack.Core“, na kterém je projekt „BugStack.Client“ závislý. To znamená upravit konfigurační soubor projektu, přidat požadované hodnoty a následně balíček vytvořit buď pomocí .NET CLI nebo Visual Studia.

Vytvořený balíček je možné umístit například do oficiální galerie nuget.org. Pro potřeby vývoje lze však použít přidání balíčku z lokálního adresáře. Pomocí .NET CLI je postup velice jednoduchý. Stačí příkaz pro přidání balíčku spustit s argumentem, který má cestu do adresáře, kde jsou lokální balíčky uloženy.

```
dotnet add package BugStack.Client --source C://LocalNuGet/
```

Ukázka kódu 13 Přidání lokálního NuGet balíčku pomocí .NET CLI; Zdroj: autor;

Vývojáři tedy stačí nainstalovat balíček klientské knihovny do své aplikace a před prováděním jakéhokoliv vlastního kódu vytvořit novou instanci třídy „BugStackClient“, která jako vstupní parametry vyžaduje API klíč a další nastavení. Následně, kdykoliv dojde v aplikaci k výjimce, tak je report o ní odeslán do aplikace pro sledování chyb.

8. Testování aplikace pro sledování chyb

Poslední částí vývoje aplikace pro sledování chyb je otestování její funkčnosti. Kromě toho by testování mělo poukázat i na možné další případy užití, které by mohly být do aplikace implementovány v budoucnu.

Proces testování začal vytvořením testovacího účtu, který simuloval budoucího uživatele aplikace pro sledování chyb. V rámci tohoto účtu byla vytvořena sledovaná aplikace, které byl při jejím vytvoření korektně přidělen API klíč.

Další krokem bylo vytvořit sledovanou aplikaci fyzicky. Jako šablona projektu byla zvolena WPF aplikace s grafickým rozhraním. Do aplikace byl přidán NuGet balíček „BugStack.Client“. Soubor „MainWindow.xaml.cs“ představuje třídu hlavního okna aplikace s konstruktorem, ve kterém dochází k inicializaci komponent okna. V rámci tohoto konstruktoru došlo k vytvoření nové instance třídy „BugStack.Client“.

Do uživatelského prostředí aplikace bylo přidáno tlačítko, které po stisknutí vyvolá neošetřenou výjimku v aplikaci. To probíhalo následovně. Nejdříve byla definovaná proměnná typu obecné kolekce a následně došlo k pokusu přidat prvek do této neinicializované proměnné, což vyústilo ve výjimkou typu „NullReferenceException“. V tuto chvíli mělo dojít k vytvoření a odeslání reportu do aplikace pro sledování chyb.

Poté došlo ke kontrole toho, zda report dorazil do aplikace pro sledování chyb. Ta proběhla úspěšně. Následovalo úspěšné odeslání ještě dalších 20 reportů různých typů výjimek, které úspěšně ověřily, že aplikace skutečně funguje, tak jak bylo zamýšleno.

Na závěr bylo otestováno webové prostředí pro správu manuálních reportů. To znamenalo vyzkoušet přidání několika typů manuálních reportů, jako např. „Bug“ nebo „Feature request“. Poté došlo k přidání několika reportů a komentářů k nim z dalších uživatelských účtů, které byly vytvořeny speciálně za účelem testování. Zároveň bylo úspěšně otestováno, že pokud dojde k uzamknutí reportu vlastníkem sledované aplikace, tak už další přihlášení uživatelé nemají právo přidávat další komentáře.

Testování kromě úspěšného fungování přineslo návrh na další rozšiřující funkcionalitu aplikace v podobě přidání různých filtrů přijatých reportů. Dále pak také jejich vizualizaci pomocí grafů a různých statistik např. podle verze a typu operačního systému, ze kterého přišly.

9. Závěr

Cílem práce bylo popsat problematiku tvorby webové aplikace pro sledování chyb a vytvořit ukázkovou aplikaci s využitím platformy .NET. Úvodní kapitoly bakalářské práce se věnovaly teoretické stránce a došlo k popsaní již existujících řešení pro sledování chyb a návrhu ukázkové aplikace. Další kapitoly se věnovaly popsání platformy .NET a frameworku ASP.NET, který byl použit pro následné vytvoření ukázkové aplikace. Poslední část práce byla zaměřená na samotný popis implementace ukázkové aplikace. Byly zde popsány jednotlivé komponenty aplikace od uživatelského prostředí, přes databázovou strukturu a funkční stránku až po tvorbu klientské knihovny. Cíl práce byl tedy splněn dle zadání.

Po řádném otestování by mělo být možné aplikaci nasadit do produkčního prostředí a používat jako podporu při vývoji dalších aplikací. Může být také použita jako jako předloha pro implementaci podobných aplikací takovéhoto druhu.

Jak bylo nastíněno v kapitole o testování, tak v budoucnu by bylo vhodné do aplikace přidat další možnosti vizualizace informací z reportů, např. pomocí grafů či jiných vhodně zvolených vizualizací.

10. Seznam použité literatury

[1] GITHUB INC. Mastering Issues. *GitHub Guides* [online]. 7. duben 2014. Dostupné z: https://guides.github.com/features/issues/

[2] AGGARWAL, Kshitij. Effective logging in Production with Firebase Crashlytics. *Envision blog* [online]. 30. červenec 2019. Dostupné z: https://www.letsenvision.com/blog/effective-logging-in-production-with-firebase-crashlytics

[3] MICROSOFT. App Center - Overview. *App Center - Overview* [online]. Dostupné z: https://docs.microsoft.com/en-us/appcenter/dashboard/index

[4] FISHMAN, Alex. Is Bugsee Any Good? *Bugsee Blog* [online]. 20. březen 2018. Dostupné z: https://www.bugsee.com/blog/is-bugsee-any-good/

[5] WIEGERS, Karl Eugene. *Software Requirements*. 3. vyd. Redmond, Washington: Microsoft Press, 2013. ISBN 978-0-7356-7966-5.

[6] TEKRIWAL, Sachin. .Net Framework all versions history and key features. *Tutorials Plus* [online]. 14. červenec 2017. Dostupné z: http://www.tutorialplus.net/Articles/dot-net-framework-version-history.htm

[7] GAMEFROMSCRATCH. C# GAME ENGINES. *GameFromScratch* [online]. 4. září 2018. Dostupné z: https://www.gamefromscratch.com/post/2018/09/04/CSharp_Game_Engines.aspx

[8] MICROSOFT. What is .NET Framework? *What is .NET Framework?* [online]. Dostupné z: https://dotnet.microsoft.com/learn/dotnet/what-is-dotnet-framework

[9] MICROSOFT. .NET Programming Languages. *.NET Programming Languages* [online]. Dostupné z: https://dotnet.microsoft.com/languages

[10] LINNIK, Irina. A Brief History of .NET Framework. *Softteco Blog* [online]. 13. červen 2019. Dostupné z: https://softteco.com/blog/a-brief-history-of-net-framework

[11] DE ICAZA, Miguel. Mono 0.3. *Mono Project Blog* [online]. 12. červenec 2001. Dostupné z: https://www.mono-project.com/docs/about-mono/releases/0.3.0/

[12] MONO PROJECT. Supported Platforms. *Mono Documentation* [online]. 6. květen 2019. Dostupné z: https://www.mono-project.com/docs/about-mono/supported-platforms/

[13] UNITY TECHNOLOGIES. Coding in C# in Unity for beginners. *Unity3D* [online]. 2019. Dostupné z: https://unity3d.com/learning-c-sharp-in-unity-for-beginners

[14] ETCHEVERRY, Ignacio Roldán. Introducing C# in Godot. *Godot Blog* [online]. 21. říjen 2017. Dostupné z: https://godotengine.org/article/introducing-csharp-godot

[15] CRYTEK GMBH. Introducing CRYENGINE V: A letter to our Community. *CryEngine Blog* [online]. 17. březen 2016. Dostupné z: https://www.cryengine.com/news/view/introducing-cryengine-v-a-letter-to-our-community

[16] GUTHRIE, Scott. Microsoft to acquire Xamarin and empower more developers to build apps on any device. *Official Microsoft Blog* [online]. 24. únor 2016. Dostupné z: https://blogs.microsoft.com/blog/2016/02/24/microsoft-to-acquire-xamarin-and-empower-more-developers-to-build-apps-on-any-device/

[17] TORGERSEN, Mads. Building C# 8.0. *.NET Blog* [online]. 12. listopad 2018. Dostupné z: https://devblogs.microsoft.com/dotnet/building-c-8-0/

[18] MONO PROJECT. C# Compiler. *Mono Documentation* [online]. 10. duben 2018. Dostupné z: https://www.mono-project.com/docs/about-mono/languages/csharp/

[19] LANDER, Richard. Introducing .NET 5. *.NET Blog* [online]. 6. květen 2019. Dostupné z: https://devblogs.microsoft.com/dotnet/introducing-net-5/

[20] LANDWERTH, Immo. Announcing .NET Standard 2.1. *.NET Blog* [online]. 5. listopad 2018. Dostupné z: https://devblogs.microsoft.com/dotnet/announcing-net-standard-2-1/

[21] VISUAL STUDIO TEAM. WPF in Visual Studio 2010 – Part 1 : Introduction. *Visual Studio Blog* [online]. 16. únor 2010. Dostupné z: https://devblogs.microsoft.com/visualstudio/wpf-in-visual-studio-2010-part-1-introduction/

[22] DE ICAZA, Miguel. Announcing the new Visual Studio for Mac. *Visual Studio Blog* [online]. 16. listopad 2016. Dostupné z: https://devblogs.microsoft.com/visualstudio/visual-studio-for-mac/

[23] MICROSOFT. *IntelliCode for Visual Studio Code overview* [online]. 24. duben 2019. Dostupné z: https://docs.microsoft.com/cs-cz/visualstudio/intellicode/intellicode-visual-studio-code

[24] SILVER, Amanda. Introducing Visual Studio Live Share. *Visual Studio Code* [online]. 15. říjen 2017. Dostupné z: https://code.visualstudio.com/blogs/2017/11/15/live-share

[25] JETBRAINS S.R.O. Fast & powerful cross-platform .NET IDE. *JetBrains Rider - Features* [online]. 2019. Dostupné z: https://www.jetbrains.com/rider/features/

[26] SHARMA, Loïc. BaGet - A lightweight NuGet and symbol server. *Github.com* [online]. 10. únor 2019. Dostupné z: https://github.com/loic-sharma/BaGet

[27] EMGARTEN, Justin. What is Sleet? *github.com* [online]. 14. říjen 2019. Dostupné z: https://github.com/emgarten/sleet

[28] TECHEMPOWER, INC. Web Framework Benchmarks - Round 18 (Plaintext). *techempower.com/benchmarks* [online]. 9. červenec 2019. Dostupné

z: https://www.techempower.com/benchmarks/#section=data-r18&hw=cl&test=plaintext

[29] ŠUTA, Ibrahim. ASP.NET Core Razor Pages – Introduction. *CodingBlast* [online]. 23. červenec 2017. Dostupné z: https://codingblast.com/asp-net-core-razor-pages/

Univerzita Hradec Králové
Fakulta informatiky a managementu

Zadání bakalářské práce

Autor:	**Tomáš Malík**
Studium:	I1700106
Studijní program:	B1802 Aplikovaná informatika
Studijní obor:	Aplikovaná informatika
Název bakalářské práce:	**Webová aplikace pro sledování chyb**
Název bakalářské práce AJ:	Web Application for Crash Reporting

Cíl, metody, literatura, předpoklady:

Cíl: Popsat problematiku tvorby webové aplikace pro sledování chyb a vytvořit ukázkovou aplikaci s využitím platformy .NET.

Osnova:

1. Úvod
2. Nástroje pro sledování chyb
3. Analýza a návrh aplikace
4. Platforma .NET
5. Popis implementace aplikace
6. Závěr

Garantující pracoviště:	Katedra informatiky a kvantitativních metod, Fakulta informatiky a managementu
Vedoucí práce:	doc. Mgr. Tomáš Kozel, Ph.D.
Datum zadání závěrečné práce:	14.1.2018