Univerzita Hradec Králové
Fakulta informatiky a managementu
KIT – Katedra informačních technologií

Moderní metody programování s využitím automatického testování a deploymentu

Bakalářská práce

Autor: Martin Rampouch
Studijní obor: Aplikovaná informatika

Vedoucí práce: Ing. Jaroslav Langer

Hradec Králové Srpen 2023

Prohlášení:

Prohlašuji, že jsem bakalářskou práci zpracoval samostatně a s použitím uvedené literatury.

V Hradci Králové dne 9.8.2023

Martin Rampouch

Poděkování:

Děkuji vedoucímu bakalářské práce Ing. Jaroslavovi Langerovi za metodické vedení práce, odborné rady, věcné připomínky, vstřícný přístup a vše co mi pomohlo při zpracování bakalářské práce

Anotace

Práce se zabývá moderními způsoby, principy a technologiemi, které aktuálně nachází využití při vývoji softwaru. Především je tato práce zaměřena na ukázání procesu vývoje, testování a nasazení aplikace. Důraz je kladen převážně na vývoj udržitelné a lehce rozšiřitelné aplikace.

V první části scénáře je realizována v podobně návrhu a implementace architektury back-end aplikace v technologii ASP NET Core a popis zvolené udržitelné architektury aplikace.

Společně s touto částí je navržena Microsoft SQL databáze pro uchování a jednoduchou manipulaci s daty. V této části je vysvětlen způsob manipulace s daty v rámci back-end aplikace skrze technologii Entity Framework Core.

Další částí je návrh a přiblížení vývoje a architektury webového uživatelského prostředí v JavaScriptovém frameworku ReactJs. Součástí je ukázka tvorby jednotlivých komponent a způsob komunikace webové aplikace s back-end aplikací.

Následně je na front-end aplikaci vytvořen a popsán jednoduchý testový scénář pro ověření funkčnosti uživatelského prostředí aplikace. Zvolenou technologií pro testování je Cypress knihovna.

Poslední částí této práce je popis způsobu nasazení aplikace do cloudového prostředí. K tomu je využita pipeline, která zajišťuje automatizované nasazení aplikace.

Klíčová slova: .net core, react.js, automatizované testování, CI/CD pipeline, principy a metody vývoje

Annotation

Title: Modern methods of programming using automatic testing and deployment

The thesis deals with modern methods, principles and technologies that are currently used in software development. In particular, this thesis aims to show the process of application development, testing and deployment. The focus is mainly on the development of a sustainable and easily extensible application.

In the first part of the scenario, the design and implementation of the back-end application architecture in ASP NET Core technology and a description of the chosen sustainable application architecture is similarly implemented.

Together with this part, a Microsoft SQL database is designed for data storage and easy manipulation. In this section, the method of data manipulation within the back-end application through Entity Framework Core technology is explained.

The next part is the design and introduction of the development and architecture of the web user interface in the JavaScript framework ReactJs. This includes a demonstration of the creation of component and how the web application communicates with the back-end application.

Subsequently, a simple test scenario is created and described on the front-end application to verify the functionality of the application UI. The chosen technology for testing is the Cypress library.

The last part of this paper describes how to deploy the application in a cloud environment. For this purpose, a pipeline is used to provide an automated deployment of the application.

Keywords: .net core, react.js, automated testing, CI/CD pipeline, development principles and methods

Obsah

Seznam obrázků

Seznam ukázek kódu

1 Úvod

V současné době dochází k vývoji a přesunu klasických desktopových aplikací do webového prostředí. S tímto trendem jsou spojeny stále rostoucí požadavky na jednoduchost a plynulost ovládání, rozšiřování a optimalizování těchto aplikací.
K dosažení těchto cílů je zapotřebí znát principy a technologie umožňující udržitelný vývoj aplikací. Pokud by docházelo k častému porušení těchto principů, stal by se vývoj aplikací neudržitelný a velice nákladný.
Kvalitu softwaru je potřeba zaručit a řádně testovat předtím, než je vývoj software u konce. K tomuto ověření slouží nespočet automatizovaných testovacích softwarů, které lze snadno implementovat přímo do aplikací nebo do procesu ověření a nasazení aplikací.
V teoretické části této práce autor popíše jednotlivé technologie, které budou využívány v praktické části. Součástí teoretické části je rozebrání základních metod a principů vývoje aplikací pro dosažení snadno rozšiřitelné a udržitelné aplikace.
Cílem práce je ukázat udržitelný způsob vývoje webové aplikace včetně testování automatizovaným softwarem a způsob nasazení jednotlivých částí aplikace. K demonstraci tohoto cíle byly využity technologie .NET Core, ReactJs, Cypress, CI/CD Pipeline.

2 Teoretická část

V této části bakalářské práce se autor zabývá teoretickou částí zvoleného tématu. Dochází zde k vysvětlení pojmů a popisu technologií a principů pojící se s tématem.

2.1 Technologie Microsoft .Net

Jedná se o open-source vývojářskou platformu, která umožňuje vývoj různých aplikací, například web, desktop, cloud, mobile, microservice, machine learning, IoT a dalších.

Umožňuje vývoj cross platform aplikací, tudíž aplikace může být spuštěna nezávisle na operačním systému, který implementuje technologii .Net. Zároveň je zde možnost vyvíjet aplikace na konkrétní operační systém.

Tato technologie byla vyvinuta firmou Microsoft a je aktuálně spravovaná nezávislou organizací .Net Foundation, která podporuje otevřený vývoj a spolupráci v rámci .Net.

Knihovna .Net především podporuje známé programovací jazyky, jakou jsou C#, Visual Basic a F #. Verze .Net lze rozdělit na .Net Core, .Net Framework, Xamarin/Mono .Net for mobile. Zároveň existuje jednotná verze .Net Standard. Tato edice poskytuje sdílenou sadu knihoven ze všech tří zmíněných platforem. Knihovny .Net se dají rozšířit o další externí knihovny pomocí balíčkového ekosystému zvanému jako NuGet. [1]

2.2 Programovací jazyk C#

Programovací jazyk C# byl poprvé distribuován během července roku 2000. Vyšel z dílen Microsoftu jako součást knihovny .Net Framework.

C# je vysokoúrovňový programovací jazyk podporující objektově orientované programování. Byl zamýšlen jako univerzální, moderní a jednoduchý programovací jazyk.

C# se pro jeho přísnou kontrolu datových typů řadí mezi typové programovací jazyky, ale i přesto obsahuje některé netypové datové typy. Zároveň poskytuje spoustu dalších funkcí, jakou jsou detekce pokusů o využití neinicializovaných proměnných a garbage collector.

Tento programovací jazyk je sice podobný programovacímu jazyku C, ovšem ve výkonu a potřebě operační paměti mu není přímím konkurentem. Ani tak nebyl zamýšlen. C# umožňuje tvorbu široké škály aplikací. Od hostovaných aplikací přes velké i malé vestavěné systémy až po miniaturní aplikace či specializované funkce. [2]

2.3 Objektově orientované programování (OOP)

Objektově orientované programování, dále jen OOP, není žádným programovacím jazykem. Jedná se o principy a myšlenkové procesy strategicky využité k programování softwaru. Je tedy založen na konceptu vidění světa jako jednotlivých objektů, které mezi sebou komunikují a vykonávají určitou činnost.

Tyto principy díky modulárním návrhům zajišťují flexibilitu a přehlednost softwaru. Flexibilní architektura je jednodušší a snazší pro úpravy kódu. Je to zejména tím, že objekty, které tvoří architekturu softwaru, mají jasné hranice, což zjednodušuje orientaci mezi jednotlivými objekty a jejich vzájemnou zaměnitelnost.

V OOP je tedy každé chování softwaru obsaženo v samostatném objektu neboli třídě. Díky tomu jsme schopni jednodušeji nahlížet na spolupráci mezi objekty. Každá tato třída obsahuje své vlastní jméno a měla by mít vlastní specifické chování. Chování třídy je vyjádřeno pomocí procedur neboli metod, které konají určitou činnost, a proměnných, které ukládají specifické hodnoty. Na každý objekt lze tedy nahlížet jako na konkrétní datový typ.

Na principech OOP jsou založeny různé programovací jazyky. Příkladem může být programovací jazyk C#. Ten dodržuje základní principy objektově orientovaného myšlení. Těmito základními principy jsou abstrakce, zapouzdření, dědičnost a polymorfismus.

Abstrakce je princip, který umožňuje objektu skrývat vnitřní implementaci. Objekty komunikující s jiným objektem by se neměly starat, jakým způsobem žádaný objekt vykoná určitou úlohu. Důraz je tedy kladen na to, co objekt nabízí, a ne na to, jak to provádí. Objekty se tedy zajímají pouze o veřejné rozhraní a návratovou hodnotu.

Každý objekt má svou vlastní privátní logiku a proměnné pro jeho funkčnost. Proto je důležité využívat princip zapouzdření. Tento princip skryje určité části objektu před okolním světem. Okolí objektu se k těmto datům může dostat pomocí

zasíláním zpráv z veřejného rozhraní. Jde o důležitou funkci z hlediska zamezení nežádoucích změn zvenčí a udržení konzistence.

Dědičnost umožňuje vývojářům zapouzdřit určité chování a vlastnosti do objektu, který bude následně použit jako rodič. Třídy dědící (child) toto chování nadále mohou doplnit, více se specifikovat a odvíjet se na zděděném, společném základu. Dědičnost umožňuje tvorbu kódu pomocí hierarchických vztahů mezi objekty a také znovupoužitelnost kódu. Tento princip není nějak omezen, to znamená, že objekt potomek může dědit z rodiče, který zároveň dědí z jeho vlastního rodiče.

Polymorfismus je způsob zaměnitelnosti objektu za jiný objekt. Tyto objekty musí dědit ze stejného předka, to znamená, že mají sdílené rozhraní. [3], [4], [5], [6]

2.4 Entity Framework core

Entity Framework Core je knihovna odvozena od starší knihovny Entity Framework vytvořena společností Microsoft. Slouží pro napojení aplikace k databázi, a to pomocí objektově-relačního mapovače (O/RM). Ten poskytuje možnost práce s daty jako s .Net objekty, tím že je mapuje na třídy aplikace.

Entity Framework Core pro přístup k datům využívá tzv. model. Ten se skládá z entitních a kontextových tříd. Představuje relaci s databází a umožní dotazování se na data a jejich ukládání do databáze. K dotazování a manipulaci dat, je využit LINQ (Language Integrated Query). [7]

```
namespace RampouchBcApi.Database;
public class Class1
{
    public class DatabaseContext : DbContext, IDatabaseContext
    {
        private readonly IHttpContextAccessor _httpContextAccessor;

        public          DatabaseContext(DbContextOptions<DatabaseContext>          options,
IHttpContextAccessor httpContextAccessor) : base(options)
        {
            _httpContextAccessor = httpContextAccessor;
        }

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            base.OnModelCreating(modelBuilder);

            var entityForeignKeys = modelBuilder.Model
                .GetEntityTypes()
                .SelectMany(e => e.GetForeignKeys())
                .Where(fk    =>    fk.PrincipalEntityType.ClrType    ==    typeof(User)    &&
fk.Properties.Any(p => p.Name == "CreatedById"))
                .ToList();
        }

        public DbSet<Event> Events { get; set; }
        public DbSet<EventDetail> EventDetails { get; set; }
        public DbSet<AppFile> AppFiles { get; set; }

        public class AppFile : Entity
        {
            public string Name { get; set; }

            public byte[] Content { get; set; }
            public string ContentType { get; set; }
            public bool IsPublic { get; set; }

        }
    }
}
```

Ukázka kódu 1 - Programovací jazyk C# [vlastní zpracování]

2.5 ReactJs

ReactJs je JavaScriptová knihovna vyvinutá společností Meta (Facebook). Jejím prvotním účelem bylo poskytnou optimalizované řešení pro komplexní webové aplikace, které neustále pracují s velkým objemem dat a vyžadují časté aktualizace. Takové aplikace jsou často založeny na principu architektury MVC (Model View Controller). Aplikace, které tento model využívají, se mohou postupem času stát velmi objemnými. To vede k velké provázanosti jednotlivých modelů a následné neudržitelnosti. U těchto aplikací je spoléháno především na výkon webového

prohlížeče na koncovém zařízení. To může následně způsobit pomalé načítaní či neschopnost aplikace fungovat správně.

ReactJs umožnil nahrazení těchto křehkých a vysoce propojených aplikací pomocí javascriptových funkcí a nových paradigmat, které de facto mění principy vývoje pomocí javascriptu a knihoven založených na tomto programovacím jazyce. Přišel s novým mechanismem, který nahrazuje stav při změně dat. Díky tomu se z něj stal účinný nástroj pro práci s rozsáhlým uživatelským rozhraním, kde se často mění data.

Přichází také s novými funkcemi, jako jsou virtuální DOM a syntaxe JSX.

Princip Reactu je založen na tvorbě deklarativních komponent. Tyto komponenty je možné snadno upravovat a kombinovat s ostatními. Díky tomu je lehce škálovatelný. Při tvorbě aplikace v ReactJs je tedy důležité vytvářet malé komponenty a ty následně strukturovat tak, že se budou skládat dohromady. [8]

2.5.1 JSX

JSX je syntaktické javascriptu. Využití tedy nachází při tvorbě frontedových aplikací převážně za pomocí ReactJs. Jedná se o velmi podobnou syntaxi dobře známého HTML tagovacího jazyka a stejně jako HTML definuje, jak má vypadat uživatelské prostředí.

```
"use client"
import { useState } from 'react'

export default function Home() {

 const [ints, setInts] = useState(() => {
 console.log('array');
 return [1 ,2 ,3 ];
 });

 const [count, setCount] = useState(4 );

 function ListItems({ints, addValue}){
 return(
  <>
  <button onClick={addValue}>Add Items</button>
  {
   ints?.map((val) => {
   return(
    <li key={val}> VAL: {val} </li>
   )
   })
  }
  </>
 )
```

```
 }

 function addValue(){
 const newVal = Math.max(...ints) + 1 ;
 setInts([...ints, newVal]);
 console.log(ints);
 }

 function incrementCount(){
 setCount(prevCount => prevCount + 1 );
 }

 function decrementCount(){
 setCount(val => val - 1 );
 }

 function CounterItem({count}){
 return(
  <>
  <div>
  <button onClick={incrementCount}>+</button>
  <span>{count}</span>
  <button onClick={decrementCount}>-</button>
  </div>
  </>
 );
 }

 return (
 <div>
  <ul>
  <ListItems ints={ints} addValue={addValue} />
  </ul>
  <div>
  <CounterItem count={count} />
  </div>
 </div>
 );
}
```

Ukázka kódu 2- JSX syntaxe [vlastní zpracování]

2.6 Principy vývoje softwaru

Jedním z nejdůležitějších aspektů při vývoji nového softwaru je jeho struktura a návrh. Je tedy dobré dodržovat důležité principy, které do budoucna určují, jak dobře se s daným softwarem bude pracovat z hlediska vývoje a architektury softwaru.

Tyto principy určují budoucí náročnost pro úpravy a rozšíření softwaru stejně tak jako výši nákladů potřebnou pro vývoj. Použití nevhodných principů může způsobit vysokou komplexitu softwaru, neudržitelnost vývoje a údržby.

Dle průzkumu z podnikového prostředí má většina softwarových inženýrů povědomí o těchto principech, ale ne vždy je umí správně využívat v praxi.

Porušením některých těchto principů je známo pod termínem „design smells“. Tyto design smells je dobré refaktorovat, aby se docílilo správného využití principů. Z principů vývoje jsou nadále odvozeny softwarové návrhové vzory a principy refactoringu softwaru. [9], [10]

2.6.1 Don’t repeat yourself – DRY

„Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.“ - *„Každá znalost musí mít jedinou, jednoznačnou, autoritativní reprezentaci v rámci systému.“* [11 - s. 80]

Princip DRY – Don’t repeat yourself, tj. „Neopakuj se“, vychází z myšlenky, že duplikace a opakování je zbytečné plýtvání úsilím a zároveň zvyšuje komplexitu a nepřehlednost kódu.

K porušení tohoto principu může dojít například v případě, že vývojář plně nezná aktuální řešení, či v případě, že neví, jak správně použít abstrakci k řešení problému. Další náchylnost k porušení může být prostým vkládáním kopírovaného kódu do jiných tříd místo toho, aby vývojář přepsal dané řešení tak, aby požadovaná funkcionalita byla dostupná na více místech. [9]

Příkladem duplikace je implementace stejné logiky na více místech zároveň. Mohou být metody se stejným tělem. Pokud dojde ke změně logiky této metody, pak vývojář nesmí zapomenout provést tyto změny v každé duplikované metodě. Pokud tak neučiní, dojde k odlišné funkcionalitě, a tudíž se program může chovat pokaždé jinak, či dokonce může skončit chybovou hláškou.

K rozpoznání porušení tohoto principu lze docílit za pomocí těchto otázek: Pokud se mění logika v kódu, je potřebu tuto změnu provést ve více různých místech? Musí se měnit dokumentace či databázové schéma? Pokud tomu tak je, tak tento zdrojový kód porušuje princip DRY. [11]

```
public class Account
{
    public double Debits { get; set; }
    public double Credits { get; set; }
    public double Balance { get; set; }
    public double Fees { get; set; }
}


public class DRYViolation
{
    public void PrintBalance(Account account)
    {
        Console.WriteLine($"Debits: {account.Debits}");
        Console.WriteLine($"Credits: {account.Credits}");
        Console.WriteLine();

        if (account.Fees < 0 )
        {
            Console.WriteLine($"Fees: - {account.Fees}");
        }
        else
        {
            Console.WriteLine($"Fees: {account.Fees}");
        }

        Console.WriteLine();

        if (account.Balance < 0 )
        {
            Console.WriteLine($"Fees: - {account.Balance}");
        }
        else
        {
            Console.WriteLine($"Fees: {account.Balance}");
        }

        Console.ReadLine();
    }
}
```

Ukázka kódu 3 - Porušení principu DRY v C# [vlastní zpracování]

Výše uvedený kód znázorňuje porušení principu DRY. Konkrétně se jedná o duplikovaný IF, který slouží k porovnání hodnot. Aby tento kód splňoval princip DRY, musí se refaktorovat tak, že duplikovaný IF bude ve vlastní metodě a ta přebírá v parametru hodnotu, která se porovnává.

Následující ukázka kódu znázorňuje tuto úpravu.

```
public class DryCorrect
    {
        public void PrintBalance(Account account)
        {
            Console.WriteLine($"Debits: {account.Debits}");
            Console.WriteLine($"Credits: {account.Credits}");
            Console.WriteLine();

            this.PrintFormatedAmout(account.Fees);

            Console.WriteLine();

            this.PrintFormatedAmout(account.Balance);

            Console.ReadLine();
        }

        public void PrintFormatedAmout(double amout)
        {
            if (amout < 0 )
            {
                Console.WriteLine($"Fees: - {amout}");
            }
            else
            {
                Console.WriteLine($"Fees: {amout}");
            }
        }
    }
```

Ukázka kódu 4- Náprava principu DRY v C# [vlastní zpracování]

2.6.2 Keep it simple, stupid – KISS

Tento princip, v překladu *„Udržujte to jednoduché, hloupé.“*, je podobný principu DRY. Stojí za myšlenkou jednoduchosti. Kód by měl být psán co nejvíce jednoduše, aby se zamezilo zbytečné komplexitě. Díky dodržování tohoto principu lze redukovat problémy, které by mohly vzniknout. [9]

KISS princip se zabývá stylem psaní zdrojového kódu. Ten by se měl skládat ze smysluplných názvů funkcí, proměnných a celkově dodržovat určitý stylistický styl. [12]

2.6.3 SOLID principy

Jedná se o vlivný set pěti principů (SRP, OCP, LSP, ISP, DIP), které při správném použití zajišťují robustnost kódu. Jsou to principy, ne paradigma jako například objektově orientované programování, a díky tomu je lze využít nezávisle na programovacím jazyce. Celkově dodržování těchto principů vede k tomu, aby byl psaný kód čistější, srozumitelnější a byl umístěn na správném místě. [13]

2.6.3.1 Single responsibility principle – SRP

Princip SRP – Single responsibility principle neboli princip jediné odpovědnosti říká, že třída má být navržena tak, aby poskytovala pouze jednu funkčnost, za kterou je zodpovědná. Nemělo by dojít k existenci třídy, která by poskytovala více různých operací, které nesouvisejí s tím, na co je primárně navržena.

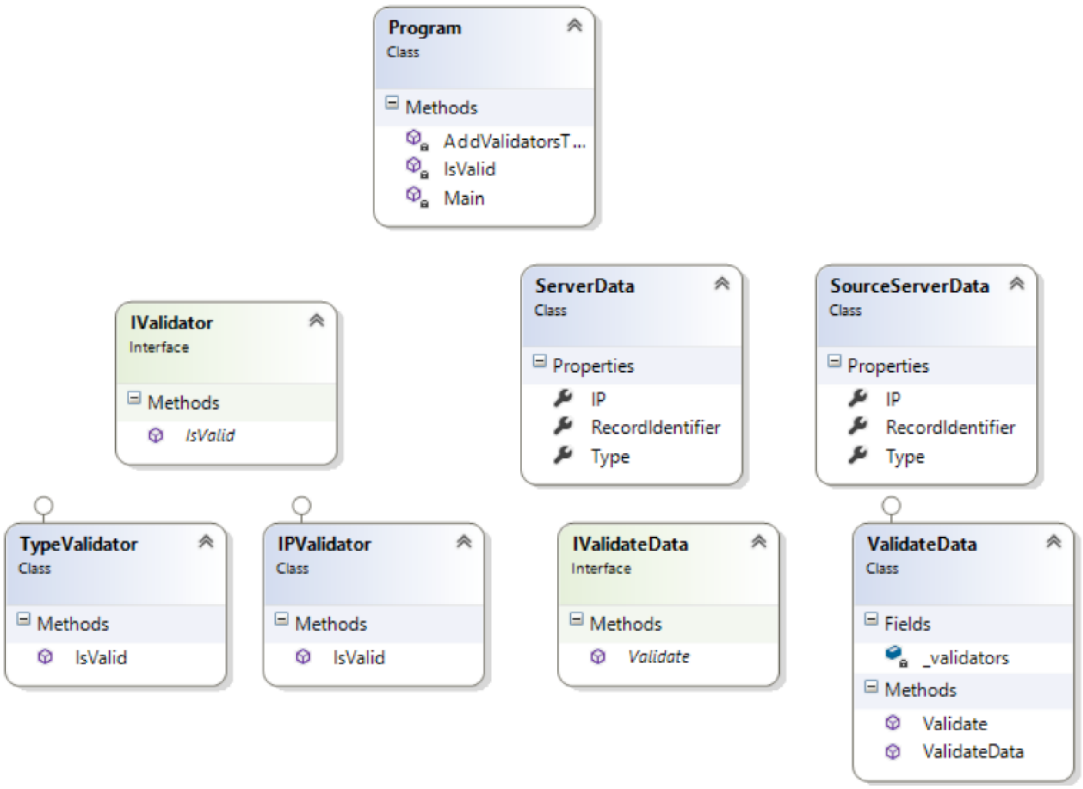
Příkladem může být program, který provádí CRUD (Create, Read, Update, Delete) operace. Kde by mohla existovat třída „DataContext“, která by implementovala každou z těchto operací jako samostatnou metodu. Pokud by vývojář vytvořil takovou třídu, porušil by princip SRP. Aby došlo k dodržení principu, musel by být program navržen tak, aby každá metoda měla vlastní třídu. Tedy třída Create bude obsahovat metodu Create a žádnou jinou veřejně dostupnou metodu. Takto musí být implementovány i ostatní metody (u každé z tříd zvlášť).

Při využití tohoto principu může dojít ke vzniku obrovského množství tříd, které mají relativně malé tělo. [13]

2.6.3.2 Open-closed principle – OCP

Princip OCP – Open-closed principle čili princip otevřenosti a uzavřenosti je způsob zacházení se základnou kódu. Řeší především rozšíření nebo změnu logiky aplikace. Vlastně říká to, že kód má být psán tak, aby byl otevřený pro změnu nebo implementaci nové logiky, ale zároveň musí být uzavřen vůči změně stávající logiky a nepřepisovat jí. V OOP lze tento problém řešit jednoduše za pomocí abstrakce. [13]

Příkladem může být validátor dat. Pokud by došlo ke změně datového modelu služby, která data zasílá, byl by tento problém řešen za využití této struktury aplikace:



Obrázek 1 - Ukázka struktury umožňující OCP [13]

Tato struktura umožňuje tvorbu libovolného počtu a druhu validátorů. Každý tento validátor musí implementovat veřejné rozhraní `IValidator` a implementovat vlastní logiku metody `IsValid`. Tím je docílena neměnitelnost již vzniklé logiky a lehká implementace nových validátorů.

Následně se díky abstrakci přidá každý validátor do listu ve třídě `ValidateData`, která implementuje rozhraní `IValidateData`. Díky implementaci tohoto rozhraní je opět docíleno možnosti tvorby nového data validátoru s jinou logikou. Třída `ValidateData` si následně projde list existujících validátorů (`IValidator`) a zavolá na něm metodu `IsValid`. Tím je docíleno spuštění jiné logiky pro každý validátor. [13]

```
namespace ukazkyBakalarka.Principles
{

    public interface IValidateData
    {
        bool Validate(ServerData data, SourceServerData sourceData);
    }


    public interface IValidator
    {
        bool IsValid(ServerData data, SourceServerData sourceData);
    }

    public class ServerData
    {
        public string IP { get; set; }
        public string Type { get; set; }
        public int RecordIdentifier { get; set; }
        //Other stuff goes here.
    }

    public class SourceServerData
    {
        public int Id { get; set; }
        public string Type { get; set; }
        public string IP { get; set; }
        public int RecordIdentifier { get; set; }
        //Other stuff goes here.
    }

    public class OCP
    {
        public OCP()
        {
            //Only for demonstration purposes.
            var sourceServerData = new List<SourceServerData>();
            var serverData = new List<ServerData>();
            foreach (var data in serverData)
            {
                var sourceData = sourceServerData.FirstOrDefault(s =>
                s .RecordIdentifier == data.RecordIdentifier);
                var isValid = IsValid(data, sourceData);
                Console.WriteLine("Record with Id {0 } is {1 }",
               data.RecordIdentifier, isValid);
            }
            Console.ReadLine();
        }

        private bool IsValid(ServerData data, SourceServerData sourceData)
        {
            List<IValidator> validators = AddValidatorsToValidate();
            IValidateData validateData = new ValidateData(validators);
            return validateData.Validate(data, sourceData);
        }

        private static List<IValidator> AddValidatorsToValidate()
        {
            return new List<IValidator>
            {
                new IPValidator(),
                new TypeValidator()
            };
        }
    }
```

```
    public class TypeValidator : IValidator
    {
        public bool IsValid(ServerData data, SourceServerData sourceData)
        {
            return data.Type == sourceData.Type;
        }
    }

    public class IPValidator : IValidator
    {
        public bool IsValid(ServerData data, SourceServerData sourceData)
        {
            return data.IP == sourceData.IP;
        }
    }

    public class ValidateData : IValidateData
    {
        private readonly IEnumerable<IValidator> _validators;
        public ValidateData(IEnumerable<IValidator> validators)
        {
            _validators = validators;
        }
        public bool Validate(ServerData data, SourceServerData sourceData)
        {
            return _validators.Any(validator => validator.IsValid(data, sourceData));
        }
    }
}
```

Ukázka kódu 5- C# převzato z [13] a upraveno, OCP

2.6.3.3 Liskov substitutional principle – LSP

Liskov substitutional principle se zabývá nahraditelností. Vlastně říká, že objekt S, který je podtypem T, pak může být nahrazen libovolným objektem typu S. A to tak aniž by se změnily vlastnosti tohoto typu. Jednoduše řečeno, rodič by měl být nahraditelný jeho libovolným potomkem. [13]

Pokud tedy vytvoříme třídu, která implementuje rozhraní nebo rozšiřuje třídu, musí se chovat dle očekávání. Jak je zde naznačeno, princip využívá principy dědičnosti.

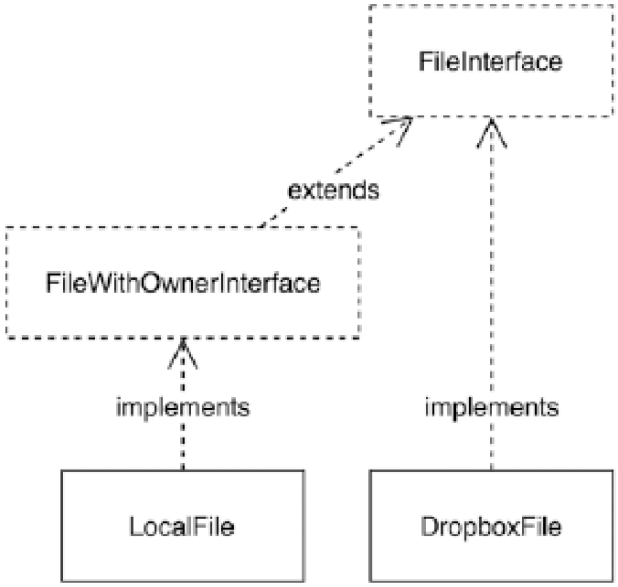
```
namespace ukazkyBakalarka.Principles
{
    public class LSP
    {
        private List<IFile> files;
        public LSP()
        {
            files = new List<IFile>() {
                new WindowsFile("system32.dll", "Microsoft Corp."),
                new CloudFile("OnedriveFile.txt")
            };
        }
    }
    public interface IFile
    {
        void RenameFile(string name);
    }

    public interface IFileWithOwner : IFile
    {
        void ChangeFileOwner(string owner);
    }

    public class WindowsFile : IFileWithOwner
    {
        public string FileName { get; private set; }
        public string FileOwner { get; private set; }
        public WindowsFile(string fileName, string fileOwner)
        {
            FileName = fileName;
            FileOwner = fileOwner;
        }
        public void RenameFile(string name)
        {
            this.FileName = name;
        }
        public void ChangeFileOwner(string owner)
        {
            this.FileOwner = owner;
        }
    }

    public class CloudFile : IFile
    {
        public string FileName { get; private set; }

        public CloudFile(string fileName)
        {
            FileName = fileName;
        }

        public void RenameFile(string name)
        {
            this.FileName = name;
        }
    }
}
```

Ukázka kódu 6- C# převzato z [14] a upraveno, LSP

Ukázka znázorňuje využití LSP v případě využití souborů. Rozhraní `IFile` specifikuje implementaci metody `RenameFile (string name)`. Následně rozhraní `IFileWithOwner` rozšiřuje `IFile` o metodu `ChangeFileOwner (string owner)`.

Příkladem užití může být soubor v prostředí Windows, kde je možné měnit jak název, tak vlastníka souboru. Pokud bychom ovšem měli například soubor z cloudové služby, tak by nám nemusela být umožněna změna vlastníka. Proto jsou v ukázce dvě rozhraní. Základní rozhraní `IFile` je použito jako datový typ pro list souborů. Díky dědičnosti je umožněno dodržet princip LSP a to tak, že do listu je uložen soubor typu WindowsFile implementující změnu názvu i vlastníka, ale je zde uložen i soubor typu CloudFile, který implementuje pouze metodu pro změnu názvu. [14]

Je dodrženo tedy toto schéma:



Obrázek 2 - Schéma návrhu struktury pro využití LSP [14]

2.6.3.4 Interface segregation principle – ISP

Interface segregation principle neboli princip oddělení rozhraní stojí na názoru, že by objekty implementující určité rozhraní neměly implementovat rozhraní, které je nutí implementovat takové metody, jaké nejsou využity. Místo toho by se mělo takovéto rozhraní rozdělit na více samostatných rozhraní tak, aby se zamezilo zbytečné implementaci nevyužitých či „prázdných“ metod.

Stejně tak to platí pro přidání nové funkcionality. Pokud by došlo k rozšíření aplikace a nutnosti přidat novou metodu nebo odebrat již existující, tak rozraní, které zajišťuje aktuální funkcionalitu, musí zůstat netknuté. Mohly by totiž nastat problémy v případě, že by tyto rozhraní byly využity jako externí knihovna ve více

aplikacích. Pokud by se měnilo existující rozhraní, musely by se měnit i všechny aplikace, které je využívají.

Řešením je přidat novou funkcionalitu do nového rozhraní. Následně díky možnosti implementace více rozhraní zároveň je docíleno zachování předchozí funkcionality a přidání nové funkcionality. Nebo lze využít i rozšíření nového rozhraní o původní, tím se docílí stejný efekt. [13]

Jinak míněné vysvětlení zní: „Rozhraní má být na míru službě či klientovi." Nemá obsahovat nadbytečné metody. Tím může vzniknout velké množství rozhraní o pár metodách, ale neměl by nastat případ, kdy se implementuje metoda, jenž nebude využita. [14]

2.6.3.5 Dependency inversion principle – DIP

Tento princip, v překladu „*Princip inverze závislostí*", určuje pravidlo, kdy závislosti musí být závislé na abstrakci, a ne na konkretizaci. [14]

```
namespace ukazkyBakalarka.Principles
{
    public interface INotifier
    {
        void SendMessage(string title, string message, Person person);
    }

    public class EmailNotifier : INotifier
    {
        public void SendMessage(string title, string message, Person person)
        {
            //sendgrid/exchange connection...
            System.Diagnostics.Trace.WriteLine($"E-mail has been send to: {person.E-mail}");
        }
    }

    public class SmsNotifier : INotifier
    {
        public void SendMessage(string title, string message, Person person)
        {
            //sms gateway connection...
            System.Diagnostics.Trace.WriteLine($"SMS      has      been      send      to:
+{(int)person.CountryCode} {person.Phone}");
        }
    }

    /// <summary>
    /// Třída pro pozvání osoby na pohovor.
    /// Ukazuje správné použití DIP.
    /// </summary>
    public class InerviewInviter
    {
        private List<INotifier> notifiers;

        public InerviewInviter(List<INotifier> notifiers)
        {
            this.notifiers = notifiers;
```

```
        }

        public void InvitePerosonToInterview(Person person)
        {
            //create interview object etc...

            //Správné využití DIP
            //Prochází celý seznam a na každém notifikátoru zavolá SendMessage()
            //Využívá abstrakci díky rozhraní INotifier, není tudíž závislá na hodnotě.
            this.notifiers.ForEach(x =>   x .SendMessage("Interview   Invitation",   $"Dear
{person.GetFullName()} we would like you to invite...", person));
        }
    }
    /// <summary>
    /// Třída pro pozvání osoby na pohovor.
    /// Třída s příkladem porušení DIP. Zároveň nevyužívá správně abstrakci.
    /// </summary>
    public class InterviewInviterViolationDIP
    {
        private SmsNotifier smsNotifier;
        private EmailNotifier emailNotifier;
        public    InterviewInviterViolationDIP(SmsNotifier    smsNotifier,    EmailNotifier
emailNotifier)
        {
            this.smsNotifier = smsNotifier;
            this.emailNotifier = emailNotifier;
        }

        public void InvitePerosonToInterview(Person person)
        {
            //create interview object etc...

            //Porušení DIP ==> odeslání je závislé na hodnotě smsNotifier a emailNotifier
            if (smsNotifier != null)
            {
                this.smsNotifier.SendMessage("Interview         Invitation",         $"Dear
{person.GetFullName()} we would like you to invite...", person);
            }
            if (emailNotifier != null)
            {
                this.emailNotifier.SendMessage("Interview        Invitation",        $"Dear
{person.GetFullName()} we would like you to invite...", person);
            }
        }
    }
    public class Person
    {
        public string E-mail { get; set; }
        public string Phone { get; set; }
        public string Name { get; set; }
        public string Surname { get; set; }
        public CountryCode CountryCode { get; set; }
        public string GetFullName()
            => $"{this.Name} {this.Surname}";
    }
    public enum CountryCode {
        CZ = 420,
        SK = 421,
        DE = 49,
        FR = 33,
        PL = 48,
    }
}
```

Ukázka kódu 7 - Příklad principu DIP v C# [vlastní zpracování]

Kód výše vyznačený má sloužit pro účely notifikace kandidáta v případě, že byl pozván na pracovní pohovor. Jsou zde SMS a e-mail notifikátory (`SmsNotificator`, `EmailNotificator`). Obě tyto třídy implementují společné rozhraní INotifier.

V případě využití třídy `InterviewInviterViolationDIP`, lze vidět v těle metody `InvitePersonToInterview`, že se rozhoduje pomocí klauzule `IF` na základě existující instance dostupných notifikátorů. Pokud existuje její instance, tak se zároveň na této instanci zavolá metodu pro odeslání zprávy dané osobě. Zde dochází k porušení DIP principu, jelikož se metoda rozhoduje na základě instance.

Řešení a správné využití DIP je znázorněno v podobné třídě `InerviewInviter`. Ta obsahuje list notifikátorů na základě jejich společného rozhraní `INotifier`. Následně se v metodě `InvitePersonToInterview` odešle notifikace každému notifikátoru v seznamu. Neřeší se zde žádná konkrétní instance notifikátoru, jediné, na čem záleží je rozhraní `INotifier`. Zde byla využita abstrakce k odeslání notifikace. [15]

2.7 Testování softwaru

Testování softwaru jsou různé postupy, které jsou využity k ověření funkčnosti aplikací. Cílem těchto postupů je odhalení chyb nebo problémů v softwaru a jejich následné odstranění. Testeři kód ověřují tak, že kód aplikace zpochybňují. Testování probíhá na základě určených funkčních a nefunkčních požadavků. Cílem by tedy mělo být splnění všech těchto požadavků a od toho se následně odvíjí kvalita software.

Software bývá testován především při konečné fázi vývoje, tedy předtím než dojte k nasazení softwaru do produkčního prostředí. To ovšem není podmínkou, testuje se i během jednotlivých fází vývoje. [16]

„*Testing is the process of executing a program with the intent of finding errors.*“ V překladu „*Testování je proces spouštění programu se záměrem najít chyby.“* [17]

Dalším označením pro softwarové testování může být proces validace a ověřování, zda aplikace či software splňuje funkční/nefunkční business požadavky, dle kterých byl řízen vývoj softwaru. Proces validace a ověření zajišťuje dodržení požadavků dle vývojové dokumentace SRS (Software Requirement Specification). [18]

Dodání softwaru s chybou může nenávratně poškodit pověst značky, či v horším případě může chyba v softwaru degradovat i systémy propojené s vadným softwarem a způsobit tak rozsáhlé a nákladné škody.
Důležitost testování softwaru je vidět na historických událostech. Příkladem může softwarová chyba airbagového systému v automobilech Nissan, kdy v roce 2016 muselo být svolány na servisní zásah miliony vozů. Oprava chyb jako je tato zmíněná, stojí následně biliony dolarů a ovlivní miliony zákazníků. Proto je velice důležité software před vydáním dobře otestovat. [19]

2.7.1 Základní terminologie testování softwaru

V této části autor podrobně vysvětluje klíčové pojmy a termíny související s testováním softwaru.

2.7.1.1 Test case – testovací případ

Testovací případě je dokument, který specifikuje očekávané výsledky jednoho konkrétního testu. Obsahuje specifikace vstupů, podmínek a postupů provedení. Tento dokument zajišťuje vyhodnocení všech oblastí programu, tak aby nebyly přehlédnuty žádné chyby. [20]

2.7.1.2 Test suite – sada testů

Jedná se o posloupnost jednotlivých testovacích případů. Jsou vytvořeny tak, aby se automaticky provedly a u každého z testů se následně zobrazí výsledek – vyhověl nebo nevyhověl. Výhodou těchto sad je možnost provádět testy opakovaně a automatizovaně, aniž by člověk musel interagovat s testovanou aplikací. Zároveň zabezpečují proces testování před lidskou chybou. [20]

2.7.1.3 Bug – chyba

Bug je chyba, která může způsobit pád programu nebo navrácení nesprávné výstupové hodnoty. Bug bývá způsoben chybnou či nedostatečnou logikou, která může způsobit selhání nebo odchylku od očekávaného výstupu. Bugem může být nazýván omyl nebo chyba vývojáře při vývoji aplikace. [20]

2.7.2 Druhy testů

Testování softwaru je rozděleno do široké škály kategorií a druhů, které se zaměřují k ověření různých aspektů kvality aplikací. V této části je představení základních druhů testů.

2.7.2.1 Black-box testování

Při využití black-box testovací metodologie nahlíží na testovaný software jako na černou skříňku. Tudíž se nenahlíží na vnitřní strukturu a implementaci softwaru, ale pouze na vstupy a výstupy, které software přijímá a generuje. Cílem tohoto testu je hledání určitého chování, kdy se testovaný software nechová tak, jak by měl.
Nevýhodou této metodologie testování je ovšem množství testovacích případů. Pokud se má využít black-box testing k nalezení všech existujících chyb, tak může vzniknout až nekonečné množství testových případů. Důvodem je, že je potřeba testovat nejen všechny validní vstupy, ale také všechny programové vstupy, které by mohly existovat. Z toho lze následně odvodit, že tuto strategii testování nelze využít k testování programu tak, aby byl zcela bezchybný. Zároveň je nutné brát ohled i na ekonomickou stránku testování. [17]
Jak již bylo zmíněno, tento způsob testování umožňuje identifikovat chyby pouze podle projevení chyby až na výstupu. [21]

2.7.2.2 White-box testování

White-box testovací metodologie využívá opačného pohledu na testovaný software než black-box metodologie. Jedná se o testy cíleny na logiku a strukturu aplikace. Tudíž je tato metodologie závislá na přímé znalosti zdrojového kódu. Jedná se tedy spíše o testování již vzniklé logiky namísto testování, zda aplikace splňuje cílové specifikace. [17]
White-box testing zahrnuje testování veškeré aplikační logiky, jako jsou smyčky, podmínky, toky dat a vnitřní datové struktury tak, aby byla zaručena platnost a validita dat i logiky. Tyto testy by měly být navrženy tak, aby otestovaly každou cestu programem minimálně jednou. [21]

2.7.2.3 Debugging

Debuggingem neboli laděním nazýváme vícestupňový proces identifikování a izolování problémů vzniklých v kódu aplikace. Jeto účinný nástroj pro odstranění chyb a porozumění logice kódu. [22]

Řadí se pod sekci testování softwaru, kde je často využit, pokud některý z testů selže. Často je využit v kombinaci s unit testy. Ovšem nemá stejný cíl jako softwarový test, který má za úkol zjistit, zda je v aplikaci chyba. Oproti tomu ladění zjišťuje místo výskytu této chyby. [17],[22]

Princip softwarového ladění je založen na umístění bodu přerušení (break point) na určitém řádku kódu. Je proto tedy nutné umět chybu opakovaně simulovat, aby bylo možné přibližně určit místo, kde je potřeba umístit tento bod. Následně, když je spuštěna aplikace v režimu ladění, dojde k bodu přerušení, a tak pozastaví svůj běh. Vývojář nebo tester má možnost si prohlédnout hodnoty proměnných uložených v paměti aplikace a v některých případech má možnost tyto hodnoty ručně změnit. Některá IDE umožňují i změny zdrojového kódu v případě přerušení a jejich kompilaci v reálném čase tak, aby se změna projevila už v průběhu ladění.

Ladění následně umožňuje krokování. To znamená, že lze kód aplikace procházet řádek po řádku i s možností vnoření se do volaných metod a zkoumat, jak funguje logika aplikace.

Díky tomuto nástroji lze detailně porozumět kódu a jeho logice, tím se snadno docílí nalezení chyb a následného odstranění jejich příčin. [22]

2.7.2.4 Unit testování

Unit testování nachází využití u větších aplikací. Jak je již v názvu naznačeno, zaměřuje se na menší jednotky, které bude testovat. Těmito jednotkami rozumíme jednotlivé metody, třídy a podprogramy apod.

Výhody nachází unit testování v kombinaci s debuggingem. Díky zaměření jen na malou část aplikace se minimalizují možnosti k umístění bodu přerušení. Další výhodou může být možnost spouštění těchto testů paralelně nebo využít test ke kombinování prvků testování. [17]

Jelikož unit testy testují kód napřímo, tak jsou tvořeny samotnými vývojáři aplikací. To umožňuje využít metodologii testem řízený vývoj (TTD), který si zakládá

na neustálém testování již během počáteční fáze vývoje softwaru. Lze tedy považovat unit testování jako jednu z prvních možných metod, jak testovat software. Tento způsob testování nabízí jak manuální spouštění testů tak i automatizované.

Používání unit testů se často dělí do tří fází. První fází je příprava samotného unit testu. Následuje implementace kódu, který bude testován již existujícím testem. V poslední fázi dojde ke spuštění testu a ověření, že napsaný kód funguje tak, jak je požadováno v testu. Pokud test selže, dochází k refactoringu testovaného kódu aplikace, dokud není test splněn.

Nevýhodu těchto testů může být fakt, že existují případy, kdy se samotný unit test skládá z více řádků kódu něž kód, který testuje. Další nevýhodou je nemožnost odhalit chyby, které vzniknou v integraci s ostatními částmi aplikace, a nutnost zaškolení vývojářů pro tvorbu těchto testů a jejich využití za pomocí frameworků. [23]

2.7.2.5 Integrační testování

Integrační způsob testování již na rozdíl od unit testů kouká na software jako jeden kompletní modul. Je zaměřen na testování funkčnosti „kompletního“ softwaru, který je složen z více modulů. Testuje, zda závislost jednoho modulu na jiných modulech má validní výstupy a zda nedochází kvůli integraci těchto modulů ke vzniku nějakých chyb.

Pro tento typ testování je potřeba využít speciální moduly/aplikace, které jsou zaměřeny pro tento typ testování. Tyto moduly fungují tak, že nahrazují určité moduly v testovaném softwaru a simulují datovou komunikaci mezi moduly softwaru. Jako příklad lze uvést moduly jako Selenium, Junit, Steam, LDRA a Jasmine. Integrační testování lze rozdělit do čtyř klíčových přístupů, pomocí kterých se tyto testy provádějí. Těmi jsou big-bang, top-down, bottom-up a sandwitch neboli hybridní testování.

Big-bang strategie je zaměřena na testování všech integrovaných modulů najednou. Výhody využití nachází tato strategie především u menších systémů nebo při urychleném vývoji. Ovšem má i své nevýhody. Těmi jsou obtížnost lokalizovat místo chyby v aplikaci, časová náročnost u velkých systémů, nutnost dostupnosti všech

modulů zároveň v jeden okamžik a nemožnost testovat moduly s vyšší prioritou přednostně.

Top-down strategie využívá posloupnosti, kdy testování započne u nejvýše postavené úrovně modulu a postupně se dostává k nižším vrstvám. Každá úroveň nebo modul jsou testovány zvlášť a následně jsou integrovány, aby se ověřily funkčnost během této integrace. To přináší výhody jako je jednoduší nalezení chyb, prioritizace modulů pro testování a možnost vytvořit prototypy. Nevýhodami jsou především komplikace při testování větší množiny modulů a časová náročnost testování níže postavených modulů.

Bottom-up strategie je přesným opakem top-down strategie. Nejdříve jsou testovány moduly na nejnižší úrovni a postupně se proces integračního testování dostává k nejvyšší vrstvě. Své vhodné uplatnění nachází v případě, kdy již existují nějaké jednotky pro testování. Hlavní výhodou je tedy úspora času, jelikož není potřeba mít všechny moduly, a také přesnější lokalizace míst chyb. Naopak tomu může dojít ke komplikovanému testování, pokud se software skládá z velkého množství modulů nízké úrovně.

Sandwitch neboli hybridní strategie je kombinace bottom-up a top-down strategií. Díky tomu je vhodnější pro testování objemných aplikací či dokonce operačních systémů. Ovšem je složitější a nákladnější. [24], [25]

2.7.2.6 Systémové testování

Systémové testování stejně jako integrační testování kouká na software jako celek. Ovšem oproti integračním testům netestuje integraci modulů. Testuje software jako již hotový produkt. Proto se nachází až v pozdních fázích procesu vývoje. Lze na něj nahlížet jako na black-box testování, kdy je jeho účelem otestovat, zda software splňuje dané uživatelské, technické a funkční požadavky. K dosažení výsledků tohoto testování mohou být využity různé typy testování, jakou jsou například výkonnostní testy, zátěžové testy a funkční testy. [26]

2.7.2.7 Testování výkonu

Výkonnostní testování je využito k nalezení a odstranění míst tzv. bottlenecků, která jsou zodpovědná za nežádané omezení výkonu aplikace. Tento typ testování

probíhá většinou na produkčním prostředí nebo v podmínkách připomínající reálné prostředí. Důvodem je potřeba simulovat zatížení aplikace, sítě nebo hardwaru v reálných podmínkách. Tento typ testování obecně odstraňuje pomalou odezvu aplikací a jejich nekonzistenci, což vede ke zlepšení uživatelského zážitku.
Důležitost tohoto testování znázorňuje fakt, že bottlenecky mohou zpomalovat celý systém nezávisle na výkonnosti počítače na kterém je tento systém spuštěn.
K znázornění metrik je využito klíčové ukazatele výkonosti tzv. KPI (Key Performance Indicators). Ty často obsahují metriky, jako jsou datová propustnost, velikost a míra využití paměti, latenci, šířku pásma a počet hardwarových přerušení.
Existuje spousta metod, jak provést testování výkonu. Dvě hlavní metody jsou Load testing (zátěžový test) a Stress testing.
Load testing testuje určitou zátěž, kdy test simuluje očekávaný počet transakcí a uživatelů v jeden okamžik. Tento test trvá tak dlouho, aby vývojáři našli konkrétní místa, kde vzniká bottleneck. Díky definovanému počtu uživatelů a transakcí umožňuje tento test určit přibližný optimální počet uživatelů, které software zvládne zpracovat v jeden okamžik.
Stress test posouvá hranice výše než zátěžový test. Konkrétně jde přes softwarem zvládnutelné hranice a zkoumá, jak se software a hardware chová v případě přetížení. Stresové testování se zaměřuje na hardwarové zdroje a určuje potencionální bod zlomu aplikace v závislosti na využívání těchto zdrojů. Tento druh vede k odhalení problémů, jako jsou zpomalení datové komunikace, poškození dat, nedostatku paměti nebo dokonce k problémům se zabezpečením. [27]

2.7.2.8 Akceptační testování

Akceptační testování je zajišťování kvality. Určuje míru, do které aplikace vyhovuje požadavkům koncového zákazníka nebo uživatelů. Probíhá povětšinou jako beta verze dané aplikace. Výhoda je, že se k této verzi dostanou reální koncoví uživatelé. Následně poskytnou feedback, který určí, zda byl test splněn či nikoli. V případě neúspěchu tento feedback může pomoci vývojářům pochopit daný obchodní model, což umožní aplikaci upravit tak, aby splňovala požadovanou funkcionalitu.
Toto testování lze rozdělit na uživatelské akceptační testování (UAT), kde probíhá testování koncovými zákazníky či skupinou uživatelů, a na provozní akceptační

testování, které zajišťuje funkčnost procesů dle očekávání a zaměřuje se na udržitelnost a použitelnost aplikace. Tento typ testu se zaměřuje na zálohování dat a obnovení aplikace či dat v případě havárie. Samozřejmostí je i způsob, jak lze daný software bezpečně předat zákazníkovi. [28], [29]

2.7.3 Software pro automatizované testování

Automatizované testování hraje klíčovou roli v moderním vývoji softwaru. Umožňuje zvýšit efektivitu, spolehlivost a rychlost testovacích procesů. V této části jsou popsané některé z existujících řešení.

2.7.3.1 Selenium

Selenium je open-source testovací framework. Jeho účelem je automatizovaný testing UI webových aplikací. Skládá z více softwarových modulů, jako jsou Selenium IDE, WebDriver a Selenium Grid. Jedná se tedy o sofistikovaný nástroj, který je schopen testovat webové prostředí aplikací napříč různými webovými prohlížeči a různými platformami.

Jeho cílem je nahradit ruční testování UI webu funkčním kódem, který nevyžaduje součinnost člověka při testování. Umí například vyplňování formulářových prvků, klikání na tlačítka a navigování skrze webovou stránku.

Dá se využít pro různé druhy testování, jako jsou funkční testování, regresivní testování a akceptační testování. Lze jej využít i s jinými testovacími frameworky, jakou jsou JUnit, Nunit a Test NG. [30], [31]

2.7.3.1.1 WebDriver

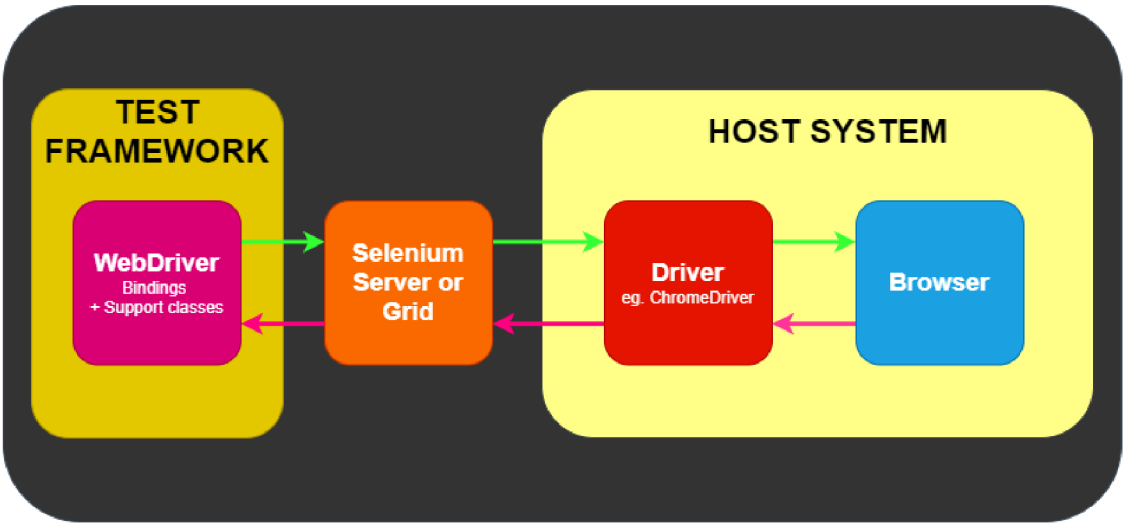
Jedná se o API rozhraní pro automatizaci prohlížeče. Ovládá prohlížeč, spouští testy, a to lokálně nebo pomocí vzdáleného přístupu. Ke své funkčnosti nevyžaduje žádnou další implementaci v testované aplikaci. Jeho chování přímo nahrazuje činnost uživatele na webu. Neví nic o testech, metodách a testovacích frameworcích. Jedná se pouze o propojovací prvek mezi frameworkem a webovým prohlížečem. Většinou bývá distribuován a vyvíjen poskytovatelem daného webového prohlížeče. [32], [33]

2.7.3.1.2 Selenium IDE

Prostředí Selenium IDE umožňuje tvorbu jednotlivých testovacích případů. Samotné prostředí je přístupné jako modul rozšíření pro prohlížeče Chrome a Firefox. Jeho účelem je zaznamenávání akcí uživatelů v prohlížeči pomocí skriptových Selenium příkazů a definovaných parametrů. [30], [31]

2.7.3.1.3 Grid

Grid komponenta je zaměřena na spuštění automatizovaných testů ve více prohlížečích, a to i v různé kombinaci operačních systémů. Umožňuje tedy spuštění testovacích případů na různých platformách. [30], [34]



Obrázek 3 - Komunikace Frameworku a aplikace za pomocí WebDriveru [32]

2.7.3.2 Cypress.io

Cypress je end-to-end, frontendový a open-source testovací nástroj. Narozdíl od Selenia se jedná o novější nástroj, který je vhodnější pro testování webových rozhraní založených především na SPA (Single Page Application). Tyto aplikace se skládají z komplexního JavaScriptu, většinou jsou tvořeny pomocí frameworků jako je React, Vue.js nebo Angular.

Samotný Cypress má jinou architekturu a jiná omezení než Selenium. Je založen na JavaScriptu, což umožňuje i jeho snadnou instalaci a implementaci. Kromě end-to-end testování podporuje i API testy pomocí HTTP požadavků anebo testování připojených komponent jiných frameworků. [35], [36]

Cypress má spoustu klíčových vlastností. Například:

- **„Cestování časem.“** - Cypress během testování pořizuje screenshoty aplikace. Je tedy možné zpětně sledovat postup testu a co přesně bylo zobrazeno.
- **Možnost ladění (debuggování)**. Díky rozšířenému logování chyb, umožňuje Cypress snadnější ladění javascriptových aplikací, což mnohonásobně urychlí hledání a opravu chyb.
- **Automatické čekání.** Díky automatickému čekání, které je přímo implementováno v knihovně Cypress, je aplikace zabezpečena vůči chybám v případě asynchronního průběhu aplikace. Cypress jednoduše čeká, než budou načteny všechny potřebné prvky a teprve pak pokračuje v testování.

Mezi další přednosti lze řadit schopnost analyzování chování funkcí, řízení síťového provozu, rychlé a konzistentní testování, nahrávání videí u nevalidních testů. [36]

2.8 Technologie CI/CD Pipelines

CI/CD je technologie umožňující automatizaci celého procesu vývoje a nasazení softwaru. Tyto zkratky zastupují hlavní pojmy continuous integration (kontinuální integrace), continuous delivery (kontinuální dodávka) a continuous deployment (kontinuální nasazení). Je nedílnou součástí DevOps či jiných postupů pro vývoj softwaru. Tato technologie zavádí průběžnou automatizaci a monitorování v celém životním cyklu aplikace. Zahrnuje fázi sestavení aplikace, testování, nasazení do produkčního prostředí až po fázi správy infrastruktury. Spojení těchto procesů se často označuje jako „CI/CD pipeline“. [37]

Správným využitím CI/CD technologie se minimalizuje doba nezbytná pro vydávání aplikací a zlepšuje se produktivita organizace, jež aplikaci vyvíjí. Vývojářské týmy díky této plné automatizaci získávají snadnější přístup k nasazení i malých změn v aplikacích a zároveň i lepší odezvu, díky čemuž se snižuje riziko chyb v průběhu sestavení aplikací a jejich nasazení. [38]

Ne vždy jsou využity všechny tři principy v „potrubí“. Principy lze využít i jednotlivě. Některé podniky začínají pouze s automatizovanou integrací a následně přidávají jak kontinuální doručení, tak i kontinuální nasazení. [37]

Obrázek 4 - Proces CI/CD pipeline [37]

2.8.1 Continuous Integration – CI

Kontinuální integrace je proces, který se stará o spojení veškerého kódu aplikace do jedné větve. Následně na tomto celku probíhá automatické testování jak jednotlivých změn, které do kódu přibyly, tak i celku. Výstupem tohoto procesu je kompletní sestavení aplikace po každém úspěšném průběhu tohoto procesu. Díky tomuto procesu se minimalizují chyby při slučování kódu v případě, že aplikaci vyvíjí více vývojářů zároveň. CI procesy by měly obsahovat systém pro správu verzí, jako je GIT, aby bylo možné zachovávat jednotlivé verze kódu. [37], [38]

2.8.2 Continuous Delivery – CD

Kontinuální dodávání je nerozlučné s předchozím CI procesem. Po finální fázi CI procesu převezme řízení CD proces, který ověřuje, zda výsledné sestavení obsahuje všechny potřebné balíčky a je připraveno pro nasazení do jakéhokoliv prostředí. Tento proces může zajišťovat infrastrukturu a finální nasazení do testovacího, produkčního nebo jiného prostředí. Výsledkem tedy není nasazení, ale stav, kdy je provozní tým schopen rychle a jednoduše nasadit ověřenou a plně funkční aplikaci. [37], [38]

2.8.3 Continuous Deployment

Kontinuální nasazení je finální proces automatizovaných procesů. Aby mohlo dojít ke spuštění tohoto procesu, je nutné, aby vývojový tým nastavil určitá kritéria. Při splnění všech kritérií se spustí tento finální proces, který aplikaci vystaví do produkčního prostředí. [38]

3 Praktická část a dokumentace aplikace

Praktická část práce popisuje jednotlivé části návrhu a implementace jednotlivých vrstev modelového řešení.

3.1 Požadavky praktické práce

Praktická práce se skládá z několika úloh. Tato práce musí pokrýt tyto požadavky:

1.) Funkce aplikace a ovládání:
   a. Aplikace představuje intuitivní uživatelské prostředí pro ovládání aplikace.
   b. Aplikace umožní registraci i přihlášení uživatele a editaci profilu.
   c. Aplikace umožní prohlížení a filtraci v obsahu eventů.
   d. Aplikace umožní tvorbu vlastní události a její editaci.
   e. Aplikace umožní účast na události i opuštění události.
   f. Aplikace umožní tvorbu komentáře a jeho smazání.
   g. Aplikace umožní přidání i odebrání obrázku na vlastní událost či profil uživatele.
2.) Technické požadavky:
   a. Aplikace je rozdělena na dvě části – front-end a back-end.
   b. Front-end poskytuje uživatelské prostředí a komunikuje skrze HTTP požadavky s back-end aplikací v datové formě JSON.
   c. Back-end aplikace poskytuje funkční API prostředí a zpracovává požadavky přicházející z front-end aplikace. Zároveň umožňuje komunikaci s kterýmkoli jiným hostem za pomocí HTTP příkazů.
   d. Back-end aplikace zpracovává data a ukládá/čte data z MSSQL databáze.
   e. Pro front-end aplikaci existuje testový scénář možný automaticky a opakovaně spouštět teto test.
   f. Všechny části aplikace jsou nasazeny a spuštěni v prostředí Azure.
   g. Aplikaci je možné automatizovaně nasadit do prostředí za pomocí pipeline.

3.2 Vývojové prostředí

Za účelem vývoje byl využit počítač s operačním systémem Microsoft Windows 11 Pro. Vývoj back-end části aplikace probíhal převáže v IDE Microsoft Visual Studio edice professional. Částečně bylo pro vývoj back-end i konkurenční IDE JetBrains Rider.

Jako databázový modul byl využit lokální MSSQL server ve verzi Developer. V pozdější části vývoje back-end aplikace došlo k přesunu databáze na cloudové prostředí Azure se studentskou licencí. Ke správě a dotazování se na data přímo z databáze byl využit SQL Server Management Studio (SSMS) ve verzi 19.

Pro běh back-end aplikace je vyžadováno poslední aktuální LTS runtime (běhové prostředí) .NET 6 . Jelikož se jedná o multiplatformní prostředí a zároveň back-end aplikaci, je psán ve formě cloud native, je možné aplikaci spustit ve větší škále platforem, jako jsou Docker, IIS a podobně. Pro účely této práce je využit lokální IIS server a cloudové prostředí AppServices v Azure.

Front-end aplikace byl vyvíjen v prostředí Microsoft Visual Studio Code s pomocnými plugin moduly Prettier – Code formatter, indent-rainbow, ReactJs code snippets, Azure AppService, Azure Virtual Machines a Azure Tools.

Pro běh front-end aplikace je vyžadováno open-source běhové prostředí Node.js a to ve verzi 18.16.0.

Stejně jako back-end část aplikace byl i front-end ve finální části přesunut z lokálního počítače na cloudové prostředí Azure AppServices.

3.3 Návrh architektury

Jak je již výše zmíněno aplikace je rozdělena do několika vrstev – databázová vrstva, vrstva aplikačního back-end a aplikační front-end.

Každá tato část má svou vlastní strukturu.

3.4 Datový model – databáze

Databázový model se skládá dohromady z 23 tabulek.

3.4.1 Seznam tabulek databáze

1. Events
2. Comments
3. EventDetails
4. EventGroups
5. EventMembers
6. EventsOnEventTags
7. EventDetailOnAppFiles
8. EventRequirements
9. __EFMigrationsHistory
10. AppFiles
11. Countries
12. EventOnAppFiles
13. Currencies
14. EventTags
15. Groups
16. Regions
17. Requirements
18. Users
19. Locations
20. GroupMembers
21. GroupRequests
22. UserOnAppFiles
23. UsersOnUserRoles

3.4.2 Popis využitých tabulek v aplikaci

V konečné implementaci aplikace, pro účely této práce, nebylo nutno využít celý navržený datový model. Model je navržen tak, aby umožnil budoucí rozšiřitelnost. Tudíž v práci byl využit pevný datový základ pro budoucí rozšíření či vylepšení.

3.4.2.1 __EFMigrationsHistory.

Jak je již zřejmé z názvu, tato tabulka uchovává informace o jednotlivých změnách datového modelu dle specifikací v back-end aplikaci. Představuje historii jednotlivých změn a uchovává unikátní identifikátor „MigrationId“ každé změny. Tyto data lze použít pro navrácení databázového modelu do předchozích stavů, a to vše za pomocí knihovny Entity Framework Core.

3.4.2.2 AppFiles

Tabulka AppFiles ukládá veškerý souborový obsah nahraný uživatelem skrze aplikaci. Samotná tabulka neurčuje žádnou vazbu na jinou tabulku a pouze

uchovává soubory a jejich metadata. Obsah souboru jsou převážně obrázky v jakémkoli formátu. Souboru je aplikací převeden na posloupnost binárních dat, které lze v databáze uchovávat.

3.4.2.3 Users

Users tabulka uchovává základní informace o registrovaných uživatelích v systému, včetně hash jejich hesla pomocí, kterého lze uživatele ověřit. Tabulka Users je jednou z primárních entit. Většina ostatních tabulek obsahuje vazbu na tuto tabulku v podobě cizího klíče na ID uživatele. Tvoří tak vazbu 1:N . Důvodem těchto vazeb je uchovávání dat o vlastnictví a historii změn ostatních entit.

3.4.2.4 Events

Events tabulka uchovává základní informace ohledně události. Neobsahuje ovšem podrobnosti eventu.
Tabulka primárně uchovává název, kapacitu a datum začátku události. Zároveň obsahuje vazby typu 1:N vůči tabulkám EventDetails, Location, EventGroups.

3.4.2.5 Comments

Entity Comments je primitivní tabulkou obsahující vazbu 1:N vůči Events. Dle názvu se jedná o pouhý komentář s textem k určitému eventu.

3.4.2.6 EventMembers

Podobně jako Comments tato tabulka obsahuje vazbu 1:N na Events. Uchovává data o registrovaných účastnících k určité události.

3.4.2.7 Číselníkové tabulky

Číselníkové tabulky uchovávají převážně neměnný obsah, který ovšem díky databázové implementaci lze v případě potřeby jednoduše doplnit, a to povětšinou bez zásahu do zdrojového kódu aplikace.
Mezi tyto tabulky, v rámci této práce, byly zařazeny Currencies, Regions, Countries. Tabulky obsahují specifická data dle názvu.

3.4.2.8 Propojovací tabulky vazby typu M:N

Tabulky s vazbou typu M:N jsou ty tabulky, které obsahují cizí klíč vůči dvou různým entitám. Tímto lze zaručit několikanásobnou vazbu různých entit vůči sobě.
V rámci implementace se jedná o tabulky EventOnAppFiles, UserOnAppFiles, EventDetailOnAppFiles, UsersOnUserRoles.
Názvy těchto tabulek odpovídají vazbám na konkrétní entity, které propojují.

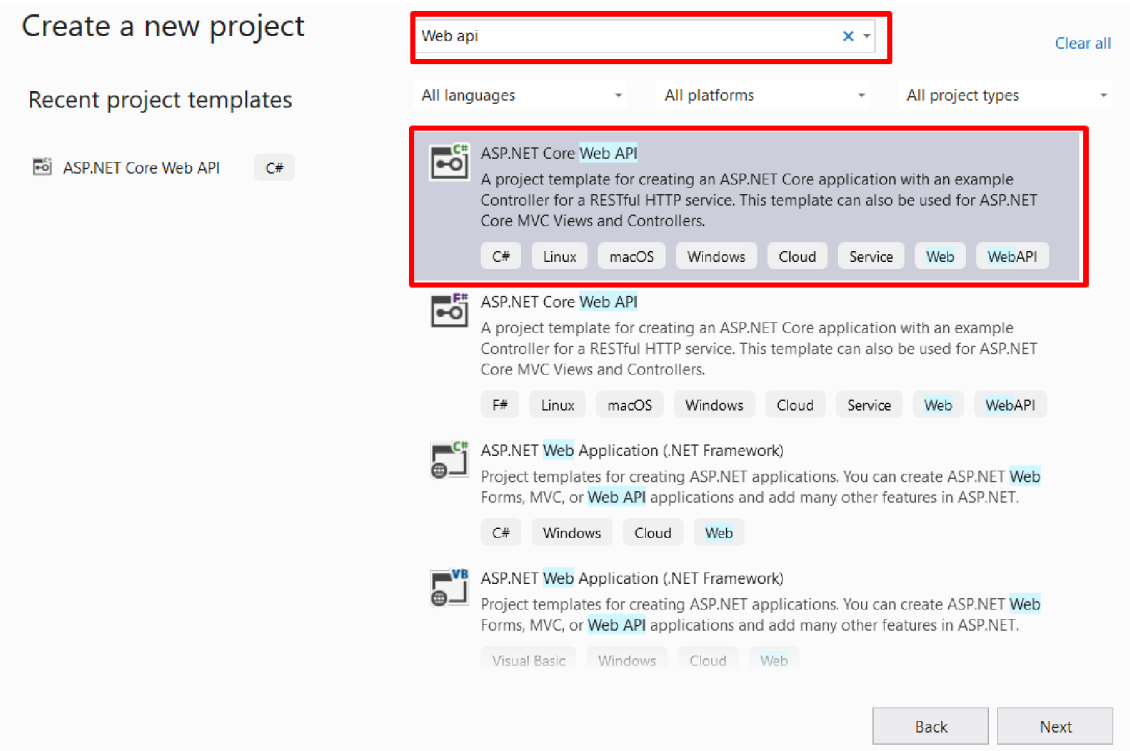
3.5 Back-end aplikace

Back-end je část aplikace zodpovědná za zpracování dat, logiku aplikace a komunikaci s databází. Tato část aplikace je povětšinou skryta před uživateli a pracuje v pozadí. Poskytuje tak potřebnou funkcionalitu prostřednictvím přijímáním, zpracováním a odesíláním požadavků front-end rozhraní. Tento přístup umožňuje oddělení logiky od uživatelského rozhraní aplikace.
Pro zpracování této práce byl zvolen pro back-end aplikaci .Net 6 web API projekt.

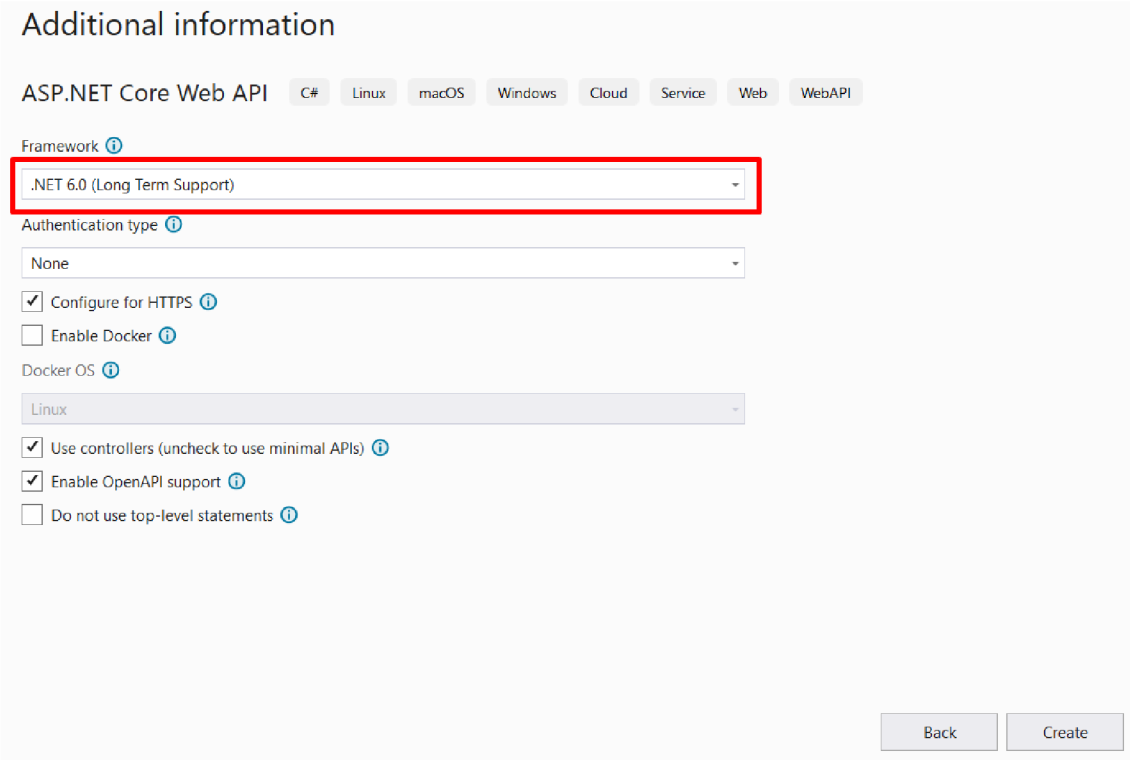
3.5.1 Založení a konfigurace projektu

Vývojové prostředí Microsoft Visual Studio lze stáhnout z webu společnosti Microsoft v bezplatné verzi jako exe, instalační balíček. Po následné instalaci a spuštění lze vytvořit web API projekt. Viz obrázky níže.

Tímto postupem byl založen prázdný projekt. Nyní je potřeba do projektu doinstalovat potřebné moduly tzv. NuGet.



Obrázek 5 - Založení nového Web API projektu, Visual Studio 2022 [vlastní zpracování]



Obrázek 6 - Volba cílového běžícího prostředí web API projektu [vlastní zpracování]

3.5.2 NuGet balíčky

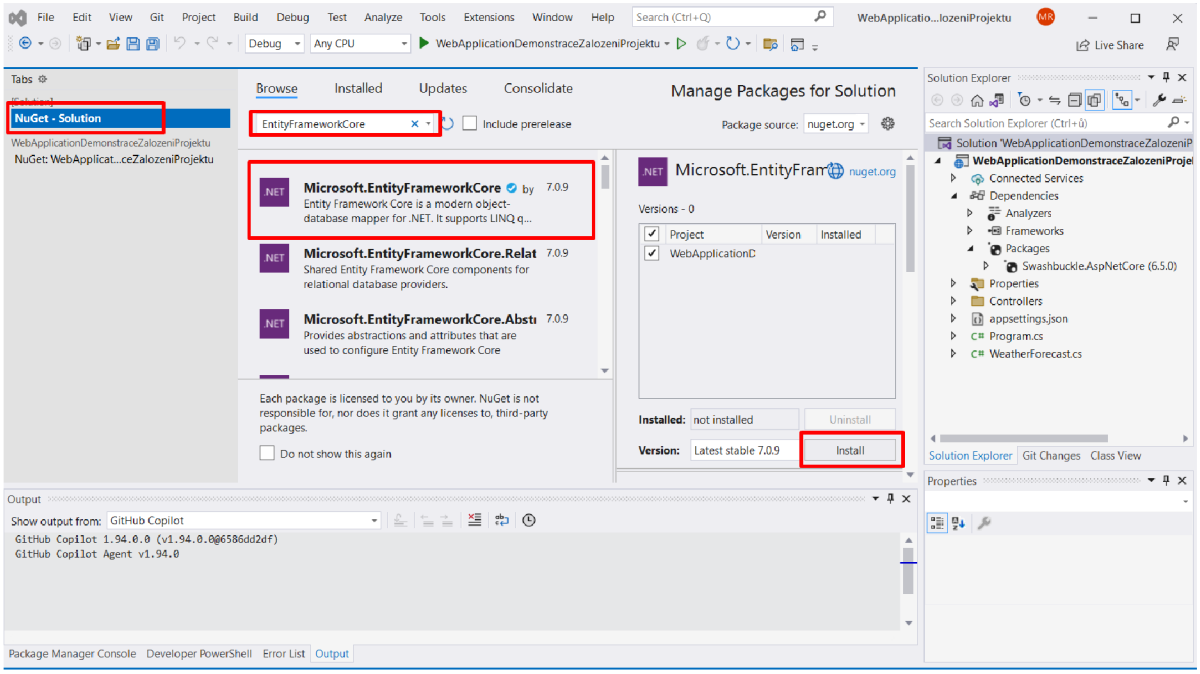
NuGet je rozšíření v rámci Visual Studia. Jedná se o otevřený systém správce balíčků, který umožňuje instalaci, aktualizaci nebo odebrání externích knihoven v rámci projektu.

V rámci této práce byly využity tyto externí NuGet knihovny:

- AutoMapper (12.0.1)
- AutoMapper.Extension.Microsoft.DependencyInjection (12.0.0)
- BCrypt.Net-Next (4.0.3)
- Microsoft.AspNetCore.Authentication.JwtBearer (6.0.0)
- Microsoft.AspNetCore.JsonPatch (6.0.15)
- Microsoft.AspNetCore.Mvc.NewtonsoftJson (6.0.15)
- Microsoft.EntityFrameworkCore (7.0.4)
- Microsoft.AspNetCore.SqlServer (7.0.4)
- Microsoft.EntitFrameworkCore.Tools (7.0.4)
- Microsoft.Extensions.Configuration (7.0.0)

- Microsoft.Extension.DependencyInjection (7.0.0)
- Microsoft.IdentityModel.Tokens (6.27.0)
- Microsoft.VisualStuio.Azure.Containers.Tools.Targets (1.17.2)
- Swashbuckle.AspNetCore (6.2.3)

Způsob instalace externího balíčku skrze NuGet správce. Viz následující obrázek



Obrázek 7 - Instalace externí knihovny pomocí NuGet správce balíčků [vlastní tvorba]

3.5.3 Architektura aplikace – REST API

Pro implementaci byla zvolena architektura REST API. Jedná se o standardizovaný způsob poskytování rozhraní pro komunikaci mezi různými systémy. Ke komunikace je využit protokol HTTP se standardními metodami `GET`, `POST`, `PUT`, `DELETE`. Každá zmíněná metoda zaručuje určitou manipulaci s daty. Tento návrhový vzor je známý převážně svou jednoduchostí a rozšiřitelností.

Implementace projektu se rozděluje do 4 vrstev. Těmi jsou Controller, Service, Model a Repository.

3.5.3.1 Controller

Controller je vrstva rozhraní obsahující koncové body (endpointy) pro komunikaci s klientem. Endpoint je reprezentován specifickou URL adresou s určitou HTTP komunikační metodou. Tato vrstva zpracovává požadavky klientů a předává je ke zpracování příslušným vrstvám aplikace. Po zpracování požadavku vrací endpoint odpověď klientovi.

```
[HttpPost("create")]
[ProducesResponseType(StatusCodes.Status200OK)]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
[ProducesResponseType(StatusCodes.Status401Unauthorized)]
[ProducesResponseType(StatusCodes.Status500InternalServerError)]
public async Task<IActionResult> CreateEventAsync([FromBody] CreateEventDto formData)
{
    var (success, resultId) =
    await _eventService.CreateEventAsync(formData, HttpContext.User.GetCurrentUserId());
    if (success)
    {
        Response.StatusCode = StatusCodes.Status201Created;
        return Ok(new DefaultResponseDto<int> { Success = true, Data = resultId });
    }

    return BadRequest(new DefaultResponseDto<object>
    { Success = false, Message = $"Error during creating event {formData}" });
}
```

Ukázka kódu 8 - POST endpoint v controlleru Events [vlastní zpracování]

3.5.3.2 Service

Service vrstva obsahuje veškerou aplikační logiku. Zpracovává a manipuluje s daty. Komunikuje s nižšími vrstvami, jako jsou modely či vrstva repository. Service vrstva přijímá konkrétní vstup a vrací zpět výstup, který je převážně předáván vrstvě Controller.

Na ukázce kódu níže lze vidět implementaci jedné ze základních služeb – `EventMemberService`.

Tato služba využívá Dependency Injection, tedy vkládání závislostí, pomocí parametrů konstruktoru služby. Tím je zaručena dostupnost požadovaných objektů. Příkladem může být `_eventMemberRepository`. Pomocí této vložené závislosti má služba přístup k rozhraní databázových operací, které jsou v této repository implementovány.

```
public class EventMemberService : IEventMemberService
{
    private readonly IEventMemberRepository _eventMemberRepository;
    private readonly IEventRepository _eventRepository;
    private readonly IMapper _mapper;

    public EventMemberService(
        IEventMemberRepository eventMemberRepository,
        IEventRepository eventRepository,
        IMapper mapper)
    {
        _eventMemberRepository = eventMemberRepository;
        _eventRepository = eventRepository;
        _mapper = mapper;
    }

    public async Task<IEnumerable<PublicUserDto>> GetList(int eventId,
        int page,
        int pageSize,
        bool active = true)
    {
        pageSize = pageSize <= AppConstants.MaxPageSize ? pageSize :
AppConstants.MaxPageSize;
        var data = await _eventMemberRepository.GetEventMembersUserAsyncByEventId(eventId,
page, pageSize, active);
        return _mapper.Map<IEnumerable<PublicUserDto>>(data);
    }


    public async Task<int> GetEventMembersCount(int eventId)
    {
        return await _eventMemberRepository.GetEventMemberCountByEventIdAsync(eventId);
    }


    public async Task<bool> JoinEventAsync(int eventId, int userId)
    {
        var eventEntity = await _eventRepository.GetEventByIdAsync(eventId);
        var currentCapacity = await
_eventMemberRepository.GetEventMemberCountByEventIdAsync(eventId);

        if (currentCapacity >= eventEntity.Capacity)
            throw new AppException("Event capacity is full");

        var isMember = await _eventMemberRepository.IsMemberOfEventAsync(userId, eventId);
        if (isMember)
            throw new Exception("Current user is already member of this event");

        return await _eventMemberRepository.CreateEventMemberAsync(eventId, userId,
userId);
    }

    public async Task<bool> LeaveEventAsync(int eventId, int userId)
    {
        var entity = await
_eventMemberRepository.GetEventMemberByEventAndUserIdAsync(userId, eventId);
        return await _eventMemberRepository.DeleteEventMembreAsync(entity);
    }
}
```

Ukázka kódu 9 – Třída EventMemberSerivice [vlastní zpracování]

3.5.3.3 Repository

Účelem vrstvy repository je abstrahovat přístup k datům a poskytnou rozhraní pro manipulaci s daty databáze. Tato vrstva skrývá podrobnosti o tom, jakým způsobem jsou data uložena a jak se manipuluje s databází. Díky této vrstvě se zajišťuje lepší oddělení odpovědnosti jednotlivých vrstev a modelů.

3.5.3.4 Model

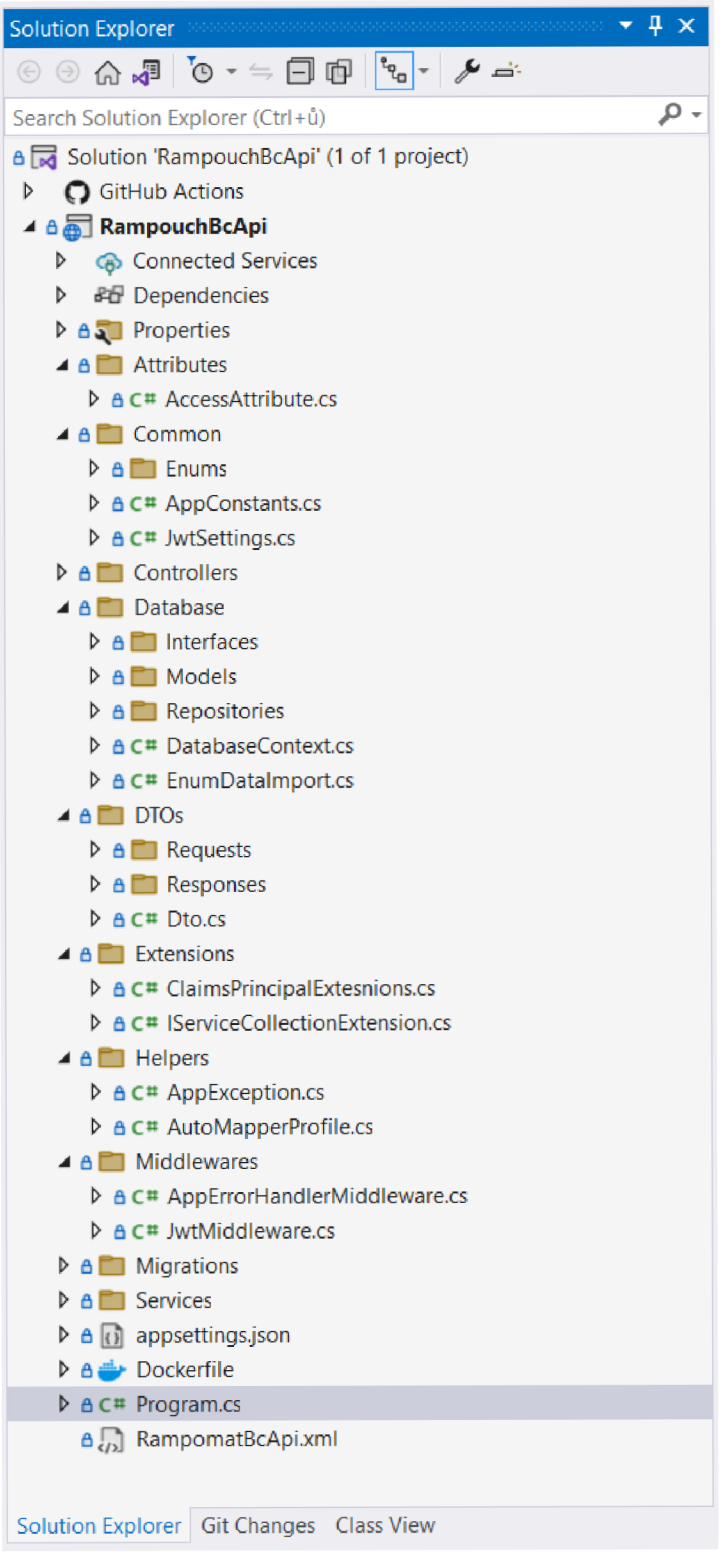
Model je vrstva představující datovou strukturu aplikace. Obsahuje třídy, které reprezentují datovou strukturu aplikace. Konkrétně v implementaci projektu odpovídá modelová třída tabulce v databázi.

```
[Table("Comments")]
public class Comment : Entity
{
    [Required]
    [ForeignKey("Event")]
    public int EventId { get; set; }

    [StringLength(SmallLength)]
    public string? Title { get; set; }

    [Required]
    public string Text { get; set; } = null!;

    public virtual Event Event { get; set; } = null!;
}
```

Ukázka kódu 10 – Modelová třída Comment [vlastní zpracování]

3.5.4 Souborové rozdělení projektu

Pro zjednodušení vývoje, zachování udržitelnosti a přehlednosti aplikace bylo potřeba zvolit logické uspořádání soubor ů v projektu. Uspořádání souborů projektu lze vidět na obrázku níže.



Obrázek 8 - Souborová struktura back-end aplikace [vlastní zpracování]

3.5.5 Využití Dependency Injection v rámci projektu

Dependency Injection (DI) je návrhový vzor umožňující oddělení a vkládání závislostí tříd. Díky této technice lze lépe dodržovat zásady SOLID, jejíž cílem je zlepšit znovupoužitelnost kódu.

DI pomáhá plnit tyto zásady tím, že odděluje vytváření a použití objektu. To umožňuje nahradit závislosti bez potřeby úpravy implementace třídy, která tyto závislosti využívá.

Díky DI lze dosáhnout plně odděleného návrhu aplikace od pevných vazeb mezi třídami. To vede k vyšší přehlednosti a udržitelnosti kódu. SOLID zásady jsou zásluhou DI výrazně zjednodušeny z důvodu, že většina moderních frameworků přesouvá odpovědnost za vytváření závislostí konkrétních tříd do DI kontejneru. Mezi tyto frameworky se řadí i.Net Core.

V rámci projektu byl tento návrhový vzor hojně využit. Především veškeré služby a repository jsou registrovány do DI kontejneru a následně framework dle potřeby tyto závislosti automaticky vkládá.

Pro správné použití tohoto vzoru je potřeba si vytvořit pro každou takovouto třídu rozhraní, které předepisuje jednotlivé dostupné metody. Následně je provedena implementace do třídy. Takto připravenou třídu lze následovně registrovat do DI kontejneru.

DI kontejner v .Net Core umožňuje více způsobů registrace. Rozdílem je převážně v přístupu, sdílení a vytváření jednotlivých závislostí.

Transient způsob registrace zaručuje vytvoření nové instance při každém vyžádání služby z DI kontejneru. Každý nový požadavek znamená tvorbu nové instance. Tento způsob nachází uplatnění v případě, že chceme dosáhnou úplné nezávislosti na ostatních službách.

Scoped způsob registrace vytváří pro každý HTTP požadavek jednu novou instanci požadované služby. To znamená, že pokud by si některá z navazujících tříd vyžádala vložení závislosti, tak bude sdílet jednu instanci.

Singleton způsob registrace vytváří v moment dotázání DI kontejneru na konkrétní instanci. Tato instance po celou dobu běhu aplikace je sdílena všemi žádajícími třídami.[39], [40]

V níže uvedené ukázce kódu je vidět, jakým způsobem se registrují jednotlivé služby. Tato část kódu se obvykle nachází přímo ve spouštěcí třídě `Program.cs`, kde se tyto služby registrují přes `WebApplication builder`. V případě této implementace je pro větší přehlednost kódu aplikace tato část vyňata z `Program.cs` do extension třídy a vlastní metody.

```
    private static void RegisterServices(IServiceCollection services)
    {
        services.AddTransient<ITokenService, TokenService>();
        services.AddTransient<IUserService, UserService>();
        services.AddTransient<ICommentService, CommentService>();
        services.AddTransient<IEventService, EventService>();
        services.AddTransient<IGroupService, GroupService>();
        services.AddTransient<IEnumService, EnumService>();
        services.AddTransient<IAppFileService, AppFileService>();
        services.AddTransient<IEventMemberService, EventMemberService>();
    }
```

Ukázka kódu 11 - Registrace služeb do DI kontejneru [vlastní zpracování]

Způsob vyžádání neboli vložení závislosti do konkrétní třídy je znázorněn na výše uvedené ukázce kódu `EventMemberSerivice`. Konkrétní závislost je vkládána při vytváření nové instance třídy skrze parametry konstruktoru třídy.

3.5.6 Pracování s Entity Framework Core

Pro komunikaci s databází byla v projektu využita open-source knihovna Entity Framework Core (EF Core). Jedná se o moderní objektově relační mapovací (ORM) nástroj, který umožňuje pracovat s relačními databázemi, konkrétně v rámci projektu s MSSQL.

Databáze byla vytvořena prostřednictvím kódu aplikace, takzvaným Code-First přístupem. Tento přístup umožňuje definovat celou strukturu databáze pomocí modelových tříd v aplikaci. Za pomocí speciálních atributů lze nastavit parametry tabulek i jejich sloupců. Lze například definovat konkrétní cizí a primární klíče, indexy a podobně. Ukázka modelové třídy viz ukázka kódu (Ukázka kódu 10).

Code-First přístup společně s EF Core poskytuje mechanismus migrací.

Tvorba migrací se provádí pomocí příkazu „*dotnet ef migrations add InitialCreate*“ v .Net konzoli. Příkaz k vytvoření migrace v případě první migrace vytvoří snapshot (snímek) databáze. Tím je soubor uložený v aplikaci, který obsahuje celkovou strukturu databáze. Následně vytvořené migrace již tvoří vlastní soubory s unikátním identifikátorem.

Tyto migrace obsahují pouze změny provedené mezi jednotlivými stavy databáze. Obsahují metody Down a Up. Metoda Up obsahuje předpis nových změn, které se v databázi mají provést. Oproti tomu metoda Down uchovává předpis k návratu do předchozího stavu.

Spuštění migrace přímo do databázového stroje se provede tímto příkazem „*dotnet ef database update <název migrace včetně unikátního čísla>*“. Příkaz lze spustit i bez specifikace migrace a tím se aplikuje nejnovější existující migrace.

V případě úspěšné migrace se vytváří záznamy o spuštěných migracích do tabulky __EFMigrationsHistory.

Příkaz ke spuštění migrace do databáze se dá využít i obráceným způsobem. Pokud je potřeba vrátit některé změny, lze specifikovat název konkrétní migrace do které je potřeba stav databáze vrátit.[41], [42], [43]

```
public partial class files : Migration
    {
        /// <inheritdoc />
        protected override void Up(MigrationBuilder migrationBuilder)
        {
            migrationBuilder.AddColumn<string>(
                name: "Path",
                table: "AppFiles",
                type: "nvarchar(max)",
                nullable: false,
                defaultValue: "");
        }

        /// <inheritdoc />
        protected override void Down(MigrationBuilder migrationBuilder)
        {
            migrationBuilder.DropColumn(
                name: "Path",
                table: "AppFiles");
        }
    }
```

Ukázka kódu 12- Kód migrace pro úpravu tabulky AppFiles [vlastní zpracování]

3.6 Front-end aplikace

Front-end neboli uživatelské prostředí aplikace je napsáno v ReactJs, v moderní javascriptové knihovně pro tvorbu uživatelského rozhraní. React.js umožňuje vývoj interaktivních a dynamických webových aplikací. Toho dosahuje využitím komponentové architektury a virtuálního DOM (Document Open Model). Výsledkem je aplikace, která efektivně reaguje na datové změny pouze aktualizováním dotčených komponent. Tím se aplikace stává uživatelsky přívětivější.

3.6.1 Založení React.js projektu včetně instalace potřebných modulů

Pro založení a vývoj projektu byla využita lokální open-source serverová platforma Node.js. Tato knihovna umožňuje spouštět javascriptový kód mimo webový prohlížeč.

K založení a instalaci externích knihoven projektu byl využit správce balíčku pro Node.js, npm (Node Package Manager). Pomocí jednoduchého příkazu „*npm init react-app my-app*“ si správce balíčku stáhl veškeré potřebné soubory a knihovny pro úspěšné vytvoření základu projektu React.js.

3.6.2 Seznam využitých externích knihoven

K instalaci externích knihoven do React projektu byl využit příkaz „*npm install <název knihovny>*“. Níže uvedený seznam obsahuje názvy a verze všech, kromě defaultních, využitých knihoven.

- @fortawesome/free-solid-svg-icons verze 6.4.0
- @fortawesome/react-fontawesome verze: 0.2.0
- axios verze: 1.3.5
- bootstrap verze: 5.2.3
- compressorjs verze: 1.2.1
- date-fns verze: 2.30.0
- jwt-decode verze: 3.1.2
- react-bootstrap verze: 2.7.2
- react-datepicker verze: 4.16.0
- react-icons verze: 4.10.1
- slick-carousel verze: 1.8.1
- web-vitals verze: 2.1.4

3.6.3 Struktura React.js projektu

React.js projekt je složen z několika složek a souborů. Níže je popis klíčových struktur.

Ve složce *node_modules* se nacházejí veškeré instalované knihovny, kterou jsou využity pro chod projektu. S touto složkou souvisí konfigurační soubor *package.json*. Obsahem tohoto souboru jsou metadata a seznam knihoven. Zároveň obsahuje definici scriptů pro spuštění projektu.

Složka *public* obsahuje veřejné a sdílené soubory. Nejdůležitějším souborem v této složce je soubor *index.html*. Tento soubor je výchozím HTML souborem webové aplikace.

Následuje složka *src*. V této složce je veškerá naprogramovaná struktura aplikace. Ta je dělena na:

Komponenty (Components) – zde jsou naprogramovány a uloženy často využívané komponenty, jako je rozložení stránek v rámci celého uživatelského rozhraní aplikace, hlavička a patička stránek, které jsou vkládány do rozložení stránky. Následně se zde nachází specifičtější komponenty.

Těmi je například *BasicDateTimePicker.js*.

Tato komponenta byla specificky napsána a nastylována tak, aby se dodržel jednotný styl komponenty pro výběr datumu a času v kterékoli části aplikace.

Další složkou je složka *Containers*. Zde se nachází zbytek vizualizovaných komponent – například `User, Event, File, Image`. Jedná se o složitější struktury skládající se z více částí, které jsou následně vkládány do jednoho celku.

Poslední důležitou složkou je složka *Utils*. Zde se nacházejí nevizuální funkce. Mezi tyto funkce patří funkce pro komunikace s back-end aplikací.

```
import React from "react";
import PropTypes from "prop-types";

const Pagination = ({ totalPageCount, currentPage, onPageChange }) => {
 return (
 totalPageCount !== null && (
  <div className="d -flex justify-content-center mt-3 ">
  <nav aria-label="Page navigation">
   <ul className="pagination">
   {/* Previous page button */}
   <li className={`page-item ${currentPage === 1 ? "disabled" : ""}`}>
    <button
    className="page-link"
    onClick={() => onPageChange(currentPage - 1 )}
    disabled={currentPage === 1 }
    >
    Previous
    </button>
   </li>

   {/* Page numbers */}
   {Array.from({ length: totalPageCount || 0 }, (_, index) => index + 1 ).map((pageNum) =>
(
    <li
    key={pageNum}
    className={`page-item ${pageNum === currentPage ? "active" : ""}`}
    >
    <button
     className="page-link"
     onClick={() => onPageChange(pageNum)}
    >
     {pageNum}
    </button>
    </li>
   ))}

   {/* Next page button */}
   <li
    className={`page-item ${currentPage === totalPageCount ? "disabled" : ""}`}
   >
    <button
    className="page-link"
    onClick={() => onPageChange(currentPage + 1 )}
    disabled={currentPage === totalPageCount}
    >
    Next
    </button>
   </li>
   </ul>
  </nav>
  </div>
 )
 );
};

Pagination.propTypes = {
 totalPageCount: PropTypes.number,
 currentPage: PropTypes.number,
 onPageChange: PropTypes.func.isRequired,
};

export default Pagination;
```

Ukázka kódu 13 - JS ukázka kódu vlastní komponenty stránkování [vlastní zpracování]

3.6.4 Komunikace s back-end aplikací

Pro zajištění komunikace webové aplikace a back-end aplikace byla využita knihovna Axios. Tato javascriptová knihovna umožňuje jednoduché posílání a přijímání základních HTTP požadavků. Podporuje asynchronní operace, umožňuje vkládat a zpracovávat hlavičky požadavků a odpovědí. Zároveň podporuje automatické převody datových formátů, jako jsou JSON, XML, form data a podobně. V níže uvedené ukázce lze vidět konfiguraci axiosu.

```
const axiosWithJwt = axios.create({
 baseURL: BASE_URL,
 headers: {
 "Content-Type": "application/json",
 },
});

axiosWithJwt.interceptors.request.use(
 (config) => {
 const storageData = localStorage.getItem("auth");
 const token = storageData !== null ? JSON.parse(storageData) : null;
 if (token) {
  config.headers.Authorization = `Bearer ${token}`;
 }
 return config;
 },
 (error) => {
 return Promise.reject(error);
 }

);
```

Ukázka kódu 14 - JS konfigurace Axios instance [vlastní zpracování]

3.6.5 React Hook

Hook neboli háček je speciální funkce Reactu, které se „zaháčkuje“ do stavu a funkcí životního cyklu React aplikace. Jedná se o čistě funkcionální programovatelnou komponentu, kterou nelze využít v rámci třídy. Hook slouží k uchování stavu komponent bez nutnosti definovat třídu.

V React knihovně existuje několik výchozích háčků. Mezi základní z nich se řadí useState, useEffect, useRef, useContext a další háčky.[44]

3.6.5.1 Hook useState

UseState je háčkem, který umožňuje funkcionální komponenty uchovávat a aktualizovat jejich lokální stav. Tento hook vrací hodnotu a funkci, pomocí které je možné hodnotu měnit.[44]

3.6.5.2 Hook useEffect

Hook useEffect oproti useState umožňuje komponentám reagovat na změny během životního cyklu komponenty. Především slouží k definování a provádění efektů, které se mají provést po změně některé z komponent nebo při jejím vykreslení.
Tělo tohoto háčku se skládá z funkce, která popisuje požadovanou logiku a takzvané „dependencies“ – závislosti. Závislosti jsou nepovinným parametrem, kterým lze daný useEffect hook spustit v okamžiku změny stavu některé ze závislostí.[44]

3.6.5.3 Hook useRef

UseRef hook umožňuje uchování hodnoty nebo reference na konkrétní DOM prvky mezi překreslovací operací. Využití nachází v situacích, kdy je potřeba uchovat hodnoty, které již nesouvisí s aktuálním stavem komponenty. [44]

V níže přiložené ukázce kódu je zobrazeno využití háčků useEffect pro načtení dat z back-end aplikace. Zároveň je zde ukázán hook useState, který uchovává hodnotu listu získaných dat z API nebo druhý useState hook sloužící k uchování hodnoty textu komentáře. Zároveň je zde zobrazena struktura samotné komponenty. Jedná se o funkci, která může přijímat předem specifikované parametry a jejím výstupem je vizuální zobrazení dat. Výsledné zobrazení komponenty je popsáno v return bloku.

```
import React, { useEffect, useState } from "react";
import {Container, Form, Button,Card, Col,Row,
} from "react-bootstrap";
import "bootstrap/dist/css/bootstrap.min.css";
import CommentCard from "./CommentCard";
import       {       createComment,       deleteComment,       getComments       }       from
"../../../Utils/Routes/CommentApi";
import { useParams } from "react-router-dom";

const CommentPage = () => {
 const [commentText, setCommentText] = useState("");
 const [commentList, setCommentList] = useState([]);
 const { eventId } = useParams();
 const commentModel = {
 eventId: eventId,
 title: null,
 text: commentText,
 };

 const handleSubmit = async (event) => {
 event.preventDefault();
 if (commentText !== "") {
  var response = await createComment(commentModel)
  setCommentList([...commentList, response]);
  setCommentText("");
 }
 };

 const handleDelete = async (comment) => {
 try {
  await deleteComment(comment.id);
  setCommentList((prevComments) =>
  prevComments.filter((prevComment) => prevComment.id !== comment.id)
  );
 } catch (error) {
  console.error("Error deleting comment:", error);
 }
 };

 useEffect(() => {
 const fetchData = async () => {
  try{
  const data = await getComments(eventId);
  setCommentList(data);
  } catch (ex){
  console.error("Error getting comments");
  }
 };

 fetchData();
 }, [])
```

```
 return (
 <Container>
  <Row className="justify-content-md-center">
  <Col md={7 }>
   <Card className="mt-1 p -4 rounded shadow-sm text-center mb-5 ">
   <Card.Header>
    <h2>Comments</h2>
   </Card.Header>
   <Card.Body>
    {commentList.length > 0 ? (
    commentList.map((item) => {
     return (
     <Col key={item.id} xs={12}>
      <CommentCard comment={item} handleDelete={handleDelete} />
     </Col>
     );
    })
    ) : (
    <Col xs={12}>
     <p className="text-center fs-4 text-muted">
     No comments yet...
     </p >
    </Col>
    )}
   </Card.Body>
   <Card.Footer>
    <Form onSubmit={handleSubmit}>
    <Form.Group controlId="comment" className="mb-3 ">
     <Form.Control
     as="textarea"
     rows={3 }
     value={commentText}
     onChange={(e ) => setCommentText(e .target.value)}
     placeholder="Write your comment here..."
     />
    </Form.Group>
    <Button type="submit" variant="success">
     Submit
    </Button>
    </Form>
   </Card.Footer>
   </Card>
  </Col>
  </Row>
 </Container>
 );
};

export default CommentPage;
```

Ukázka kódu 15 - JS vizuální komponenta Reactu s využitím háčků [vlastní zpracování]

3.7 Ukázkový test Cypress.io

Součástí front-end aplikace byl implementován testovací modul Cypress s cílem ověření funkčnosti uživatelského rozhraní aplikace. Byl vytvořen jednoduchý scénář pro ověření validace registračního formuláře. První scénář ověřuje funkčnost formuláře v případě, kdy byl formulář vyplněn validními daty. Druhý scénář testuje

stejný formulář, ovšem jeho cílem je ověřit, zda se zobrazí validační hlášky v případě, že je formulář vyplněn nevalidně.

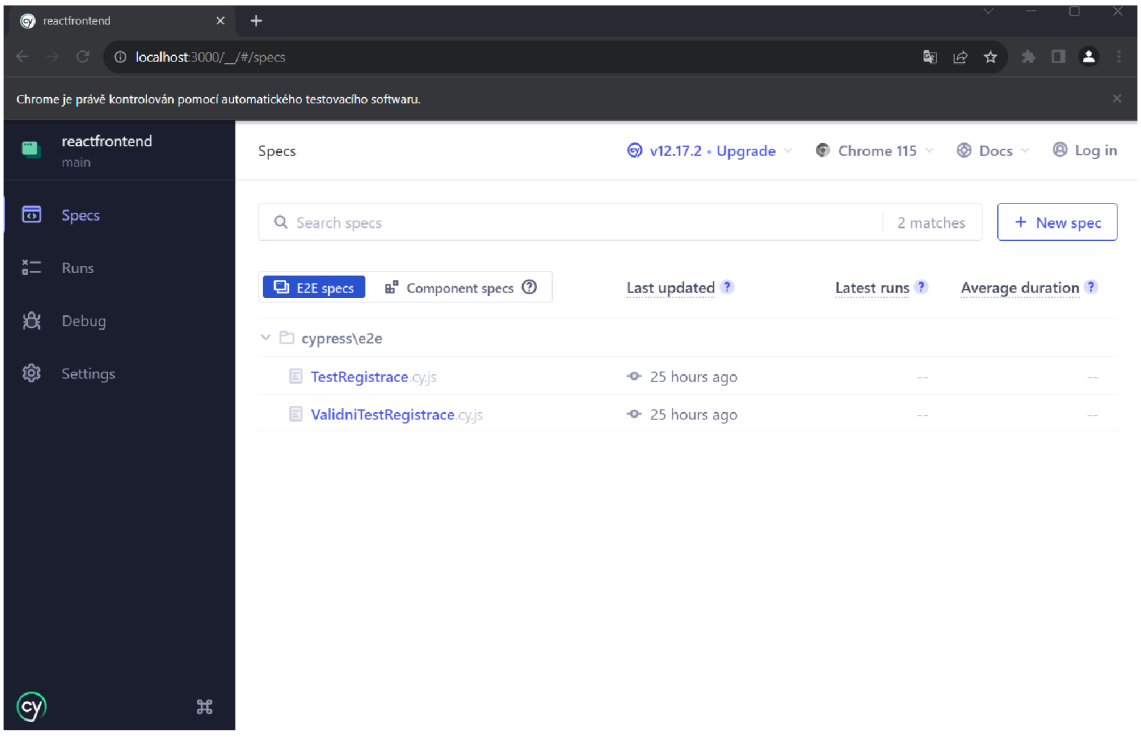
3.7.1 Instalace Cypress

Instalace Cypress probíhala stejně jako instalace jakéhokoli jiného externího modulu aplikace. K úspěšnému nainstalování byl spuštěn v konzoli příkaz „*npm install cypress --save-dev*“.

Po úspěšné instalaci se v projektu vytvořila složka *cypress*, která obsahuje samostatné e2e testové scénáře. Dále se zde uchovává i videonahrávka samotného testu ve formátu MP4. Dále se zde nachází složka *fixtures* v níž jsou obsaženy soubory, které umožňují takzvané mockování dat.

3.7.2 Rozhraní Cypress

Cypress umožňuje nejen čistě kódovou tvorbu testů, ale i tvorbu testových scénářů pomocí uživatelského rozhraní. Zapnutí uživatelského rozhraní je umožněno příkazem „*npx cypress open*“. Po zadání příkazu do konzole se otevře webové prostředí, viz obrázek níže.



Obrázek 9 - Webové prostředí frameworku Cypress [vlastní tvorba]

3.7.3 Tvorba testu

Testový scénář je vytvořen za pomocí přidání nového „*specs*“. Tím dochází k vytvoření javascriptového souboru v projektu. Tento soubor následně obsahuje popis testu. Test je možné vytvořit i prostým vytvořením souboru v příslušné složce projektu, čímž je možné se vyhnout uživatelskému rozhraní.

Soubor se skládá z funkcí. Primární funkcí v testu je funkce `describe`. Tato funkce vytváří skupinu testů s konkrétním popisem. Funkce `describe` slouží primárně pro organizaci a členění testů.

V těle funkce `describe` se následně nachází funkce `it`. Ta definuje samostatné testy, součástí funkce je i popisek testu.

Následně se v těle funkce `it` nachází samotné příkazy knihovny Cypress. Těmi jsou například:

- `cy.visit(url)` – navigace na konkrétní url
- `cy.get(selektor)` – získání konkrétního elementu na stránce
- `cy.contains(text)` – vyhledání elementu obsahující konkrétní text
- `cy.click()` - provedení akce kliknutí na konkrétním elementu
- `cy.type(text)` – simulace psaní textu
- `cy.should()` – funkce ověřující obsah konkrétního elementu
- `cy.wait(time)` – funkce umožňující vyčkání v rámci konkrétního intervalu

Tyto příkazy lze řetězit, příkladem může být následující příkaz.

```
cy.get(".text-danger").should("exist").and("have.text", "Please enter a valid e-mail address.");
```

Ukázka kódu 16 – JS řetězený příkaz testovací knihovny Cypress [vlastní tvorba]

Výše uvedený příkaz si nejprve získá přístup ke komponentě obsahující CSS třídu `.text-danger`. Následně je tato komponenta ověřována funkcí `should`, která v parametru vyžaduje existenci této komponenty. Poslední řetězenou funkcí je funkce `and`, která ověřuje, že tato získaná a existující komponenta obsahuje text „*Please enter a valid e-mail address.*“ Pokud by některý z těchto parametrů neodpovídal popisu v testu, bude test vyhodnocen jako neúspěšný.

Pro zjednodušení práce tvorby testu poskytuje webové rozhraní Cypress nástroje pro výběr komponenty přímo v uživatelském prostředí testované aplikace. Výstupem je Cypress příkaz, kterým je možno identifikovat patřičný element. Ten lze následovně kopírovat a využít v testu.

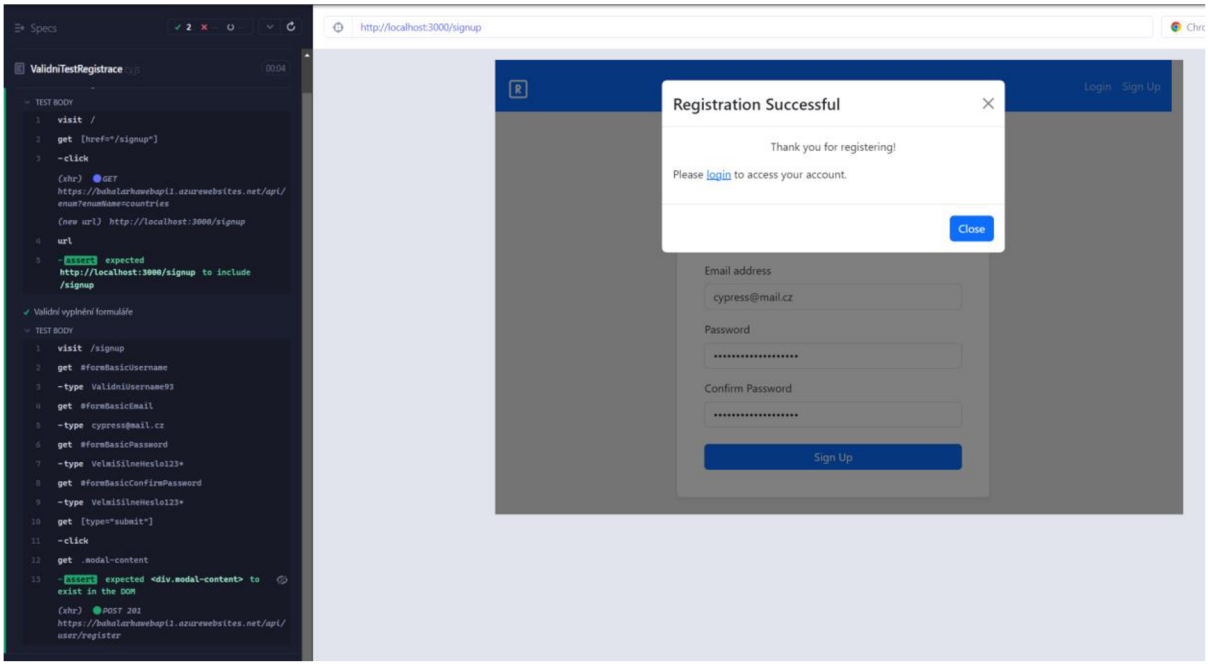
```
describe("Validni Test registrace", () => {
 it("Přesunutí na login stránku", () => {
  cy.visit("/");
  cy.get(' [href="/signup"]' ).click();
  cy.url().should("contain", "/signup");
 });

 it("Validní vyplnění formuláře", () => {
  cy.visit("/signup");

  //UserName
  cy.get("#formBasicUsername").type(
  `ValidniUsername${Math.floor(Math.random() * 100) + 1 }`
  );

  //Mail
  cy.get("#formBasicEmail").type("cypress@mail.cz");

  //PWD 1
  cy.get("#formBasicPassword").type("VelmiSilneHeslo123*");

  //PWD 2
  cy.get("#formBasicConfirmPassword").type("VelmiSilneHeslo123*");

  //submit
  cy.get(' [type="submit"]' ).click();

  //Úspěšná registrace
  cy.get(".modal-content").should("exist");
 });
 });
```

Ukázka kódu 17 –JS - Cypress - jednoduchý test registračního formuláře [vlastní tvorba]

3.7.4 Spuštění a výsledky testů

Test se spouští v průběhu tvorby automaticky při každém vložení souboru, nebo ho lze spustit ručně pomoci tlačítka ve webovém rozhraní.

Cypress zároveň umožňuje spuštění testů pomocí příkazu „*npx cypress run*“. Díky této možnosti spuštění lze testování plně automatizovat tím, že se příkaz zakomponuje do pipeline.



Obrázek 10 - Cypress výsledek úspěšného testu [vlastní zpracování]

3.8 Srovnání práce s tetovacími softwary Selenium a Cypress

Za účelem porovnání testovacích softwarů byl vytvořen stejný test za pomocí konkurenčního software Selenium.

3.8.1 Instalace Selenium

Instalace Selenia oproti Cypress probíhala ve více krocích.

Nejprve bylo potřeba si nainstalovat samotné rozšíření do prohlížeče chrome. Konkrétně se jednalo o oficiálně vydané rozšíření s názvem Selenium IDE. Instalace probíhala skrze Internetový obchod chrome. Tento plugin není nutností, ovšem s jeho využitím, lze zjednodušit a zrychlit tvorbu testů.

V následujícím kroku instalace bylo zapotřebí nainstalovat ovladač prohlížeče do ReactJs projektu. To stejně jako u Cypress probíhalo skrze Node Package Manager a to pomocí tohoto příkazu „*npm install selenium-webdriver*“.

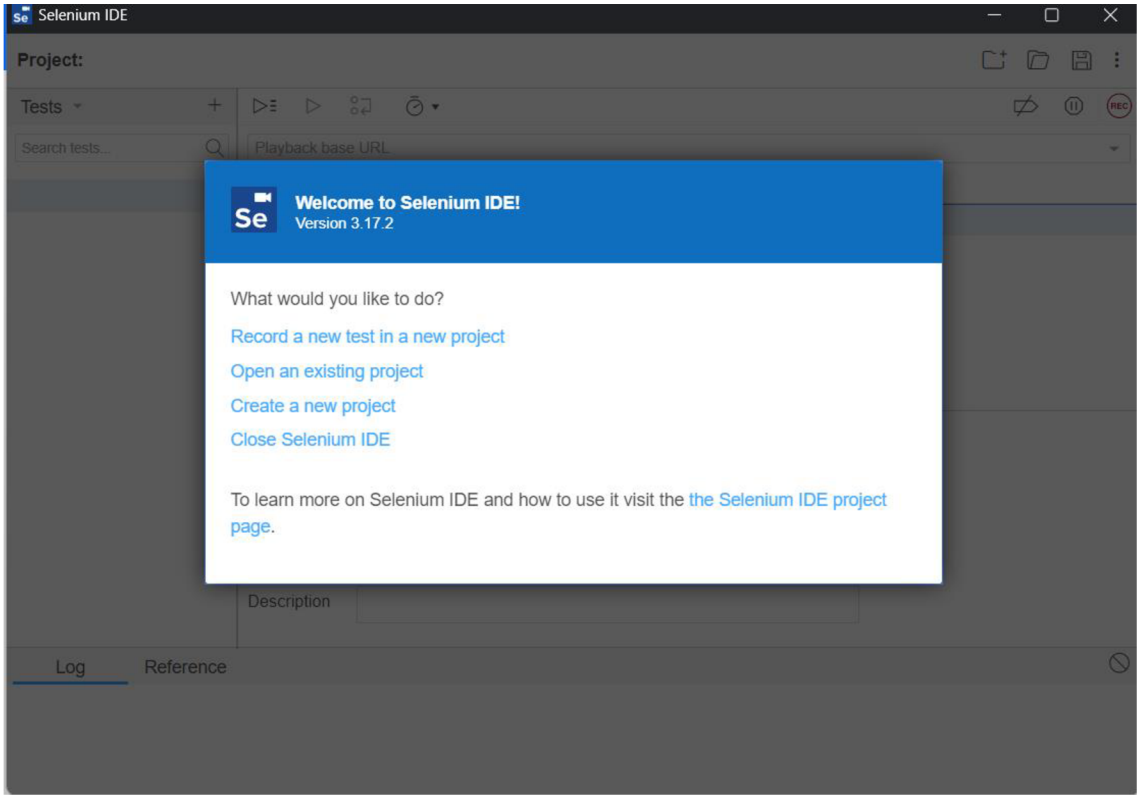
Aby bylo možné bezproblémově spouštět testy, které byly vygenerovány pomocí Selenium IDE, bylo zapotřebí nainstalovat dodatečný javascriptový balíček *„npm install mocha“*.

3.8.2 Rozhraní Selenium IDE

Selenium IDE prostředí je na rozdíl od Cypress integrováno do webového prohlížeče v podobě pluginu. Rozšíření je možné instalovat do široké škály známých prohlížečů jako jsou prohlížeče založené na plaformě Chromium a nebo například Mozilla Firefox a další.

Samotné rozhraní se tedy na rozdíl od Cypress nespouští příkazem, ale je spuštěno společně s celým webovým prohlížečem.

Prostředí samotného IDE je jednoduché a intuitivní. Po spuštění vypadá viz následující obrázek.



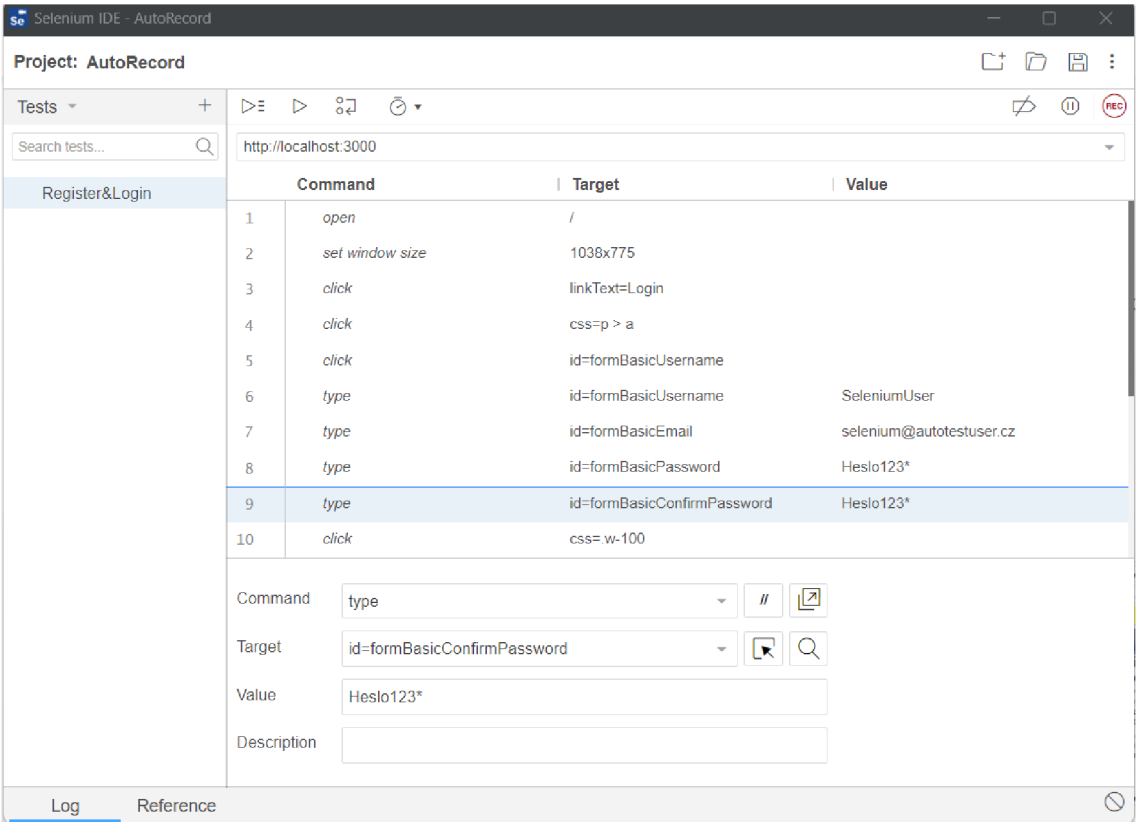
Obrázek 11 - Selenium IDE [vlastní zpracování]

3.8.3 Tvorba testu pomocí Selenium IDE

V prostředí Selenium IDE pluginu, lze jednoduše vytvořit testový projekt. Následně prostředí umožní spustit nahrávání. Toto nahrávání otevře novou kartu prohlížeče s cílovou URL testovaného webu.

Po spuštění nahrávání zaznamenává IDE každé kliknutí a každou uživatelskou interakci s testovanou stránkou. Díky tomu je po dokončení nehrávání vytvořen testovací scénář.

Tvorba testovacího scénáře pomocí nahrávání je oproti Cypress mnohem intuitivnější a rychlejší. Cypress sice umožňuje tvorbu testu za pomocí webového rozhraní, ovšem webové rozhraní Cypress neumožňuje nahrávání jako Selenium. Slouží spíše k identifikování jednotlivých komponent, aby je bylo možné kopírovat do ručně psaného scénáře v kódu.



Obrázek 12 - Nahrávka testu v Selenium IDE [vlastní zpracování]

Na obrázku výše je dokončená nahrávka scénáře testu. V levé části je záložka, kde se zobrazují jednotlivé testy v projektu. V pravé části je již tabulka s kroky konkrétního nahraného testu. Je zde zaznamenán každý krok včetně identifikace elementu, na kterém došlo k interakci a popřípadě i hodnoty, která byla zadána.

Takto vytvořený test je možné ihned spustit pomocí tlačítek v horní části nad tabulkou. Je zde možnost i nastavení rychlosti průběhu testu.

Pro účel spouštění testu automatizovaně i bez využití uživatelského prostředí bylo zapotřebí tento test uložit jako spustitelný kód. K tomu byl využit export testu ve formě kódu.

Defaultně Selenium IDE nabízí export do C# NUnit, C# xUnit, Java jUnit, Javascript Mocha, Python pytest, Ruby RSpec jazyků a knihoven. Za účelem testování front-end části byla proto vybrána možnost Javascript za pomocí knihovny Mocha.

```
// Generated by Selenium IDE
const { Builder, By, Key, until } = require('selenium-webdriver')
const assert = require('assert')

describe('Register&Login', function() {
  this.timeout(30000)
  let driver
  let vars
  let username
  let password
  beforeEach(async function() {
    driver = await new Builder().forBrowser('chrome').build()
    username = `SeleniumAutomatedTest${Math.floor(Math.random() * 100) + 1}`
    password = "Heslo123*"
    vars = {}
  })
  afterEach(async function() {
    await driver.quit();
  })
  it('Register&Login', async function() {
    // Test name: Register&Login
    // Step # | name | target | value
    // 1 | open | / |
    await driver.get("http://localhost:3000/")
    // 2 | setWindowSize | 1038x775 |
    await driver.manage().window().setRect({ width: 1038, height: 775 })
    // 3 | click | linkText=Login |
    await driver.findElement(By.linkText("Login")).click()
    // 4 | click | css=p > a |
    await driver.findElement(By.css("p > a")).click()
    // 5 | click | id=formBasicUsername |
    await driver.findElement(By.id("formBasicUsername")).click()
    // 6 | type | id=formBasicUsername | SeleniumUser
    await driver.findElement(By.id("formBasicUsername")).sendKeys(username)
    // 7 | type | id=formBasicEmail | selenium@autotestuser.cz
    await driver.findElement(By.id("formBasicEmail"))
          .sendKeys(`selenium${Math.floor(Math.random() * 100) + 1}@autotestuser.cz`)
    // 8 | type | id=formBasicPassword | Heslo123*
    await driver.findElement(By.id("formBasicPassword")).sendKeys(password)
    // 9 | type | id=formBasicConfirmPassword | Heslo123*
    await driver.findElement(By.id("formBasicConfirmPassword")).sendKeys(password)
    // 10 | click | css=.w-100 |
    await driver.findElement(By.css(".w-100")).click()
    // 11 | click | linkText=login |
    await driver.sleep(3000)
    await driver.findElement(By.linkText("login")).click()
    // 12 | click | id=formBasicUsername |
    await driver.findElement(By.id("formBasicUsername")).click()
    // 13 | type | id=formBasicUsername | SeleniumUser
    await driver.findElement(By.id("formBasicUsername")).sendKeys(username)
    // 14 | click | id=formBasicPassword |
    await driver.findElement(By.id("formBasicPassword")).click()
    // 15 | type | id=formBasicPassword | Heslo123*
    await driver.findElement(By.id("formBasicPassword")).sendKeys(password)
```

```
    // 16 | click | css=.w-100 |
    await driver.findElement(By.css(".w-100")).click()
    // 17 | click | linkText=Profile |
    await driver.findElement(By.linkText("Profile")).click()
  })
})
```

Ukázka kódu 18 - Selenium generovaný kód testu [vlastní zpracování]

V ukázce výše vygenerovaného spustitelného kódu testu lze vidět jistou podobnost s kódem testu knihovny Cypress (Ukázka kódu 17 –JS - Cypress - jednoduchý test registračního formuláře [vlastní tvorba]). To je dáno tím, že obě tyto knihovny vyhledávají a pracují s komponentami velice podobně.

Pro finální spuštění tohoto testu bez uživatelského rozhraní je na rozdíl od Cypress u Selenia potřeba knihovna Mocha. Následné spuštění probíhá pomocí příkazu „*npx mocha <cesta k souboru s testem>*“. Výstupem je výpis názvů testů do konzole společně s označením, zda tento test byl úspěšný nebo neúspěšný.

3.9 Pipeline pro automatizovaný deployment

Způsob nasazení aplikace do prostředí Azure AppService probíhal nejprve ručním způsobem. Pro úspěšné nasazení do tohoto prostředí bylo potřeba do IDE doinstalovat patřičné pluginy, pomocí kterých bylo následně možné spojit se s cloudovou službou Azure. Aby byl tento manuální postup nahrazen, byla do projektů přidána pipeline neboli v českém překladu potrubí.

Kvůli kompatibilitě byly zdrojové kódy aplikací přesunuty ze služby GitLab do Azure DevOps. Tím se výrazně zjednodušilo napojení pipeline na cloudové prostředí, jelikož se jedná o služby jedné společnosti.

Tento krok byl v rámci mého projektu nezbytný. Pipeline je možné nastavit i z konkurenční služby pomocí vytvoření patřičných přihlašovacích údajů pro pipeline. Ovšem tato možnost v rámci studentského předplatného Azure je omezena.

K vytvoření pipeline pomocí Azure DevOps byla relativně jednoduchá záležitost. Uživatelské prostředí totiž zpřístupňuje několik základních šablon pro aplikace v různých technologiích. Mezi nimi je jak ASP NET Core, tak i Node.js s ReactJs.

Šablona následně vytvořila soubor s popisem jednotlivých akcí, které se mají v DevOps automaticky spustit (viz ukázka kódu pipeline níže).

```yaml
# ASP.NET Core (.NET Framework)
# Build and test ASP.NET Core projects targeting the full .NET Framework.
# Add steps that publish symbols, save build artifacts, and more:
# https://docs.microsoft.com/azure/devops/pipelines/languages/dotnet-core

trigger:
- master

pool:
 vmImage: 'windows-latest'

variables:
 solution: ' **/*.sln'
 buildPlatform: 'Any CPU'
 buildConfiguration: 'Release'

steps:
- task: NuGetToolInstaller@1

- task: NuGetCommand@2
 inputs:
  restoreSolution: ' $(solution)'

- task: VSBuild@1
 inputs:
  solution: ' $(solution)'
  msbuildArgs: ' /p :DeployOnBuild=true /p :WebPublishMethod=Package /p :PackageAsSingleFi
le=true /p :SkipInvalidConfigurations=true /p :DesktopBuildPackageLocation="$(build.artifa
ctStagingDirectory)\WebApp.zip" /p :DeployIisAppPath="Default Web Site"'
  platform: ' $(buildPlatform)'
  configuration: ' $(buildConfiguration)'

- task: VSTest@2
 inputs:
  platform: ' $(buildPlatform)'
  configuration: ' $(buildConfiguration)'
```

Ukázka kódu 19 – YAML, nastavení pipeline pro sestavení .NET aplikace [vlastní zpracování]

3.9.1 Popis kódu pipeline

Výše uvedený kód pipeline je v programovacím jazyce YAML (Yet Another Markup Language). Kód se skládá z několika částí:

- `Trigger` – Tento příkaz specifikuje, při jaké akci se má pipeline automaticky spustit. V případě ukázky se pipeline spustí vždy, když dojde ke změně ve vývojové Git větvi master.
- `Pool` – Určuje, ve kterém prostředí dojde ke spuštění pipeline. V kódu `pipeline` lze vidět nastavení spuštění v nejnovější verzi Windows.
- `Variables` – Definice využitých proměnných.
- `Steps` – Tento příkaz seskupuje jednotlivé kroky provedené v rámci pipeline.

- `Task` – Task je úlohou kroku, která provádí konkrétní definovanou akci, například sestavení kódu (`VSBuild@1`), instalace balíčků (`NuGetCommand@2`) nebo spuštění testů (`VSTest@2`).

Výše ukázaná pipeline úspěšně kód sestavila a otestovala. Pro nasazení bylo ovšem potřeba vytvořit takzvanou „Release“ pipeline. Ta se v prostředí DevOps váže k předchozí pipeline. Po jejím úspěšném dokončení provede dle své specifikace nasazení sestaveného kódu aplikace do prostředí Azure AppService.
Kód této pipeline vypadá následovně:

```
steps:
- task: AzureRmWebAppDeployment@4
  displayName: 'Deploy Azure App Service'
  inputs:
    azureSubscription: ' $(Parameters.ConnectedServiceName)'
    appType: ' $(Parameters.WebAppKind)'
    WebAppName: ' $(Parameters.WebAppName)'
    packageForLinux: ' $(System.DefaultWorkingDirectory)/_Bakalářka REST API'
```

Ukázka kódu 20 - YAML, krok pipeline pro nasazení do AppService [vlastní zpracování]

Takto nastavená pipeline provede nasazení a tím je proces u konce. Aplikace je spuštěná a je možné ji plně využívat.

4 Shrnutí výsledků

Vytvořená aplikace splňuje požadavky stanovené v rámci zadání práce. Aplikace je rozdělena do 3 logických vrstev.

Těmi je datová vrstva, která je implementována pomocí MSSQL. Zároveň databázový model je navržen tak, aby umožnil budoucí rozšíření.

Další vrstvou je back-end, který díky využití Entity Framework Core plně komunikuje a manipuluje s databází. Díky své architektuře odpovídá návrhu struktur moderních, snadno udržitelných a rozšířitelných aplikací. Implementuje veškerou aplikační logiku a přijímá, zpracovává data dle požadavků jednotlivých endpointů.

Komunikace dle požadavků probíhá mezi back-end a front-end aplikací pomocí formy HTTP požadavků. Tyto požadavky jsou rozšířeny o autorizační JWT token v hlavičce, který garantuje ověření přihlášeného uživatele. Veškerá datová komunikace přenáší data ve formě JSON dat, které jsou následně přijatou stranou zpracovány. Komunikovat s back-end aplikace lze i pomocí HTTP požadavků zasílaných ze samotného prohlížeče nebo ze softwaru k tomu určenému – například Postman.

Třetí vrstvou je front-end aplikace, která poskytuje uživateli interaktivní a intuitivní webové rozhraní pro ovládání aplikace. Garantuje možnost využití modelového příkladu aplikace, čímž je aplikace pro tvorbu a správu událostí.

Poskytuje možnost registrace, přihlášení uživatele. Dále umožňuje správu uživatelského profilu, včetně nahrání v souborů ve formě obrázku. Dále lze po přihlášení uživatele vytvořit a spravovat samostatnou událost nebo lze prohlížet, filtrovat a vyhledávat existující události i v případě, že uživatel není registrován.

Přihlášený uživatel může následně vytvořit komentář pod událostí anebo účastnit se této události.

Front-end dodržuje tvoření struktury stránky za pomocí vlastních komponent skládajících se do sebe a zároveň využívá funkce knihovny ReactJs, jako jsou háčky.

Vytvořený testový scénář lze automaticky pustit z příkazové řádky nebo ve webovém rozhraní. A to jak v Cypress tak i v Selenium. Každý software má svůj testovací script, scénář je ovšem podobný.

V případě Cypress existují dva testovací scénáře. Prvním je validace funkčnosti registračního formuláře pomocí validních dat. Druhým scénářem se testuje, zda stejný registrační formulář obsahuje validační kontroly a úspěšně je zobrazí v případě, že se ve formuláři vyskytují nevalidní datové vstupy.

V případě Selenia je testován stejný registrační formulář v případě vyplnění validních dat a následné umožnění přihlášení za pomocí stejných přihlašovacích údajů jako v registračním formuláři.

Posledním požadavkem je funkční pipeline, která je napsána v jazyce YAML a vytváří automatický proces pro sestavení kódu aplikace a nahrání tohoto výsledku do cloudového prostředí Azure.

5 Závěry a doporučení

V průběhu psaní zdrojového kódu aplikace jsem dospěl k závěru, že způsob, jakým jsem psal kód aplikace, nebyl zrovna nejlepší. I přestože jsem zdrojový kód aplikace dělil do odpovídající struktury a snažil jsem se dodržovat zmíněné principy, dal by se kód aplikace více zpřehlednit.

Toho by bylo možno dosáhnout logičtějším pojmenováním metod a nahrazením opakovaných metod za pomocí dědičnosti a implementace generických tříd, které by v sobě defaultně předpisovaly jednoduché operace typu Create, Update, Delete, Get. To by umožnilo výraznou redukci napsaného zdrojového kódu a zároveň poskytlo možnost snadné úpravy konkrétních funkcí.

To samé bych mohl říct i o front-end aplikaci. Vzhledem k tomu, že se jednalo o mojí první aplikaci tvořenou v ReactJs frameworku, nebyl jsem si zezačátku plně vědom veškerých možných funkcí, které knihovna poskytuje. Zároveň prvotní rozložení struktury nebylo ideální a kód aplikace jsem několikrát přepisoval.

Největším problémem v rámci tvoření front-end aplikace bylo rozhodnutí, zda určitá část má mít vlastní komponentu, nebo má být pevnou součástí jiné. V průběhu tvorby a přidávání dalších funkcí jsem ovšem narazil na momenty, kdy bylo potřeba některé části vyjmout a udělat z nich samotnou komponentu.

Hlavním kamenem úrazu strukturování ReactJs projektu byl v tom, že jsem si našel spoustu různých projektů a snažil jsem se v nich vyhledat nejlepší možnou architekturu. Prozkoumal jsem spoustu projektů zveřejněných především na GitHub platformě, ovšem každý projekt byl složen jiným způsobem a každý tutoriál zhlédnutý na YouTube doporučoval něco jiného. To ve výsledku způsobilo ještě větší zmatenost ohledně struktury projektu. Doporučil bych proto vybrat si jeden vzorový projekt a dle něj si odvodit vlastní strukturu.

Poslední doporučení se týká zvolené platformy pro verzování kódu. Zvolil jsem platformu GitLab, která poskytuje cloudový Git pro správu a verzování zdrojových kódu, a to společně s DevOps. Vzhledem k tomu, že cílové prostředí pro vydání aplikace bylo u konkurenční společnosti, tak toto nebylo zrovna nejlepší rozhodnutí. Ovšem na to jsem přišel až ve finální verzi, kdy mělo dojít k nasazení projektu přímo z Gitu pomocí pipeline.

Prostředí GitLab sice umožňuje tvorbu pipeline stejně jako konkurenční GitHub nebo Azure DevOps, bohužel vzhledem k omezeným funkcím ve studentském předplatném Azure jsem nebyl schopen vygenerovat přihlašovací údaje pro pipeline spuštěnou z GitLab DevOps.

Řešením proto bylo přesunout zdrojové kódy do Gitu v Azure DevOps nebo do GitHub. Jelikož se jedná o platformy vlastněné stejnou společností, Microsoft, byla implementace pipline pro nasazení do Azure AppServices mnohem jednoduší. Doporučil bych proto na začátku každého projektu dobře zvážit možnosti jednotlivých platforem a jejich kompatibilitu s prostředím, kde bude finální aplikace spuštěna.

Pokud bych měl tedy tuto práci zhodnotit, rád bych zmínil, že vypracování této práce mi přineslo lepší odhad ohledně času potřebného pro přípravu a vývoj software. Také jsem se díky této práci uvědomil, jak lépe a jednodušeji psát software aplikace. Zároveň jsem byl konfrontován s novou technologií ReactJs, která mi otevřela nadhled a ukázala možnosti javascriptových front-endových knihoven.

Dále bych zmínil, způsob automatizovaného testování. Díky této práci jsem se dozvěděl, jak je vlastně relativně jednoduché nějaký takový test vytvořit a využívat. Vidím v tom velký přínos pro budoucí zaručení kvality software.

Posledním osobním přínosem této práce pro mě byla tvorba pipeline. Tato technologie se mi dříve zdála být složitá. Ovšem po podrobném prozkoumáním a konfrontaci s jazykem YAML jsem byl překvapen, jak moc je tato technologie přínosná a zároveň relativně jednoduchá pro nastavení.

6 Seznam použité literatury

[1] „What is .NET? An open-source developer platform.", *Microsoft*. https://dotnet.microsoft.com/en-us/learn/dotnet/what-is-dotnet (viděno 22. leden 2023).

[2] „ECMA-334", *Ecma International*. https://www.ecma-international.org/publications-and-standards/standards/ecma-334/ (viděno 22. leden 2023).

[3] B. Smith, „Object-Oriented Programming", in *AdvancED ActionScript 3.0: Design Patterns*, B. Smith, Ed., Berkeley, CA: Apress, 2011, s. 1–25. doi: 10.1007/978-1-4302-3615-3_1.

[4] BillWagner, „Object-Oriented Programming (C#)". https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/tutorials/oop (viděno 22. leden 2023).

[5] B. P. Pokkunuri, „Object Oriented Programming", *ACM SIGPLAN Not.*, roč. 24, č. 11, s. 96–101, lis. 1989, doi: 10.1145/71605.71612.

[6] BillWagner, „Polymorphism". https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/object-oriented/polymorphism (viděno 23. leden 2023).

[7] ajcvickers, „Overview of Entity Framework Core - EF Core". https://learn.microsoft.com/en-us/ef/core/ (viděno 23. leden 2023).

[8] A. Fedosejev, *React.js Essentials*. Packt Publishing Ltd, 2015.

[9] G. Aroraa a J. Chilberto, *Hands-On Design Patterns with C# and .NET Core: Write clean and maintainable code by using reusable solutions to common software design problems*. Packt Publishing Ltd, 2019.

[10] T. Sharma, G. Samarthyam, a G. Suryanarayana, „Applying Design Principles in Practice", in *Proceedings of the 8th India Software Engineering Conference*, in ISEC '15. New York, NY, USA: Association for Computing Machinery, úno. 2015, s. 200–201. doi: 10.1145/2723742.2723764.

[11] D. Thomas a A. Hunt, *Pragmatic Programmer, The: Your journey to mastery, 20th Anniversary Edition*, 2nd edition. Boston: Addison-Wesley Professional, 2019.

[12] I. Cabezas, R. Segovia, P. Caratozzolo, a E. Webb, „Using Software Engineering Design Principles as Tools for Freshman Students Learning", in *2020 IEEE Frontiers in Education Conference (FIE)*, říj. 2020, s. 1–5. doi: 10.1109/FIE44824.2020.9274177.

[13] G. K. Arora, *SOLID Principles Succincitly*, 31.10.2016., roč. 2016. 2501 Aerial Center Parkway Suite 200 Morrisville, NC 27560 USA: Syncfusion Inc., 2016.

[Online]. Dostupné z: https://www.syncfusion.com/succinctly-free-ebooks/solidprinciplessuccinctly

[14] *Principles of Package Design*. Viděno: 25. leden 2023. [Online]. Dostupné z: https://link.springer.com/book/10.1007/978-1-4842-4119-6

[15] „Understanding Interfaces and Dependency Inversion", *GoForGoldman*, 12. duben 2021. https://goforgoldman.com/posts/interfaces/ (viděno 26. leden 2023).

[16] „What is Software Testing? Definition, Types and Importance", *WhatIs.com*. https://www.techtarget.com/whatis/definition/software-testing (viděno 19. únor 2023).

[17] G. J. Myers, C. Sandler, a T. Badgett, *The Art of Software Testing*. John Wiley & Sons, 2011.

[18] P. M. Jacob a M. Prasanna, „A Comparative analysis on Black box testing strategies", in *2016 International Conference on Information Science (ICIS)*, srp. 2016, s. 1–6. doi: 10.1109/INFOSCI.2016.7845290.

[19] „What is Software Testing and How Does it Work? | IBM". https://www.ibm.com/topics/software-testing (viděno 19. únor 2023).

[20] „Software Testing Glossary: A-Z Guide To Testing Terminology<". https://www.lambdatest.com/learning-hub/glossary (viděno 26. únor 2023).

[21] M. A. Khan a Mohd. Sadiq, „Analysis of black box software testing techniques: A case study", in *The 2011 International Conference and Workshop on Current Trends in Information Technology (CTIT 11)*, říj. 2011, s. 1–5. doi: 10.1109/CTIT.2011.6107931.

[22] M. Heusser, „What is debugging?", *Techtarget Software Quality*, listopad 2022. https://www.techtarget.com/searchsoftwarequality/definition/debugging (viděno 27. únor 2023).

[23] „What is Unit Testing? Definition from WhatIs.com", *Software Quality*, srpen 2019. https://www.techtarget.com/searchsoftwarequality/definition/unit-testing (viděno 28. únor 2023).

[24] R. Awati, „What is Integration Testing (I&T)?", *Software Quality*, leden 2022. https://www.techtarget.com/searchsoftwarequality/definition/integration-testing (viděno 28. únor 2023).

[25] M. B. Dwyer a A. Lopes, Ed., *Fundamental Approaches to Software Engineering: 10th International Conference, FASE 2007, Held as Part of the Joint European Conferences, on Theory and Practice of Software, ETAPS 2007, Braga, Portugal, March 24 - April 1, 2007. Proceedings*, roč. 4422. in Lecture Notes in Computer Science, vol. 4422. Berlin, Heidelberg: Springer, 2007. doi: 10.1007/978-3-540-71289-3.

[26] R. Black, „What is system testing? | Definition from TechTarget", *Software Quality*, prosinec 2018. https://www.techtarget.com/searchsoftwarequality/definition/system-testing (viděno 1. březen 2023).

[27] A. S. Gillis, „What is Performance Testing?", *Software Quality*, prosinec 2022. https://www.techtarget.com/searchsoftwarequality/definition/performance-testing (viděno 1. březen 2023).

[28] A. S. Gillis, „What is acceptance testing? Definition from SearchSoftwareQuality", *Software Quality*, říjen 2019. https://www.techtarget.com/searchsoftwarequality/definition/acceptance-test (viděno 2. březen 2023).

[29] R. Goel, „User Acceptance Testing(UAT)", *Medium*, 29. červen 2022. https://blog.zipboard.co/user-acceptance-testing-uat-beta-testing-end-user-testing-or-application-testing-6cb094f8afd5 (viděno 2. březen 2023).

[30] „Selenium Overview", *Selenium*, 4. září 2022. https://www.selenium.dev/documentation/overview/ (viděno 8. březen 2023).

[31] K. Rungta, „What is Selenium? Introduction to Selenium Automation Testing", 2. leden 2020. https://www.guru99.com/introduction-to-selenium.html (viděno 8. březen 2023).

[32] „Selenium components", *Selenium*, 4. září 2022. https://www.selenium.dev/documentation/overview/components/ (viděno 8. březen 2023).

[33] „WebDriver", *Selenium*. https://www.selenium.dev/documentation/webdriver/ (viděno 8. březen 2023).

[34] „Grid", *Selenium*. https://www.selenium.dev/documentation/grid/ (viděno 8. březen 2023).

[35] M. Yagudaev, „How to Test Your Frontend with the Cypress.io Framework", *Medium*, 1. únor 2019. https://medium.com/free-code-camp/how-to-test-your-frontend-with-the-cypress-io-framework-f048070f4330 (viděno 8. březen 2023).

[36] „Why Cypress? | Cypress Documentation". https://docs.cypress.io/guides/overview/why-cypress (viděno 8. březen 2023).

[37] „What is CI/CD VB". https://www.redhat.com/en/topics/devops/what-is-ci-cd-vb (viděno 9. březen 2023).

[38] „What is CI/CD?" https://about.gitlab.com/topics/ci-cd/ (viděno 24. leden 2023).

[39] A. Chiarelli, „Understanding Dependency Injection in .NET Core", *Auth0 - Blog*, 10 2021. https://auth0.com/blog/dependency-injection-in-dotnet-core/ (viděno 7. srpen 2023).

[40] IEvangelist, „Dependency injection - .NET", 19. červenec 2023. https://learn.microsoft.com/en-us/dotnet/core/extensions/dependency-injection (viděno 7. srpen 2023).

[41] B. Lambson, S. Rojansky, Ł. Bartek, a R. Anderson, „Migrations Overview - EF Core", *Microsoft Technical documentation*, 12. leden 2023. https://learn.microsoft.com/en-us/ef/core/managing-schemas/migrations/ (viděno 7. srpen 2023).

[42] B. Lambson, A. Vickers, a S. Rojansky, „Applying Migrations - EF Core", *Microsoft Technical documentation*, 18. únor 2023. https://learn.microsoft.com/en-us/ef/core/managing-schemas/migrations/applying (viděno 7. srpen 2023).

[43] B. Lambson, A. Vickers, a S. Rojansky, „Managing Migrations - EF Core", *Microsoft Technical documentation*, 12. březen 2023. https://learn.microsoft.com/en-us/ef/core/managing-schemas/migrations/managing (viděno 7. srpen 2023).

[44] „Built-in React Hooks – React", *React.dev*. https://react.dev/reference/react (viděno 7. srpen 2023).

7 Přílohy

1) Zdrojový kód back-end aplikace (https://github.com/rampouch-martin/BachelorThesisBeRestApi)
2) Zdrojový kód front-end aplikace (https://github.com/rampouch-martin/BachelorThesisFeReactJs)

Příloha č . 1

UNIVERZITA HRADEC KRÁLOVÉ
Fakulta informatiky a managementu
Akademický rok: 2021/2022

Studijní program: Aplikovaná informatika
Forma studia: Prezenční
Obor/kombinace: Aplikovaná informatika (ai3-p)

Podklad pro zadání BAKALÁŘSKÉ práce studenta

Jméno a příjmení:	**Martin Rampouch**
Osobní číslo:	**I2000406**
Adresa:	Jarní 240, Poděbrady – Kluk, 29001 Poděbrady 1, Česká republika
Téma práce:	Moderní metody programování s využitím automatického testování a deploymentu
Téma práce anglicky:	Modern methods of programming using automatic testing and deployment
Jazyk práce:	Čeština
Vedoucí práce:	Ing. Jaroslav Langer Katedra informatiky a kvantitativních metod

Zásady pro vypracování:

Cílem práce je seznámení s moderními způsoby vývoje programů. Využití moderních frameworků pro vývoj backend i frontend aplikace a jejich testování. Seznámení s technologií CI/CD Pipeline a využití této technologie k nasazení aplikace. Zálohování zdrojových kódů za pomocí technologie GIT.

Pro demonstraci těchto metod bude vytvořena jednoduchá backend aplikace v technologii Microsoft .Net s využitím databázového stroje pro ukládání dat. Webové uživatelské rozhraní v technologii React js. Na této aplikaci bude následně provedeno testování jednotlivých funkcí a automatizovaný deployment.

Osnova:

- Úvod
- Cíl práce
- Teoretická část
- Praktická část
- Shrnutí a porovnání výsledků analýzy
- Závěr
- Použitá literatura

Seznam doporučené literatury:

Podpis studenta: 

Datum: 9. 8. 2023

Podpis vedoucího práce: 

Datum: