Česká zemědělská univerzita v Praze

Provozně ekonomická fakulta

Katedra informačního inženýrství

Diplomová práce

Architektura webové aplikace ASP.NET

Bc. Adam Hrouda

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Provozně ekonomická fakulta

ZADÁNÍ DIPLOMOVÉ PRÁCE

Bc. Adam Hrouda

Informatika

Název práce

Architektura webové aplikace ASP.NET

Název anglicky

Architecture for ASP.NET web application

Cíle práce

Cílem diplomové práce je navrhnout architekturu serverové části webové aplikace ASP.NET a následně vytvořit šablonu, která vygeneruje zdrojový kód navržené architektury pro nové projekty. Dílčím cílem je implementace navržené architektury v již existující aplikaci pro sjednávání studentských prací a také rozšíření funkčnosti dané aplikace.

Metodika

Diplomová práce sestává ze dvou částí – teoretické a praktické. Metodika zpracování teoretické části spočívá ve studiu odborných informačních zdrojů. Na základě syntézy zjištěných poznatků budou formulována teoretická východiska práce.

V praktické části bude navržena architektura serverové části webové aplikace ASP.NET. Na základě tohoto návrhu bude vytvořena šablona generující zdrojový kód pro nově vytvořené projekty. V rámci testování bude architektura implementována v již existující aplikaci pro sjednávání studentských prací, která bude dále rozšířena o pokročilé funkce. Při zpracování praktické části budou využívány standardní nástroje a metody softwarového inženýrství.

Poznatky a výstupy z teoretické i praktické části budou na závěr shrnuty.

Doporučený rozsah práce

60-80 stran

Klíčová slova

ASP.NET, softwarová architektura, webová aplikace

Doporučené zdroje informací

MARTIN, Robert C. Clean architecture: a craftsman's guide to software structure and design. 2018. ISBN 978-0134494166.

MICROSOFT. Microsoft Docs [online]. Dostupné z: https://docs.microsoft.com/en-gb/

PRICE, Mark J. C# 9 and .NET 5 – modern cross-platform development: build intelligent apps, websites, and services with Blazor, ASP.NET Core, and Entity Framework Core using Visual Studio Code. 2020. ISBN 978-1-80056-810-5.

SMITH, Steve. Architecting Modern Web Applications with ASP.NET Core and Microsoft Azure. Microsoft Developer Division.

Předběžný termín obhajoby

2022/23 LS – PEF

Vedoucí práce

Ing. Jiří Brožek, Ph.D.

Garantující pracoviště

Katedra informačního inženýrství

Elektronicky schváleno dne 4. 11. 2022

Ing. Martin Pelikán, Ph.D.

Vedoucí katedry

Elektronicky schváleno dne 28. 11. 2022

doc. Ing. Tomáš Šubrt, Ph.D.

Děkan

V Praze dne 22. 03. 2023

Čestné prohlášení

Prohlašuji, že svou diplomovou práci "Architektura webové aplikace ASP.NET" jsem vypracoval samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou citovány v práci a uvedeny v seznamu použitých zdrojů na konci práce. Jako autor uvedené diplomové práce dále prohlašuji, že jsem v souvislosti s jejím vytvořením neporušil autorská práva třetích osob.

V Praze dne 31. 3. 2023 _____

Poděkování

Rád bych touto cestou poděkoval Ing. Jiřímu Brožkovi, Ph.D. za poskytnuté rady a ochotu při konzultacích. Dále děkuji své rodině a blízkým za podporu při studiu.

Architektura webové aplikace ASP.NET

Abstrakt

Tato diplomová práce se zabývá návrhem architektury webové aplikace a následnou implementací jejích zásad v šabloně pro ASP.NET aplikace. První část práce tvoří formulace teoretických východisek. Jsou popsány principy při vývoji softwaru, návrhové vzory, webová API a architektury webových aplikací. Také jsou představeny technologie ASP.NET, Entity Framework a Microsoft Azure. V praktické části práce jsou stanoveny požadavky na novou architekturu, dle kterých je vytvořen návrh vrstev a zásad architektury. Architektura je navržena tak, aby byla vhodná pro individuální vývojáře či malé týmy. Následuje samotná implementace šablony včetně mechanismů, kterými jsou například automatické cachování, ošetření chybových situací, validace dat a autorizace. Dále je představena již existující aplikace pro sjednávání prací pro studenty a jsou stanoveny požadavky na další rozšíření aplikace. Aplikace je poté převedena na novou architekturu pomocí vytvořené šablony včetně nových rozšíření. Ve výsledcích jsou shrnuty výhody a nedostatky architektury i šablony objevené při migraci aplikace.

Klíčová slova: softwarová architektura, monolitická architektura, ASP.NET, C#, .NET, webová aplikace, webové API, vkládání závislostí

Architecture for ASP.NET Web Application

Abstract

This diploma thesis is focused on the design of the web application architecture and the implementation of its principles in a template for ASP.NET applications. The theoretical part of the thesis describes the principles of software development, design patterns, web API and web application architectures. Technologies such as ASP.NET, Entity Framework and Microsoft Azure are also introduced. In the second part of the thesis, the requirements for the new architecture are defined, according to which the design of the architecture is created. The architecture is designed to be suitable for individual developers or small development teams. The following is the implementation of the template itself, including mechanisms such as automatic caching, handling of error situations, data validation and authorization. Furthermore, an already existing application for arranging student jobs is introduced and requirements for further expansion of the application are defined. The application is then migrated to the new architecture using the created template including the new application extensions. The advantages and disadvantages of the architecture and the template discovered during the application migration are summarized in the results of the thesis.

Keywords: software architecture, monolithic architecture, ASP.NET, C#, .NET, web application, web API, dependency injection

Obsah

1 Úvod

S rostoucí poptávkou po webových aplikacích v různých oblastech, například e-commerce, sociální média či podnikové systémy, narůstají také požadavky na škálovatelnost, robustnost, bezpečnost a dlouhodobou udržitelnost vyvíjených aplikací. Pro uspokojení těchto požadavků hraje klíčovou roli výběr softwarové architektury ve fázi návrhu aplikace a také dodržování jejích zásad při vývoji.

Výběr softwarové architektury je ovlivněn mnoha faktory. Jde především o rozsah projektu, velikost týmu, schopnosti a dosavadní zkušenosti vývojářů, očekávání zadavatele či budoucí údržbu a rozšiřování. Architektura webové aplikace spočívá v návrhu a organizaci komponent aplikace, kterými jsou kupříkladu způsob prezentace, aplikační logika, persistence dat a integrace s jinými službami. Tyto komponenty musí hladce spolupracovat, aby vytvořily spolehlivé řešení. Dva základní typy architektur webových aplikací jsou monolitické architektury a čím dál více využívané mikroservisové architektury.

Výběrem konkrétní architektury webové aplikace je zároveň výrazně ovlivněna vývojářská zkušenost (developer experience, zkráceně DX). Vývojářská zkušenost je vše, co vývojář zažívá při vývoji produktu. Vývojářskou zkušenost dále ovlivňují využívané nástroje, procesy a prostředí.

V této práci bude navržena architektura webové aplikace vhodná pro jednoho vývojáře či malý tým. Dále bude vyvinuta šablona pro ASP.NET aplikaci, která bude dodržovat zásady navržené architektury. Tato šablona umožní vygenerovat společný zdrojový kód pro nově vytvářené projekty. Následně bude šablona využita pro již existující aplikaci Studentby, která bude převedena na nově navrženou architekturu a zároveň bude rozšířena o nové funkce. Tím dojde k objevení možných nedostatků a omezení architektury i šablony z pohledu vývojářské zkušenosti. Webová aplikace Studentby byla autorem vytvořena v rámci bakalářské práce *Aplikace pro vyhledávání a nabízení prací pro studenty* [1], kterou tato práce rozšiřuje.

2 Cíl práce a metodika

2.1 Cíl práce

Hlavním cílem diplomové práce je navrhnout architekturu serverové části webové aplikace a následně vytvořit šablonu pro ASP.NET aplikaci respektující zásady navržené architektury. Dílčím cílem je zavedení navržené architektury v již existující webové aplikaci pro sjednávání prací pro studenty a také rozšíření funkčnosti dané aplikace. Tím budou objeveny možné nedostatky architektury a také šablony z pohledu vývojářské zkušenosti.

2.2 Metodika

Diplomová práce je rozdělena do dvou částí – teoretické a praktické. Metodika zpracování teoretické části spočívá ve studiu odborných informačních zdrojů. Na základě syntézy zjištěných poznatků budou formulována teoretická východiska práce.

V praktické části bude navržena architektura serverové části webové aplikace dle stanovených požadavků. Dle navržené architektury bude vytvořena šablona pro aplikace ASP.NET, která umožní generaci společného zdrojového kódu pro nově vytvořené projekty. Následně bude pomocí šablony vytvořen nový projekt, do kterého bude přenesena již existující aplikace pro sjednávání prací pro studenty. Aplikace bude navíc rozšířena o další funkce, které budou autorem specifikovány v analytické fázi. Konkrétním využitím navržené architektury a šablony dojde k odhalení možných nedostatků a omezení. Při zpracování praktické části budou využívány standardní nástroje a metody softwarového inženýrství.

Poznatky a výstupy z teoretické i praktické části budou na závěr shrnuty.

3 Teoretická východiska

3.1 Objektově orientované programování

Objektově orientované programování je přístup vývoje softwaru, při kterém je software chápán jako soubor objektů. Výhody nad procedurálním přístupem získává především u větších systémů, kde zjednodušuje údržbu kódu, zavádění nových funkcí a změn. [2]

Objekty jsou základní stavební jednotkou a probíhají mezi nimi interakce ve formě zpráv. Objekty na tyto zprávy mohou reagovat vykonáním určité akce. Protokol objektu označuje množinu zpráv, které lze objektu zaslat a on je schopen na ně reagovat. Objekt má tři následující složky:

- identita – jednoznačně identifikuje objekt,
- stav – určuje hodnoty atributů objektu,
- chování – definuje, jak objekt reaguje na zprávy od jiných objektů. [3]

Důležitým pojmem při práci s objekty je třída. Třída představuje šablonu pro tvorbu objektů. Tyto objekty jsou nazývány instance dané třídy. Třída definuje atributy a chování pro všechny její instance. [4] Hlavními pilíři objektově orientovaného programování jsou zapouzdření, abstrakce, dědičnost a polymorfismus. [3]

Zapouzdření spočívá v zamezení přímého přístupu k atributům objektu. S objektem je zacházeno jako s tzv. černou skříňkou, která určuje způsob, jakým lze s jejími daty pracovat. Některé programovací jazyky umožňují ve třídě definovat vlastnosti. Vlastnosti poskytují řízený přístup k atributům pomocí setterů a getterů. [2]

Díky abstrakci lze uživatele odstínit od implementačních specifik, která pro něj nejsou důležitá. Uživatel poté pracuje pouze se zjednodušeným vystaveným rozhraním.

Dědičnost podporuje znovupoužitelnost, neboť umožňuje sdílení definicí mezi třídami na hierarchickém principu. Nadtřída definuje společné charakteristiky pro všechny podtřídy. Podtřídy k těmto společným definicím dodávají vlastní specifické metody či atributy.

Polymorfismus je schopnost dvou objektů zareagovat na stejnou zprávu odlišným způsobem. Tuto schopnost lze vztáhnout na zmíněnou dědičnost, kdy nadtřída definuje společné chování, ale podtřídy si toto chování upraví podle potřeby.

Objektově orientované programování se objevilo v 60. letech 20. století. Od té doby vznikaly jazyky, které byly na tomto přístupu zcela založeny nebo jej pouze podporovaly. Jako příklad lze uvést SmallTalk, C++, Java či C#. [2]

3.2 Principy při vývoji softwaru

Tyto principy lze také chápat jako nejlepší praktiky. Nejedná se tedy o pravidla, která musí být za každých podmínek dodržena, ovšem jejich respektováním je tvořen lépe udržitelný a rozšiřitelný systém bez pachů v kódu. [5] [6]

Pachy v kódu (code smells) nemusí být nutně problémy. Jedná se spíše o indikátory problémů ve zdrojovém kódu. Výskyt těchto pachů často znamená porušení některého z principů. Jako příklad pachů v kódu lze uvést příliš dlouhou metodu ve třídě nebo metodu, která provádí více činností. [7]

3.2.1 KISS

KISS nebo také „Keep It Simple Stupid“ je princip, dle kterého má programátor předcházet zbytečné komplexitě při vývoji softwaru. Jednoduchost totiž napomáhá lepší přehlednosti zdrojového kódu a snazšímu ladění chyb. Tento princip lze využít pro jakkoliv velké aplikace, dokonce i mimo oblast softwarového vývoje.

Posouzení, zda je zdrojový kód v souladu s principem KISS, může být složité. Je nutné najít rovnováhu mezi jednoduchostí a dodržováním dalších principů, které mohou navýšit komplexitu kódu výměnou za jiné přínosy. [6] [8]

3.2.2 DRY

DRY je akronymem pro „Don’t Repeat Yourself“ a jeho dodržování spočívá v eliminaci duplicit. Toho lze dosáhnout rozčleněním kódu do znovupoužitelných bloků, s čímž pomáhá i objektový přístup. Výhodu lze spatřit především při zavádění změn ve zdrojovém kódu. Stačí totiž změnu provést pouze jednou a předchází se tak možným nekonzistencím. [8]

3.2.3 SOLID

SOLID je akronym pěti dílčích principů pro objektově orientovaný návrh. Tyto principy popisují ověřené postupy, jak správně uspořádat třídy a jakým způsobem by měly být třídy propojeny. [9]

Návrhové pachy

Cílem SOLID je vyhnout se návrhovým pachům (design smells). Návrhové pachy stejně jako pachy v kódu indikují nějaký problém, tentokrát je však pohled širší než pouze na samotný zdrojový kód. Níže jsou popsány příklady zmíněných návrhových pachů.

Ztuhlost (rigidity) softwaru se projevuje obtížným zaváděním, třeba i malých, změn. V případě jedné změny vyžaduje rigidní návrh kaskádu dalších změn v závislých modulech.

Křehkost (fragility), stejně jako zmíněná ztuhlost, se týká změn. V tomto případě jediná změna způsobuje problémy v oblastech, které nemají na změněnou oblast konceptuální vazby.

Obtížná znovupoužitelnost (immobility) omezuje možnost zavedení částí softwaru i v jiných aplikacích, kde by byly užitečné.

Zbytečně komplexní (needless complexity) návrh obsahuje prvky, které nejsou užitečné. Mohou existovat z důvodu, že vývojář očekává jisté změny v požadavcích, které nemusí přijít. [6]

Principy pro objektově orientovaný návrh

Bylo zmíněno, že SOLID pod sebou skrývá pět dílčích principů, kde každý představuje jedno písmeno. Jedná se o následující principy:

- princip jedné odpovědnosti (Single Reponsibility Principle),
- princip otevřenosti a uzavřenosti (Open-closed Principle),
- Liskovové princip zaměnitelnosti (Liskov Substitution Principle),
- princip oddělení rozhraní (Interface Segragation Principle),
- princip obrácení závislostí (Dependency Inversion Principle). [9]

Dle principu jedné odpovědnosti má mít každá třída právě jednu odpovědnost. Tato odpovědnost má být zřejmá, například již z názvu. Také je uváděna definice, že každá třída má mít pouze jeden důvod pro změnu. Při porušení tohoto principu může nastat situace, že změnou ve třídě jsou ovlivněny dvě oblasti, jedna úmyslně a jedna neúmyslně. Pokud je tento princip respektován vznikají menší třídy a vývojář snáze najde místo, kde má změnu provést. [6]

Princip otevřenosti a uzavřenosti říká, že třída má být otevřená pro rozšíření a uzavřená vůči změnám. Znamená to tedy, že již využívaná a správně fungující třída nemá být jakkoliv měněna, jelikož takováto změna může být důvodem vzniku chyb fungujícího softwaru.

Otevřenost pro rozšíření však umožňuje reagovat na změny existujících požadavků či na nové požadavky, lze k tomu využít například dědičnost a polymorfismus. [10]

Liskovové princip zaměnitelnosti říká, že podtřídy mají být zaměnitelné s jejich nadtřídami. Pokud tedy třída *S* dědí ze třídy *T*, pak lze kdekoliv ve zdrojovém kódu nahradit instanci třídy *T* instancí třídy *S* bez narušení funkčnosti. Porušení tohoto principu mohou nastat především při využívání polymorfismu skrze dědičnost. [9]

Princip oddělení rozhraní není porušen, pokud uživatelé tříd nejsou závislí na částech třídy, které nevyužívají. Pokud má tedy třída více uživatelů, měla by implementovat více menších rozhraní. Problém při porušení tohoto principu může vzniknout ve chvíli, kdy je požadováno nahradit původní implementaci. Aby byl zachován kontrakt původní třídy, tak je nutné v nové třídě přidat prvky i v případě, že nejsou využívány. [9]

Naplnění principu obrácení závislostí spočívá ve zrušení závislostí tříd vyšší úrovně na konkrétních třídách nižší úrovně. Závislost má být místo toho na rozhraních či abstraktních třídách. Tímto principem je odstraněno úzké provázání (tight coupling) mezi třídami. Pro tyto postupy je také užíván pojem „programování proti rozhraní“. Problémy úzkého spojení opět souvisí se změnami, pokud je požadována změna jedné třídy, pak musí být přizpůsoben kód i úzce provázané třídy. [6] [10]

3.3 Návrhové vzory

Návrhové vzory v objektově orientovaném programování poskytují obecná řešení pro problémy vyskytující se v kontextu vývoje softwaru. Tím se liší od principů zmíněných v předešlé kapitole, které byly pouze abstraktními pokyny pro zajištění lepší kvality softwaru. Návrhové vzory jsou zaměřeny na dobře známé a často se vyskytující problémy. Ve chvíli, kdy vývojář takovému problému čelí, má k dispozici již otestované řešení, které mu ušetří čas. Zároveň, díky obecné znalosti těchto vzorů, je pro ostatní vývojáře zdrojový kód snáze pochopitelný. [10] [11]

Důležité je, že návrhové vzory nepředstavují hotové kusy zdrojového kódu, které stačí jen zavést do vlastního řešení. Jedná se spíše o šablony, které poradí, jaké struktury vytvořit a jak je vzájemně provázat. Samotná implementace je čistě na programátorovi, který má rovněž možnost řešení dále upravit dle potřeb. [11]

Velmi populárním zdrojem těchto vzorů je kniha *Design Patterns: Elements of Reusable Object Oriented Software* [12], jejíž autoři jsou Erich Gamma, Richard Helm, Ralph Johnson a John Vlissides. Návrhové vzory z tého knihy jsou dnes známy jako

GoF (Gang of Four) vzory označující čtyři autory uvedené literatury. Existují však i návrhové vzory z jiných zdrojů. [10]

Návrhových vzorů Gang of Four je uvedeno celkově 23 a jsou rozděleny do tří kategorií: vytvářející (creational), strukturální (structural), chování (behavioral). Do těchto kategorií jsou však často členěny i některé návrhové vzory z jiných zdrojů. Vytvářející vzory mají na starost vytváření objektů. Strukturální vzory řeší kompozici tříd. Vzory chování se vztahují k interakci tříd a objektů. Dle GoF jsou vzory definovány následujícími elementy:

- název vzoru,
- problém řešený vzorem,
- řešení,
- důsledky.

Dále je uvedena rozsáhlá doporučená struktura popisu každého návrhového vzoru. Popis má obsahovat například využívané třídy a jejich komunikaci, grafické znázornění vzoru nebo příklad implementace v konkrétním programovacích jazyce. [12]

3.3.1 Vkládání závislostí

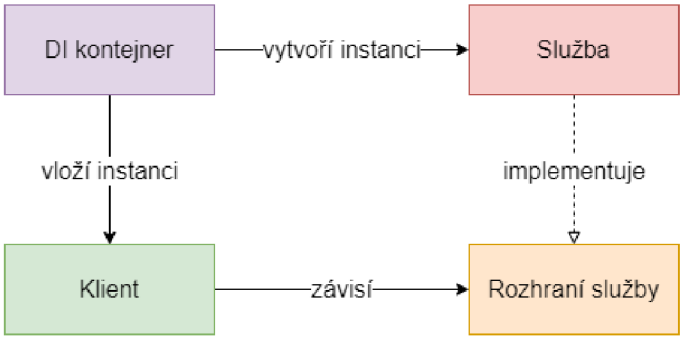
Návrhový vzor Dependency Injection (DI), bývá překládán jako vkládání závislostí, ruší úzkou provázanost mezi třídami. Použití vkládání závislostí je v souladu s již popsaným principem obrácení závislostí. Závislé třídy tedy nepracují s implementacemi jiných tříd (závislostí), ale pouze s jejich rozhraními. Avšak závislé třídy stále potřebují získat jejich instance a sami si je vytvořit nesmějí, jelikož by zůstaly svázány s danou implementací. Zde přichází vkládání závislostí, které zajistí vytvoření instancí těchto konkrétních tříd a jejich vložení do závislých tříd. [13] [14]

Jednotlivé implementace tohoto návrhového vzoru bývají označovány jako DI kontejnery a starají se o vytváření a předávání instancí závislostí. Existují různé způsoby, jak je vložení řešeno. Často je využíváno vložení skrze konstruktor, kdy závislá třída v parametrech konstruktoru specifikuje, které závislosti vyžaduje. Využíváno je však i vkládání skrze vlastnosti nebo skrze metody. Některé DI kontejnery také umožňují změnu vkládané závislosti při běhu aplikace. Vkládání závislostí je vizualizováno na obrázku 1. [14]

Hlavní výhodou využití tohoto návrhového vzoru je především snadná nahraditelnost závislostí. Stačí pouze zajistit, aby nová implementace závislosti dodržela původní rozhraní. Protože vytváření instancí má na starosti pouze DI kontejner, tak stačí změnit původní kód

pouze na jednom místě a do závislé třídy není nutné vůbec zasahovat. Toto může být vhodné při využívání externích knihoven, jelikož se může vyskytnout spousta objektivních důvodů pro výměnu implementace, například nevýhodné licenční podmínky, neopravené chyby nebo ukončení podpory. [13]

Snadná nahraditelnost je však důležitá i v další oblasti vývoje, a tou je testování. Pokud je potřeba otestovat třídu, která závisí na klientovi komunikujícím s databází, tak je možné se komunikaci s databází vyhnout. Lze toho docílit vytvořením nové implementace dané závislosti, která ve skutečnosti s databází komunikovat nebude. Do testované třídy se poté pouze vloží tato nová implementace. Tento postup nahrazování původních implementací pro účely testování se nazývá mockování (mocking). [13]



Obrázek 1 – Vkládání závislostí, vlastní zpracování dle zdroje: [15]

3.3.2 Opožděná inicializace

Opožděná inicializace (Lazy Initialization) patří mezi tvořivé návrhové vzory a odkládá inicializaci proměnné až na okamžik prvního přístupu k dané proměnné. Toto je vhodné především v situacích, kdy je inicializace proměnné nákladná, a navíc existuje možnost, že proměnná nebude v daném kontextu vůbec využita. Je ovšem nutné dávat pozor, pokud je při ladění kódu využíváno nahlížení na hodnoty proměnných pomocí vypisování na obrazovku, jelikož vypsáním hodnoty může být spuštěna inicializace a tedy i změněno chování aplikace. [16]

3.3.3 Adaptér

Adaptér (Adapter nebo Wrapper) je strukturální návrhový vzor z rodiny GoF vzorů. Motivací pro zavedení adaptéru je možnost využít existující třídu, přestože je očekáváno jiné rozhraní. Může se jednat o situaci, kdy aplikace využívá knihovnu *A*, která implementuje rozhraní *I*, ale je požadováno nahradit *A* novou knihovnou *B*, která implementuje rozhraní *J*.

Bez adaptéru by bylo nutné přepsat všechen kód, který závisí na rozhraní *I*, tak aby nyní pracoval s rozhraním *J*. Adaptér tuto situaci řeší tak, že vznikne nová třída *C*, která implementuje původní rozhraní *I*. Úpravy, aby byla rozhraní *I* a *J* kompatibilní, jsou tedy pouze v nové třídě *C*. [10] [12]

3.3.4 Repozitář

Repozitáře (Repository) slouží k oddělení přístupu k datům od zbytku logiky aplikace. Repozitář je třída, která obsahuje metody pro práci s jedním datovým zdrojem, v případě relačních databází pracuje repozitář s jednou tabulkou. Může tedy zapouzdřovat volání konkrétních SQL dotazů, využívání ORM nástroje či volání API. Lze rovněž vytvořit generický repozitář, který slouží jako nadtřída pro jiné repozitáře. Repozitáře bývají využívány ve třídách, které implementují aplikační logiku. Využití repozitářů je v souladu s principem jedné odpovědnosti a v případě využití vkládání závislostí umožňuje mockování přístupu k datům. [10] [16]

3.3.5 Guard

Guard nebo také Guard Clause pomáhá implementovat techniku fail-fast. Dle této techniky je doporučováno snažit se o včasnou detekci chybových stavů a v případě, že došlo k chybě, přerušit standardní tok. Výhodou fail-fast techniky je snadnější dohledávání chyb, jelikož se může stát, že by chybový stav způsobil přerušení až mnohem později a původce problému by se poté obtížně dohledával. [17] [18]

Guard může být využit na začátku metody, kde validuje hodnoty vstupních parametrů. Lze tedy na začátek metody rozepsat jednotlivé podmínky a v případě selhání validace ukončit metodu nebo vyhodit výjimku. V případě často opakovaných validací, například pro hodnoty *null*, lze validace umístit do vlastní statické třídy a tím být v souladu s principem DRY. [18]

3.3.6 Stavitel

Stavitel (Builder) je vytvářející návrhový vzor GoF, který odděluje konstrukci složitých objektů od jejich reprezentace. Pomáhá tedy vytvořit novou instanci objektu ve chvíli, kdy je přímé volání konstruktoru příliš komplexní. Může se jednat o konstruktory s velkým množství parametrů, které mohou být navíc nepovinné. Stavitel je tedy třída, jejíž jedinou zodpovědností je vytvoření nové instance objektu. Stavitel má vnitřní stav, který

určuje výsledný produkt a klient může tento stav měnit pomocí veřejných metod, dále vystavuje metodu, která dle vnitřního stavu vytvoří nový objekt. [10] [12]

3.3.7 Řetěz odpovědnosti

Řetěz odpovědnosti (Chain of Responsibility) poskytuje řešení zpracování libovolného druhu požadavků. Navrhuje, aby proces zpracování nenáležel jednomu objektu, ale více objektům, přičemž každý z nich aplikuje na požadavek jinou logiku. Tento vzor tedy vychází z principu jedné odpovědnosti. Požadavek je v řetězci postupně předáván mezi handlery. Jakmile tedy první objekt v řetězci požadavek zpracuje, předá jej dalšímu a ten po zpracování opět předává požadavek dál. Tento vzor může být využit například při zpracování HTTP požadavků webovou aplikací. [10]

3.3.8 Prostředník

Mezi návrhové vzory chování z rodiny GoF patří prostředník (Mediator). Prostředník se vztahuje k situaci, kdy je systém rozčleněn do mnoha objektů, které vzájemně komunikují. Problém spočívá v úzkém provázání těchto objektů a také v nepřehlednosti. Návrhový vzor prostředník definuje způsob, jak spolu mohou objekty komunikovat, aniž by věděly o implementačních detailech příjemce. [12]

Prostředník představuje objekt, přes který probíhá veškerá komunikace. Pokud chce objekt *A* komunikovat s objektem *B*, tak musí *A* zaslat zprávu prostředníkovi, kde jsou nastavena směrovací pravidla takovým způsobem, aby se zpráva dostala k *B*. Provázání objektů se tedy již nenachází v komunikujících objektech, ale v prostředníkovi. Tento způsob komunikace navíc umožňuje definovat další logiku, která může být aplikována na každou zaslanou zprávu, zde lze využít zmíněný návrhový vzor řetěz odpovědnosti. [10] [12]

3.4 HTTP

HTTP (Hypertext Transfer Protocol) je síťový protokol aplikační vrstvy, který je využíván při komunikaci v modelu klient-server. Klient funguje aktivně a posílá požadavky na server a ten odesílá zpět odpovědi. Tato komunikace je bezstavová, jelikož server neudržuje žádný stav mezi dvěma požadavky.

HTTP požadavek je tvořen čtyřmi základními prvky. Prvním je identifikátor cílového zdroje na serveru, může se jednat o absolutní či relativní adresu. Dále je uvedena metoda

požadavku, ta specifikuje akci, která má být se zdrojem provedena. Požadavek je dále tvořen hlavičkami, ve kterých jsou uvedeny další informace o požadavku. Čtvrtým prvkem je tělo zprávy, které může být i prázdné. [19]

Odpověď ze serveru má, stejně jako požadavek, hlavičky a tělo, dále je však odesílán i stavový kód. Stavový kód je tříciferné číslo, které udává výsledek zpracování požadavku. Kupříkladu 200 – Ok, 201 – Created, 400 – Bad request, 401 – Unauthorized, 404 – Not found, 500 – Internal server error.

Výše bylo zmíněno, že požadavek je tvořen také metodou, která určuje prováděnou operaci se zdrojem. Zde je přehled často používaných metod.

- GET – slouží k získání dat dotazovaného zdroje.
- POST – zasílá data na server, kde poté bývá vytvořen nový zdroj.
- PUT – nahradí existující zdroj novým.
- PATCH – slouží k parciální úpravě příslušného zdroje.
- DELETE – odstraní příslušný zdroj. [19] [20]

Pro metody HTTP požadavků jsou dále uvedeny dva pojmy – bezpečnost a idempotence. Bezpečné metody jsou takové, které nezpůsobují změnu na serveru, jedná se tedy o čtecí operace. Z uvedených metod se jedná o GET. Existují však i jiné bezpečné metody, například HEAD a OPTIONS. Metoda je idempotentní právě tehdy, pokud zaslání více identických požadavků má stejný účinek jako zaslání pouze jednoho požadavku. Všechny bezpečné metody jsou tedy zároveň idempotentní, ovšem idempotentní jsou i metody PUT a DELETE. [21] [22]

HTTP protokol a identifikace zdrojů URI jsou dále rozvedeny v bakalářské práci *Aplikace pro vyhledávání a nabízení prací pro studenty* [1], kterou tato práce rozšiřuje.

3.5 Webová API

API (Application Programming Interface) umožňuje v kontextu webu komunikaci mezi klientem a serverem, přičemž klient je reprezentován webovým prohlížečem. Existuje více způsobů, jak tuto komunikaci realizovat, například REST, SOAP, RPC a WebSocket. [23]

REST (Representational State Transfer) je návrh od Roye Fieldinga pro přenos dat mezi systémy. Samotný koncept není vázaný na konkrétní protokol ani formát přenášených dat. V současné době je nejčastější způsob implementace REST API založen na využití protokolu HTTP a formátu JSON. Architektura REST definuje následující pravidla.

- Architektura klient-server – klient aktivně posílá požadavky a server odpovídá.
- Bezstavovost – server neukládá stav klienta.
- Cache – lze uvést, zda mohou být odpovědi načítány z mezipaměti.
- Jednotné rozhraní – rozhraní serveru zůstává stejné pro jakéhokoliv klienta.
- Vrstvený systém – služba je rozčleněna do oddělených vrstev.
- Code-On-Demand – zasílání spustitelného kódu ze serveru na klienta. [24]

SOAP (Simple Object Access Protocol) je založen na výměně zpráv ve formátu XML. Na rozdíl od REST je SOAP orientován na služby. Nevýhody spočívají ve vazbě na formát XML, který bývá náročnější pro serializaci a deserializaci.

RPC (Remote Procedure Call) má velmi jednoduchou podstatu. Klient zde pomocí požadavků spouští akce uložené na serveru, po dokončení může server odeslat výsledek akce zpět. RPC není vázán na konkrétní formát zpráv.

WebSocket umožňuje oboustrannou komunikaci klienta a serveru. Podporuje tak přenos dat ze serveru na klienta v reálném čase. [23]

S webovými API dále souvisí dotazovací jazyk GraphQL. GraphQL specifikuje, jakým způsobem tvořit dotazy, které jsou zasílány na serverem vystavený koncový bod. Tento požadavek může dotazovat více zdrojů najednou a přesně specifikovat, které údaje mají být k danému zdroji vráceny. [25]

3.6 Zabezpečení webových API

Webová API jsou kritickou částí moderních webových aplikací, jelikož zajišťují aplikační logiku, zpracovávají citlivá data a mohou být i veřejně dostupná, právě proto jsou častým terčem útoků. [26] Dále budou uvedeny oblasti ochrany relevantní pro praktickou část této práce.

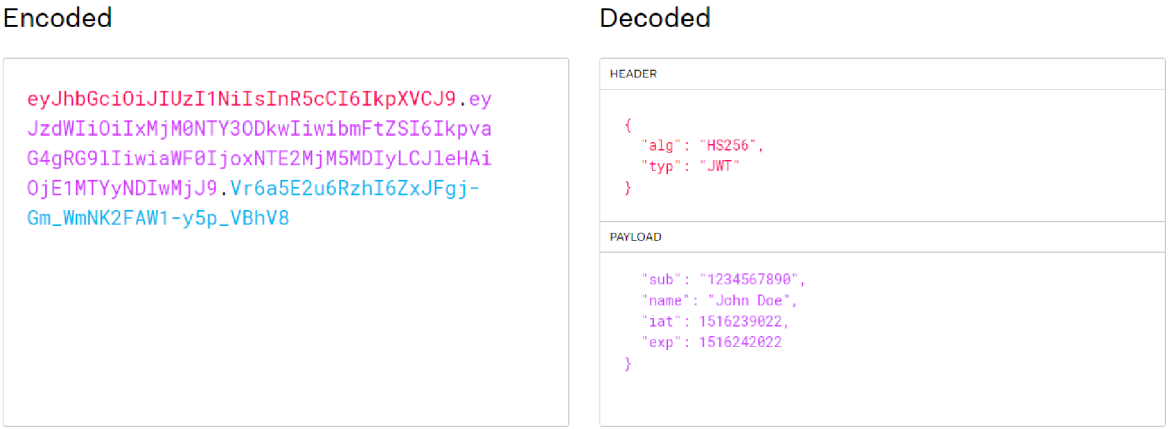
CORS (Cross-Origin Resource Sharing) spočívá ve využívání speciálních HTTP hlaviček. Většina webových prohlížečů automaticky odesílá hlavičku *Origin*, ve které je adresa webové stránky, která požadavek započala. Na cílovém serveru lze následně nastavit domény, ze kterých server smí požadavky přijmout. Výchozím nastavením bývá přijímaní požadavků pouze ze stejné domény. Je však nutné zmínit, že CORS funguje pouze při odesílání požadavků z prohlížečů, které CORS podporují. Existují prohlížeče či jiné způsoby odeslání požadavku, které CORS hlavičku ignorují, pak lze toto zabezpečení snadno obejít. [27]

Důležitou oblastí při zabezpečení API je řízení přístupu. Vystavené API bývá zřídka veřejně přístupné v plném rozsahu, často je žádoucí povolit přístup na koncový bod pouze ověřeným uživatelům. Případně lze řídit přístup dle uživatelských rolí či jiných vlastností.

V současné době se pro řízení přístupu často využívá JWT (JSON Web Token). JWT je tvořen hlavičkou, tělem a podpisem. Je však zakódován a má tedy podobu textového řetězce. V těle tokenu bývá uveden například čas vytvoření tokenu, expirace tokenu či informace o uživateli. Token je vygenerován autorizační službou, která jej podepíše privátním klíčem. Tato služba může být zabudovaná v daném webovém API či se může jednat o samostatnou komponentu. Ukázka zakódovaného JWT včetně dekódované hlavičky a těla je na obrázku 2.

Celý proces může vypadat následovně. Klient odešle autorizační službě přihlašovací údaje, například email a heslo. Služba vygeneruje přístupový token a odešle jej v odpovědi. Klient při požadavku na zabezpečený koncový bod posílá token v hlavičce *Authorization*, přičemž hodnota hlavičky má tvar „Bearer <token>“. Cílové API poté zvaliduje přijatý token a vrátí zabezpečený zdroj.

Využití JWT funguje bezstavově, server nemá token nikde uložený a zda nebyl token změněn lze ověřit díky zmíněnému podpisu. Toto se však pojí s nevýhodou, že nelze token zrušit, životnost tokenu tedy bývá krátká, aby v případě odcizení stihl útočník napáchat co nejméně škod. Aby se uživatel nemusel přihlašovat pokaždé, když přístupový token vyprší, byl zaveden do procesu ještě refresh token. Refresh token již na serveru uložen je a umožňuje získat nový přístupový token bez interakce uživatele. [28] [29]



Obrázek 2 – Dekódovaný JWT, snímek obrazovky z nástroje https://jwt.io

Dále je vhodné, aby komunikace mezi serverem a klientem probíhala zabezpečeným protokolem HTTPS, který chrání komunikaci před odposlechem či útokem man-in-the-middle. Také by měly být validovány vstupní parametry jednotlivých koncových bodů, aby

bylo znemožněno například zneužití filtrování k získání dat, která nemají být přístupná. Soukromé klíče a jiná tajemství je nutné bezpečně uchovávat. V cloudovém prostředí lze pro tyto potřeby využít speciální služby pro správu klíčů. [28]

Téma REST API je dále rozvedeno v bakalářské práci *Aplikace pro vyhledávání a nabízení prací pro studenty* [1], kterou tato práce rozšiřuje.

3.7 Typy webových aplikací

Tvorba webových aplikací se neustále vyvíjí a je využívána v mnoha oblastech. Různé požadavky a technologický vývoj zapříčinily vznik více typů webových aplikací. V současné době se rozlišují dynamické a statické webové aplikace, přičemž dynamické lze dále členit na jednostránkové a vícestránkové. [30]

3.7.1 Statické a dynamické webové aplikace

Statické webové aplikace zobrazují všem uživatelům stejný neměnný obsah. Jediný způsob, kterým lze změnit obsah, je zásah do kódu, poté je však nutné změny znovu zveřejnit, tedy provést nasazení. Jsou využívány především technologie HTML a CSS, JavaScript je využit pro zlepšení uživatelského rozhraní, ale ne pro změny obsahu.

Dynamické webové aplikace představují opak statických. Využívají se v případě, že se zobrazovaný obsah mění. Například stejná stránka s detailem události může pokaždé zobrazit informace o jiné události. Zobrazovaná data nejsou trvale zapsána ve zdrojovém kódu, ale bývají získávána z databází či jiných služeb. Dva využívané způsoby, jak získání dat realizovat jsou jednostránkové a vícestránkové webové aplikace. [30]

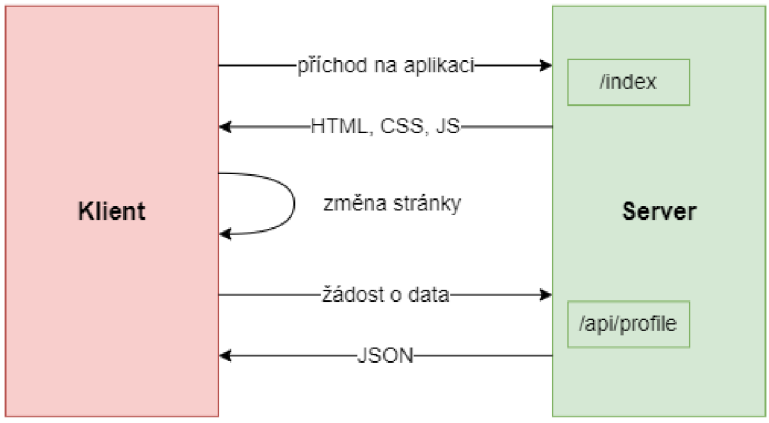
3.7.2 Jednostránkové a vícestránkové webové aplikace

Jednostránkové webové aplikace, v angličtině single-page application (SPA), nevyžadují znovunačtení stránky při změně obsahu. Nejprve je do prohlížeče načtena aplikace a veškeré další změnty již obstarává JavaScript. Zdánlivá navigace mezi stránkami je ve skutečnosti pouhá změna části HTML kódu a případně URL adresy. Získávání dat je realizováno asynchronním voláním webových API. Mezi serverem a prohlížečem jsou tedy přenášena pouze data a o jejich zobrazení se postará kód na straně klienta. Vývoj klientské a serverové části aplikace může probíhat odděleně, je však nutné respektovat dohodnutá rozhraní. Pro tvorbu jednostránkových aplikací dnes existuje mnoho tzv. frontendových frameworků, mezi populární zástupce patří React, Vue či Angular. [14]

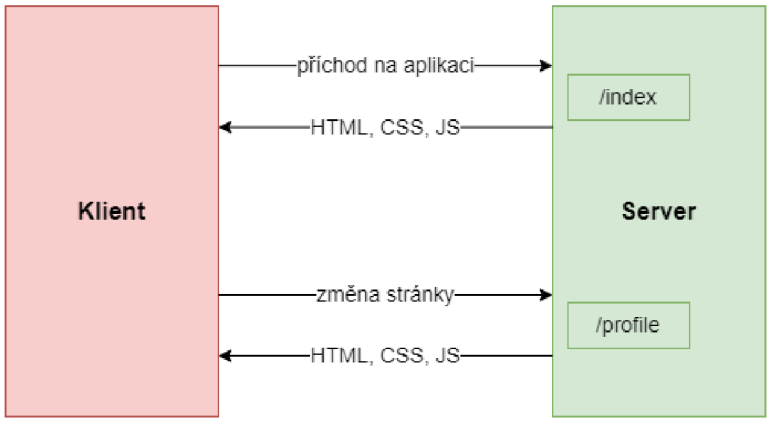
Výhodou tohoto typu webových aplikací je především rychlost načítání, jelikož není nutné čekat než server odešle celou stránku a než ji prohlížeč celou vyrenderuje. Problém je však s optimalizací pro vyhledávače (SEO), jelikož web crawler nemůže indexovat obsah, který je zkompletován teprve až na straně klienta. V současné době ale existují způsoby, jak při prvním načtení SPA vyrenderovat obsah již na straně serveru. [30]

Vícestránkové webové aplikace jsou na rozdíl od jednostránkových sestaveny na straně serveru. Pokud si tedy prohlížeč vyžádá webovou stránku, tak se tato stránka na serveru vygeneruje z dat, která mohou být získána například z databáze. Nevýhodou je delší načítání při navigaci v aplikaci. Technologie využívané při vývoji tohoto typu aplikací jsou kupříkladu PHP frameworky, ASP.NET, Ruby on Rails. [14] [31]

Schéma fungování jednostránkových a vícestránkových aplikací je na obrázcích 3 a 4.



Obrázek 3 – Jednostránková webová aplikace, vlastní zpracování dle zdroje: [14]



Obrázek 4 – Vícestránková webová aplikace, vlastní zpracování dle zdroje: [14]

3.8 Architektury webových aplikací

Softwarová architektura určuje strukturu komponent softwaru a také popisuje jejich role a vzájemné vazby. Při návrhu architektury je kladen důraz především na udržitelnost řešení. Softwarová architektura však neřeší výběr konkrétních technologií, například databázové systémy, to jsou implementační detaily, o kterých může být rozhodnuto později. V oblasti webových aplikací lze softwarové architektury rozdělit na monolitické a mikroservisové (microservices). [14] [32]

Webová aplikace s monolitickou architekturou je nasazována jako samostatná jednotka. Díky této skutečnosti je zajištěno snadné a rychlé nasazení. Celá aplikace je tvořena jednou základnou zdrojového kódu, což přispívá ke snadnějšímu vývoji a ladění chyb. S přírůstkem požadavků na software však narůstá komplexita zdrojového kódu a s tím i riziko nepřehlednosti. Nevýhodnou je také horizontální škálovatelnost, kterou lze řešit jedině přidáním nové instance celé aplikace. Nyní existuje více dílčích architektur, které se řadí mezi monolitické, jsou to kupříkladu all-in-one, n-vrstvá a clean architecture. [14] [33]

Na druhé straně je mikroservisová architektura. Zde je webová aplikace tvořena několika samostatnými službami, které spolu mohou komunikovat. Každé z těchto služeb je přidělena určitá oblast řešení, kterou zajišťuje. Tento přístup neredukuje komplexitu celého řešení, ale dělá ho přehlednějším. Zároveň umožňuje, aby na každé službě pracoval jiný tým. Na rozdíl od monolitické architektury podporuje horizontální škálování. Nasazení celé aplikace je však obtížné, jelikož je nutné nasadit a nakonfigurovat každou službu zvlášť. V případě, že aplikaci vyvíjí jeden tým, může být vývoj pomalejší, jelikož je aplikace rozložena do více základen zdrojového kódu. [33]

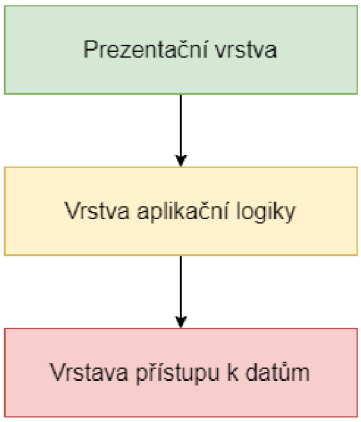
3.8.1 All-in-one architektura

All-in-one architektura představuje nejjednodušší organizaci zdrojového kódu. Strukturování je zajištěno pomocí adresářů a souborů. V případě platformy .NET je celá aplikace reprezentována jedním projektem a výstupem je jedna sestava (assembly). Nevýhodou této architektury je, že se v jednom projektu nachází prezentační logika, aplikační logika i přístup k datům. S rostoucím projektem vynikají další soubory a je čím dál obtížnější udržovat přehled o závislostech. Architektura zároveň znemožňuje znovuvyužití izolované aplikační logiky aplikace pro jinou formu prezentace. [14]

3.8.2 N-vrstvá architektura

N-vrstvá architektura spočívá v logickém rozdělení aplikace do vrstev. Každá vrstva má určitou úroveň abstrakce a zodpovědnost. Architektura přesně stanoví způsob, jak mohou vrstvy na sobě záviset. Stále se však jedná o monolitickou architekturu, existuje tedy jen jeden celek, který je nasazován. [9] [14]

Webové aplikace s touto architekturou často využívají tři vrstvy: prezentační vrstva, vrstva aplikační logiky a vrstva přístupu k datům. Jsou zde dána omezení, že prezentační vrstva může využívat pouze vrstvu aplikační logiky a ta smí využívat pouze vrstvu přístupu k datům (viz obrázek 5). Důležité je, aby vrstvy byly implementovány bez jakékoliv vazby na vyšší vrstvy, například vrstva aplikační logiky nezná způsob, jakým bude poskytnuta uživateli, zda to bude webové API, konzolová aplikace či desktopová aplikace. N-vrstvá architektura je v .NET platformě realizována pomocí více projektů, přičemž výstupem každého projektu je jedna sestava, kterou lze následně znovupoužít. [14]



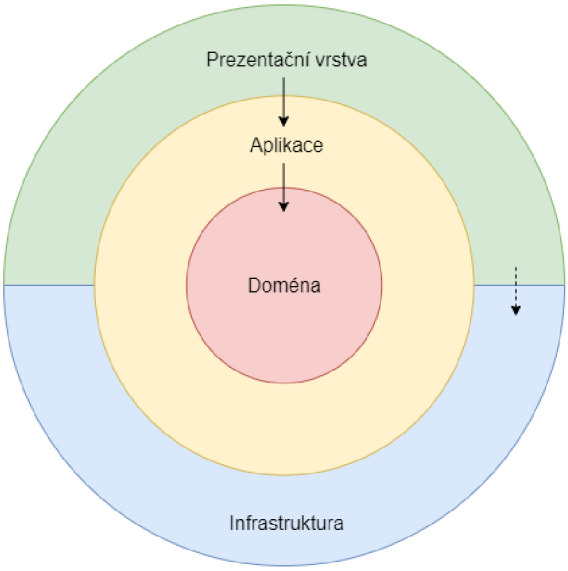
Obrázek 5 – N-vrstvá architektura, vlastní zpracování dle zdroje: [14]

3.8.3 Clean architecture

Clean architecture je architektura podobná n-vrstvé, avšak respektuje princip obrácení závislostí. Tato architektura se objevuje i pod jinými názvy například onion architecture, hexagonal architecture, ports-and-adapters. Tyto varianty jsou založeny na stejném principu, mohou se však lišit v uspořádání, pojmenování, počtu a závislosti vrstev. Podstatnou nevýhodou n-vrstvé architektury je závislost na implementačních detailech, například vrstva přístupu k datům je závislá na využívaném databázovém systému a vrstva aplikační logiky je závislá na konkrétních typech přístupu k datům. Tato silná provázanost pak způsobuje obtíže při změně využívaného databázového systému či při izolovaném testování jednotlivých vrstev. [14]

Architektury respektující princip obrácení závislostí přesouvají implementační detaily do vrstvy, která bývá označována jako infrastruktura (infrastructure). Závislosti mezi vrstvami mohou vznikat pouze na rozhraních. Použití konkrétních implementací bývá řešeno pomocí vkládání závislostí, které bylo popsáno v kapitole 3.3.1. Clean architecture je často zobrazována jako několik soustředných kruhů, viz obrázek 6. Na tomto obrázku jsou vrstvy doplněny šipkami. Plná šipka představuje standardní závislost mezi vrstvami a přerušovaná šipka je nestandardní závislost pro využití vkládání závislostí. Prezentační vrstva obsahuje veškerý kód související s interakcí uživatele s aplikací, v případě webového API se jedná o vystavení koncových bodů. Prezentační vrstva je vstupním bodem programu a zajišťuje tedy registraci závislostí do DI kontejneru, kvůli tomu musí mít nestandardní závislost na infrastruktuře. Vrstvy aplikace a doména spolu tvoří aplikační jádro. Vrstva aplikace obsahuje rozhraní pro aplikační logiku, ale i konkrétní implementace, jelikož se zde nejedná o implementační detaily. Ve vrstvě domény se nachází třídy entit či jiných struktur pro přístup k datům. [9] [14]

Tento typ monolitické architektury podporuje znovupoužitelnost aplikačního jádra pro různé typy aplikací, jelikož stačí vytvořit nový projekt prezentační vrstvy s referencí na existující projekty ostatních vrstev. Dále umožňuje snadnou změnu implementačních detailů, tu stačí provést pouze v místě, kde je implementace registrována do DI kontejneru. Díky principu vkládání závislostí lze jednoduše a izolovaně otestovat jednotlivé vrstvy za využití mockování. Dodržováním výše uvedených zásad architektury vzniká přehledné řešení s volně provázanými vrstvami. [14] [34]



Obrázek 6 – Clean architecture, vlastní zpracování dle zdroje: [14]

3.9 Microsoft Azure

Microsoft Azure je cloudová platforma, která zákazníkům poskytuje různorodé zdroje v podobě služeb prostřednictvím Internetu. Původní název při zveřejnění v roce 2010 byl Windows Azure, avšak podporou operačních systémů s linuxovým jádrem přerostla platforma hranice systému Windows a došlo ke změně názvu. Populárními alternativami pro tuto platformu jsou AWS (Amazon Web Services) a GCP (Google Cloud Platform). [35]

3.9.1 Zdroje

Zdroje v Microsoft Azure jsou například virtuální stroje, databázové servery nebo služby pro strojové učení. Platforma nabízí velké množství různých typů těchto zdrojů a pro lepší přehlednost jsou rozděleny do kategorií. Kategorie jsou uvedeny v následujícím výčtu spolu s příkladem zařazených zdrojů.

- **AI and Machine Learning** – Cognitive Services, Speech API.
- **Analytics** – Azure Synapse Analytics, Azure Databricks.
- **Compute** – App Services, Azure Functions, Virtual Machine.
- **Containers** – Azure Kubernetes Service.
- **Databases** – Azure SQL, Azure Cosmos DB.
- **Developer tools** – Azure DevOps.
- **Identity** – Azure Active Directory.
- **Integration** – Logic App, Service Bus.
- **Internet of Things** – IoT Hub.
- **Mixed Reality** – Azure Digital Twins.
- **Networking** – Virtual Network, Load Balancer.
- **Security** – Azure DDoS Protection, Key Vault.
- **Storage** – Storage Account. [36]

Azure Active Directory

Služba Azure Active Directory, také známá jako AAD, spravuje uživatelské účty daného tenantu. Lze v ní také spravovat uživatelské skupiny, licence, role, bezpečnostní politiky a další.

AAD v souvislosti s vývojem aplikací umožňuje vytvářet aplikační objekty (application objects). Tyto aplikační objekty otevírají možnost využívat AAD jako externího

poskytovatele identit, uživatelé se pak mohou do aplikace přihlašovat Microsoft účtem. V aplikačním objektu lze však také nastavit oprávnění aplikace přistupovat k dalším zdrojům patřícím do ekosystému Microsoft, například čtení emailů uživatele. [37] [38]

App Services

App Services je služba pro hostování webových aplikací na platformě Microsoft Azure. Jedná se o službu typu PaaS (Platform as a Service), poskytovatel tedy zajišťuje infrastrukturu včetně operačního systému a běhového prostředí pro aplikaci. Služba podporuje využití různých technologií, například .NET, Java, Python, Node.js, Ruby či PHP.

Služba dále umožňuje automatické škálování, tedy přiřazení více výpočetních zdrojů pro danou aplikaci. Toto je řízeno pomocí App Service Planu, jedná se o službu, kterou musí mít každá instance App Service přiřazenou. App Service Plan udává operační systém, počet instancí a hardwarové parametry využívaného virtuálního stroje.

V App Services lze také upravovat konfigurace hostované aplikace, jedná se o záznamy typu klíč-hodnota. Pomocí těchto konfigurací lze ovlivňovat chování aplikace bez nutnosti zásahu do zdrojového kódu, takto může být nastaven kupříkladu připojovací řetězec do databáze.

Nasazení aplikace do této služby lze provést několika způsoby. V případě vývojového prostředí Visual Studio existuje integrace s platformou Azure a nasazení lze provést přímo v prostředí. Podobné to je i u editoru kódu Visual Studio Code, které však vyžaduje instalaci rozšíření. Nasazení lze provést i pomocí příkazů z knihovny Azure CLI nebo ve webovém rozhraní Azure Portal. [37] [14]

Azure SQL Database

Služba Azure SQL Database umožňuje vyžívat Microsoft SQL Server v cloudovém prostředí. Microsoft SQL Server je relační databázový systém, data jsou tedy ukládána v tabulkách, které jsou tvořeny sloupci s pevným datovým typem. Při komunikaci s Microsoft SQL Serverem je používán dotazovací jazyk T-SQL (Transact Structured Query Language), který rozšiřuje standardizovaný jazyk SQL.

Azure SQL Database však není srovnatelná s použitím Microsoft SQL Serveru na vlastním či hostovaným stroji. Služba nabízí přístup pouze k uživatelem vytvořené databázi, neumožňuje vytvoření více databází či přístup k systémovým databázím. Pokud je potřeba spravovat celou instanci SQL Serveru, pak lze využít službu Azure SQL Managed Instance.

Stejně jako u App Services lze službu škálovat dle potřeby, zde se jedná především o alokované místo na disku. Také je možné měnit typ ukládání záloh. Základní variantou je LRS (Locally Redundant Storage) vytvoření tří záloh v rámci stejného datacentra. Příkladem bezpečnější varianty zálohování je GRS (Geo Redundant Storage), kdy existují tři zálohy v rámci stejného datacentra a poté další tři zálohy v datacentru v jiném regionu. [39]

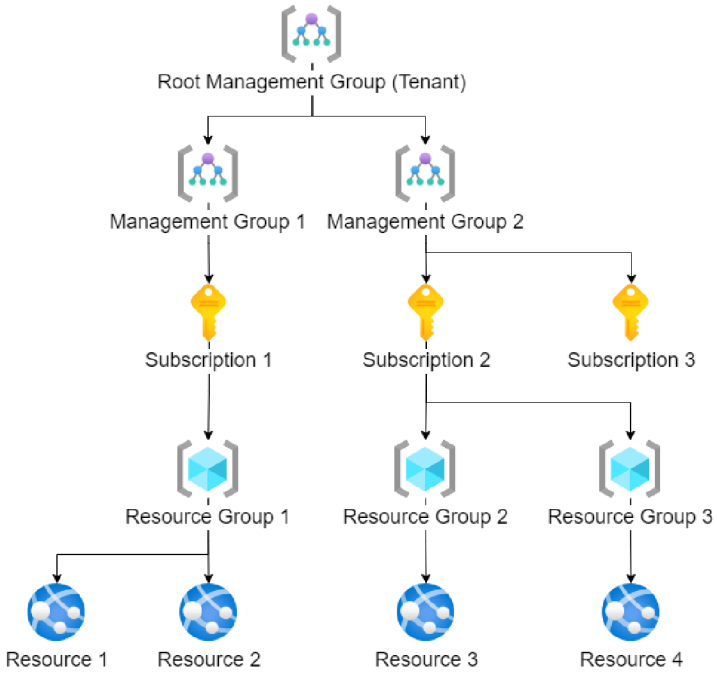
Key Vault

Služba Key Vault slouží jako uložiště pro tajemství, mohou to být šifrovací klíče, přihlašovací údaje či připojovací řetězce. Umožňuje vlastníkovi zdroje pokročilá nastavení přístupu k jednotlivým tajemstvím a také monitoring využívání služby. Key Vault má sloužit jako centrální uložiště pro zmíněná tajemství, má tedy být tzv. jediným zdrojem pravdy. Ostatní zdroje pak k uloženým hodnotám mohou přistupovat pomocí AAD autentizace či pomocí spravovaných identit. [37]

3.9.2 Organizace zdrojů

Každý zdroj musí mít příslušnost k určité skupině zdrojů (resource group). Tyto skupiny zlepšují přehlednost, ale také zajišťují sdílení životního cyklu zdrojů. Skupina zdrojů může představovat kompletní softwarové řešení, tedy samotnou webovou aplikaci, databázi a pomocné služby. V případě, že je skupina smazána, jsou smazány i související zdroje.

Dalším prvkem, který pomáhá organizaci zdrojů v Azure, je subskripce (subscription). Každá skupina zdrojů musí mít tuto subskripci přiřazenou. Jejím hlavním cílem je však zpoplatnění jednotlivých zdrojů. V případě, že organizace (tenant) využívá více subskripcí, lze využít další strukturální prvek a tím je skupina pro správu (management group). Pomocí těchto skupin je řízen přístup uživatelů k jednotlivým subskripcím. Příklad organizace zdrojů v Microsoft Azure je znázorněn na obrázku 7. [40]



Obrázek 7 – Organizace zdrojů v Microsoft Azure, vlastní tvorba dle zdroje: [40]

3.9.3 Správa platformy a zdrojů

Platforma Microsoft Azure nabízí více způsobů, jak s ní lze pracovat. Práce s platformou většinou spočívá ve správě dostupných zdrojů, subskripcí či různých nastavení.

První variantou je webová aplikace Azure Portal, která poskytuje intuitivní rozhraní a je tedy vhodná pro běžné uživatele. V aplikaci jsou k dispozici různé přehledy informující o využívání zdrojů a souvisejících nákladech. [41]

Azure CLI umožňuje pracovat s platformou Azure pomocí knihovny příkazů, které lze využívat v různých interpretech, kupříkladu PowerShell či bash. Výhoda využití Azure CLI spočívá především v možnosti automatizace pomocí skriptování. To je vhodné pro administrátory, kteří opakovaně provádí stejné či podobné úkony.

Další možností je využít Azure Cloud Shell. Jedná se o rozšíření Azure Portal v podobě příkazového řádku, který je vybaven příkazy ze zmíněného Azure CLI. Mimo to má také integrovaný textový editor a další nástroje. [42]

Azure SDK je knihovna, která je dostupná pro vývojáře a umožňuje jim pracovat s platformou Azure pomocí některého z podporovaných programovacích jazyků. SDK je k dispozici například pro .NET, JavaScript, Javu či Python. Azure SDK zapouzdřuje volání na vystavené API a je ve skutečnosti využíváno i samotným Azure CLI. [43]

Služba ARM (Azure Resource Manager) je webové API, které umožňuje správu zdrojů. Toto webové API je využíváno všemi výše uvedenými nástroji buď přímo, nebo prostřednictvím Azure SDK. Se službou ARM souvisí také ARM šablony. Jedná se o možnost odeslání souboru ve formátu JSON na vystavené API. Tato šablona definuje, jak má vypadat cílová skupina zdrojů a služba ARM požadovaný stav zajistí. K ARM šablonám byla nově uvedena alternativa v podobě jazyka Bicep, který má jednodušší syntaxi a umožňuje použití cyklů či podmínek. Pro tento koncept vytváření zdrojů pomocí kódu se používá pojem IaC (Infrastructure as Code). [44]

3.10 .NET

.NET je platforma pro softwarové vývojáře od firmy Microsoft, která je tvořena programovacími jazyky, běhovým prostředím, knihovnami tříd, balíčkovacím systémem NuGet a jinými nástroji. Platformu lze využívat v podobě několika implementací, v současné době je primární implementací .NET Core, nazývá se však jen jako .NET s číslem verze. .NET Core lze označit za nástupce implementace .NET Framework, která má stále zajištěnou podporu, jelikož ji využívá mnoho aplikací, avšak již není rozšiřována o nové funkce. Na rozdíl od svého předchůdce je .NET Core multiplatformní a vyvíjen jako open-source. Existují i další implementace, například Mono, každá z nich má své využití a je tvořena různými frameworky. [45] [46]

Při vývoji pomocí jakékoliv implementace je využívána stejná struktura. Zdrojový kód musí být součástí tzv. projektu (project), který může mít závislosti na jiných projektech či NuGet balíčcích. Obvykle z jednoho projektu vzniká jedna sestava (assembly), tou může být například dynamicky linkovaná knihovna či spustitelný soubor. Projekty jsou následně organizovány do řešení (solution). [46]

Nejznámějším zástupcem programovacích jazyků platformy .NET je C#. Jedná se o silně typovaný objektově orientovaný programovací jazyk s automatickou správou paměti. Dále podporuje zpracování výjimek, lambda výrazy, reflexi, asynchronní operace či využívání LINQ. Jelikož jde o objektově orientovaný jazyk, tak je všechen zdrojový kód organizován do tříd, které jsou vždy součástí jednoho jmenného prostoru (namespace). Dalšími jazyky platformy .NET jsou Visual Basic a F#. [45] [46]

3.10.1 Asynchronní volání

Předávání zpráv mezi objekty může probíhat synchronně či asynchronně. Synchronní volání je výchozí způsob předávání zpráv ve většině programovacích jazycích a pro volající objekt znamená, že je jeho činnost pozastavena, dokud není volaná metoda dokončena. V případě aplikace s uživatelským rozhraním to znamená, že pokud bude synchronně volat externí službu, tak aplikace zamrzne, dokud neobdrží odpověď, jelikož je dané vlákno blokováno voláním dané služby. Naopak asynchronní volání umožní objektu po zavolání metody provádět další operace nebo uvolnit vlákno pro obsloužení uživatelské rozhraní. [46]

V C# je v současnosti asynchronní volání realizováno pomocí konstruktů *async* a *await*. Metoda volající asynchronním způsobem musí v hlavičce obsahovat klíčové slovo *async*. Klíčové slovo *await* uvnitř metody představuje čekání na dokončení asynchronní operace. Dále je nutné, aby metoda měla návratový generický typ *Task*. [2] [46]

3.10.2 LINQ

LINQ (Language Integrated Query) je dotazovací jazyk integrovaný do jazyka C# a slouží k dotazování různých zdrojů dat. Se kterými daty lze pomocí LINQ pracovat, je určeno využívaným poskytovatelem (provider). Jedním z poskytovatelů zabudovaných přímo do C# je LINQ-to-Objects, který umožňuje dotazování nad kolekcí implementující rozhraní *IEnumerable*. Dále také existuje LINQ-to-SQL, který dané dotazy převádí do jazyka SQL. Při využívání LINQ jsou k dispozici dva typy zápisu. Prvním je syntaxe podobná SQL, příkladem je zdrojový kód 1. Druhým způsobem je dotazování pomocí metod s lambda výrazy (viz zdrojový kód 2). Výhodou využití LINQ je, že se programátor nemusí učit syntaxi dalšího jazyka. [46] [47]

```
var scoreQuery = from score in scores where score > 80 select score;
```

Zdrojový kód 1 – Zápis LINQ s využitím query syntaxe, dle zdroje [47]

```
var scoreQuery = scores.Where(s => s > 80);
```

Zdrojový kód 2 – Zápis LINQ s využitím metod, dle zdroje [47]

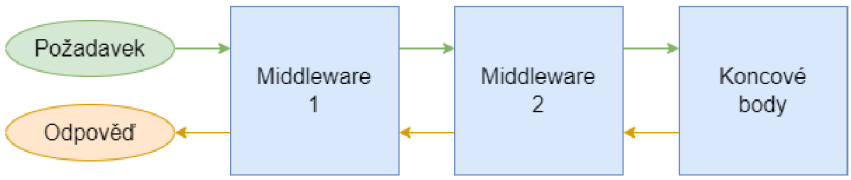
3.10.3 ASP.NET

ASP.NET Core (zkráceně jen ASP.NET) je framework pro tvorbu webových aplikací, který je součástí implementace .NET Core. Lze pomocí něj vytvořit REST API, RPC službu nebo vícestránkovou webovou aplikaci. Do frameworku je již zabudovaný DI kontejner, systém správy konfigurací, služby pro lokalizaci a pro žurnálování nebo také nástroj pro automatické vytváření dokumentace webového API. [31] [46]

Zpracování požadavků

V ASP.NET vstupují příchozí požadavky do tzv. pipeline, kde jsou zpracovány pomocí middlewarů. Jakmile jsou zpracovány i posledním middlewarem, vracejí se opačným směrem ke klientovi. Každý middleware může na požadavek aplikovat vlastní logiku v obou směrech, zároveň má možnost buď předat požadavek dalšímu middlewaru, nebo jej vrátit zpět. Tento způsob zpracování implementuje návrhový vzor řetězec odpovědnosti, popsaný v kapitole 3.3.7. Poslední middleware mívá na starost volání příslušného koncového bodu. Zpracování pomocí middlewarů je zachyceno na obrázku 8. Koncové body jsou reprezentovány jako metody ve třídách zvaných kontrolery (controllers). [48]

K odbavení jednoho požadavku je v ASP.NET využíváno jedno vlákno. Je vhodné využívat asynchronní volání, kupříkladu při komunikaci s databází, jelikož je tím vlákno uvolněno pro odbavení dalších požadavků, následně je původnímu požadavku přiděleno jiné volné vlákno. Tímto způsobem je redukován počet vláken pro odbavení požadavků a tedy i zvýšena propustnost webového serveru. [46]



Obrázek 8 – Middlewary v ASP.NET, vlastní zpracování dle zdroje: [48]

Zabudovaný DI kontejner

Zabudovaný DI kontejner je konfigurován při vytváření webové aplikace, kdy jsou zaregistrovány všechny služby. Služby lze registrovat s různým typem životnosti. Při využití metody *AddTransient* je vytvořena nová instance služby při každém jejím vyžádání. *AddScoped* vytvoří jednu instanci, která je využívána v rámci zpracování jednoho

požadavku. Poslední možností je registrace metodou *AddSingleton*, která vytvoří jedinou instanci služby pro celý životní cyklus aplikace. [31]

Výběr typu životnosti závisí především na udržování vnitřního stavu služby, ovšem lze zohlednit i dopady na výkonnost. Při registraci jiných než transientních služeb je nutné se vyvarovat závislostí na službách s kratší životností, jelikož bude jejich životnost prodloužena dle závislé služby. Například pokud je služba typu jedináček závislá na transientní službě, pak je instance transientní služby udržována v paměti po celou dobu životního cyklu aplikace. [31] [46]

Správce konfigurací

Chování ASP.NET aplikace lze ovlivnit pomocí konfigurací. Konfigurace je množina záznamů typu klíč-hodnota, které mohou pocházet z různých zdrojů dat. Konfigurační data jsou přístupná skrze registrovanou službu *IConfiguration*, případně lze zaregistrovat pouze část konfigurace jako vlastní službu. Nastavení zdrojů dat je prováděno při vytváření aplikace a využívají se tzv. poskytovatelé, jež umožňují získávat hodnoty z proměnných prostředí, argumentů příkazu, souborů či externích služeb (např. Azure Key Vault). Pořadí, ve kterém jsou zdroje konfigurace nastaveny, je důležité, jelikož později nastavené zdroje nahrazují již existující hodnoty. Pokud je tedy zdroj konfigurací z proměnných prostředí uveden jako poslední, pak mají hodnoty největší prioritu. [49]

3.10.4 Entity Framework

Entity Framework je ORM (Object Relation Mapping) nástroj pro .NET Core. Umožňuje pracovat s databázovým systémem pomocí konstruktů objektově orientovaného programování místo využívaného jazyka, kterým může být například SQL. Entity Framework komunikuje s různými typy databázových systémů pomocí poskytovatelů (provider), kupříkladu SQL Server, SQLite nebo PostreSQL. Vytváří tedy abstrakci mezi aplikací a databází. Při využití vkládání závislostí nabízí snadnou změnu databázového systému. [50]

Entity Framework pro práci s databázovým systémem využívá model, který je tvořen entitními třídami a databázovým kontextem. Pro vytváření tohoto modelu jsou využívány dva přístupy: code first, database first. Code first spočívá ve vytvoření entitních tříd včetně vazeb a atributů. Následně jsou pomocí entitních tříd definovány tabulky v databázovém kontextu. Entity Framework na základě těchto definic zajistí, aby cílová databáze obsahovala

požadované tabulky. Přístup database first předpokládá již existující databázi včetně tabulek a na základě těch vygeneruje entitní třídy s databázovým kontextem. [31]

Změny modelu jsou v Entity Frameworku realizovány pomocí migrací. Migrace představuje změny, které je nutné provést, aby byl model ve stejném stavu jako databáze. V migraci jsou rovněž definovány změny nutné pro návrat do původního stavu. [31]

Programátor pracuje s databází pomocí instance třídy databázového kontextu, ve kterém jsou tabulky reprezentovány kolekcemi entitních tříd. Pro dotazování nad těmito kolekcemi lze využít zmíněný LINQ v obou syntaxích. Změny hodnot atributů záznamu lze provést jednoduchou změnou vlastností objektu a následným potvrzením pomocí metody *SaveChangesAsync*. Tato metoda slouží k potvrzení transakce, tudíž může provést i více změnových operací a je zajištěno převedení databáze do konzistentního stavu. [46]

Téma .NET je dále rozvedeno v bakalářské práci *Aplikace pro vyhledávání a nabízení prací pro studenty* [1], kterou tato práce rozšiřuje.

3.11 Vue.js

Vue.js je open-source frontendový framework, který vývojáři umožňuje rychlejší a snazší tvorbu jednostránkových webových aplikací. Vue.js skrze komponentový model podporuje znovupoužitelnost a lepší udržitelnost kódu. Celá aplikace je tedy vyjádřena stromem instancí komponent. Komponenty jsou tvořeny HTML kódem, který může být doplněn i kaskádovými styly. Dále má každá komponenta vnitřní stav, který lze libovolně měnit a také zobrazovat v HTML pomocí speciálních atributů. Důležitou částí Vue.js je reaktivní systém, který zajišťuje, aby se změny stavu komponent projevily i v závislých HTML prvcích. [51]

Vue.js lze doplnit dalšími pluginy. Často využívaným je Vue Router, který umožňuje renderování komponent dle URL. Dále existuje plugin Vuex, jedná se o nástroj pro správu stavu (state management). Vuex je vhodné využít ve chvíli, kdy je nutné mít stav společný pro více komponent. Navíc umožňuje nastavit přesný způsob, jakým bude možné tento stav měnit pomocí tzv. mutací. [51] [52]

Téma Vue.js je dále rozvedeno v bakalářské práci *Aplikace pro vyhledávání a nabízení prací pro studenty* [1], kterou tato práce rozšiřuje.

4 Vlastní práce

Praktická část práce je tvořena návrhem architektury webové aplikace a následnou implementací v podobě šablony pro ASP.NET aplikace. Dále zahrnuje využití architektury v již existující aplikaci pro sjednávání prací pro studenty, která vznikla v rámci bakalářské práce *Aplikace pro vyhledávání a nabízení prací pro studenty* [1]. Původní aplikace bude rozšířena novými funkcemi a nasazena do cloudového prostředí. Tím budou ověřeny možné nedostatky navržené architektury a implementované šablony z pohledu vývojářské zkušenosti. K dosažení těchto cílů budou využity teoretické základy z předchozích kapitol a další znalosti, kterých autor nabyl během studia.

4.1 Architektura webové aplikace

Prvním krokem návrhu architektury je analýza, v rámci které budou autorem stanoveny požadavky na danou architekturu. Dle těchto požadavků bude vybrán vhodný typ architektury, budou vymezeny její komponenty a odpovědnosti komponent. Po dokončení návrhu bude architektura implementována jako .NET řešení, které bude možné využít jako šablonu pro nové ASP.NET projekty.

4.1.1 Požadavky

Vytvořená architektura by měla být nezávislá na využívaných technologiích, jedná se především o databázové systémy a jiné služby pro persistenci či integraci. Nezávislá by však měla být i na způsobu prezentace. Zmíněné nezávislosti zajišťují snadné změny využívaných technologií a snížení rizika vzniku chyb při změnách. Dalším požadavkem je snadná testovatelnost. Například, aby bylo možné testovat pravidla aplikační logiky bez datové perzistence a bez prezentace. Architektura by neměla být příliš komplexní, měla by sloužit především středně velkým aplikacím a malým vývojovým týmům.

Součástí implementace architektury mají být také mechanismy zajišťující cachování, ošetření chyb, lokalizaci, validaci a autorizaci. Tyto mechanismy musí být zcela nezávislé na konkrétním využití architektury.

4.1.2 Návrh

Z architektur popsaných v kapitole 3.7 splňují většinu požadavků mikroservisová architektura a clean architecture (či její variace). Důvodem nevyužití mikroservisové architektury je především komplexita řešení, která může být pro individuální vývojáře či malé týmy nezvládnutelná.

Clean architecture vyhovuje i požadavkům na komplexitu, avšak přichází s ní jiné problémy. Architektura neposkytuje řešení pro prvky, které jsou využívány všemi vrstvami, například guard metody. Dále specifikuje vrstvu domény aplikace, která vychází z návrhu řízeného doménou (Domain Driven Design), který je vhodný spíše pro velké projekty. Dalším nedostatkem je, že není přesně stanoveno, kam umístit kontrakty (rozhraní) přístupu k datům. Správně by se měly nacházet mezi vrstvou aplikace, která obsahuje aplikační logiku, a vrstvou domény, která obsahuje entitní třídy, agregáty a hodnotové objekty. Nová architektura bude tedy vycházet ze základů clean architecture, jež patří mezi monolitické architektury a je založena na obrácení závislostí.

Vrstvy architektury

Prezentační vrstva zůstává v nové architektuře stejná jako v clean architecture. Obsahuje třídy související pouze s konkrétním způsobem využití jádra aplikace. V této práci je architektura navržena pro webové aplikace, ovšem prezentační vrstvou může být i jednoduchá konzolová aplikace, aniž by musely být změněny ostatní vrstvy.

Vrstva aplikační logiky představuje vrstvu aplikace ve výchozí architektuře. Tato část obsahuje třídy, jejichž metody implementují aplikační logiku a související pravidla. Do vrstvy aplikační logiky patří také definice pro DTO (Data Transfer Object), které odstraňují závislost prezentační vrstvy na struktuře dat v nižších vrstvách.

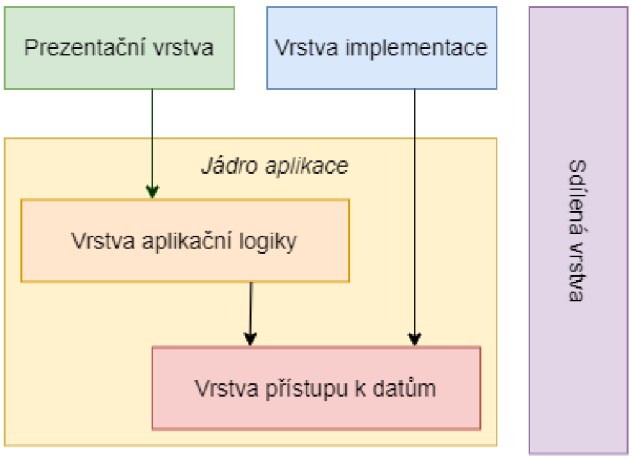
Vrstva domény je nahrazena vrstvou přístupu k datům. Již tedy není vycházeno z doménového návrhu, místo toho je vše budováno dle abstraktního přístupu k datům. Tento pohled může být pro vývojáře více intuitivní, jelikož je pro ně přístup k datům snáze uchopitelný než samotná doména aplikace. Vrstva přístupu k datům tedy obsahuje třídy entit, které jsou aplikací využívány. Do přístupu k datům však patří i rozhraní repozitářů, slovníky pro lokalizaci, rozhraní pro cachování či jiné služby. Spolu s vrstvou aplikační logiky tvoří vrstva přístupu k datům tzv. jádro aplikace.

Původní vrstva infrastruktury obsahuje konkrétní implementace pro rozhraní související s využíváním externích systémů, služeb a nástrojů. Role této vrstvy zůstává

v nové architektuře stejná. Vrstva však bude nově označována jako vrstva implementace, jelikož může obsahovat i implementace jiných prvků než těch infrastrukturních.

Vrstva, kterou nově navržená architektura přidává, se nazývá sdílená vrstva. Jedná se o speciální vrstvu, která prostupuje všemi ostatními. Patří sem všechny třídy, které jsou využívány ve více vrstvách. Je vhodné sem umístit i komponenty, které jsou využívány pouze jednou vrstvou, avšak jejich odpovědnost není s danou vrstvou nijak svázaná. Ve sdílené vrstvě tedy budou různé pomocné třídy. Důležité však je, aby sdílené prvky měly obecnou povahu. Jinými slovy, sdílená vrstva netuší, jaký je účel aplikace. Opět je vhodné dodržovat princip obrácení závislostí a konkrétní implementace umístit do vrstvy implementace.

Po vyčlenění vrstev je nutné určit závislosti mezi nimi. Obecně v architektuře platí, že závislosti musí být jednosměrné. Závislosti mezi vrstvami jsou zachyceny pomocí plných čar na obrázku 9. Výjimkou je sdílená vrstva, jelikož na té mohou záviset všechny ostatní vrstvy a zachycení této skutečnosti by zhoršilo přehlednost diagramu.

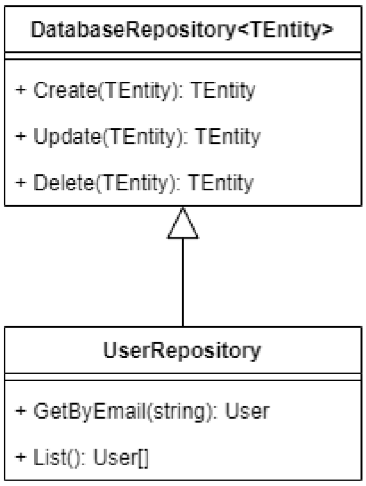


Obrázek 9 – Návrh vrstev nové architektury

Persistence dat

Nejdůležitějším článkem vrstvy přístupu k datům bývá persistence dat. Nabízí se zde využití návrhového vzoru repozitáře, který umožňuje zapouzdřit operace s daty. Repozitáře navíc poskytují přehled operací, které jsou nad daným zdrojem dat prováděny. Podle principu jedné odpovědnosti není správné vybudovat jeden univerzální repozitář, kde budou metody s dalším vývojem narůstat a třída přestává být přehledná. Řešením je definovat repozitáře podle entitních tříd. Jakmile existuje více repozitářů, může nastat, že se některé metody budou opakovat, typickým příkladem je úprava či smazání záznamu. Duplicitě kódu lze předejít vyčleněním nadtřídy, kterou je tzv. generický repozitář.

Generický repozitář je generická abstraktní třída, která provede danou operaci pro jakýkoliv typ entity. Obecné repozitáře někdy zahrnují i často využívané čtecí metody, například *GetById* či *List*, ovšem to může vést k tomu, že klienti jsou zbytečně závislí na částech rozhraní, které vůbec nevyžadují. Objektový model generického a konkrétního repozitáře je na obrázku 10.



Obrázek 10 – Objektový model repozitáře

Cachování

Dalším mechanismem, který má architektura dle požadavků poskytovat, je cachování. Cachování spočívá v ukládání výsledků čtecích operací nad zdrojem dat do paměti, aby byly k dispozici rychleji než při jejich opakovaném čtení. Cachovací služby bývají implementovány jako seznamy datových typů klíč-hodnota, kde hodnotou je uložený výsledek a klíčem je cachovací klíč. V případě čtení dat je nejprve ověřeno, zda se výsledek již nachází v cache a pokud ne, tak je získán z daného zdroje dat a následně uložen do cache. Při generování cachovacího klíče se musí zajistit, aby byl pro stejný dotaz pokaždé vytvořen stejný cachovací klíč. Tuto odpovědnost by bylo možné svěřit jednotlivým metodám v repozitáři, ovšem znamenalo by to opakování velmi podobné operace. Vhodnější je, aby byl cachovací klíč sestaven automaticky dle metody repozitáře, která byla volána a dle jejích parametrů.

Při cachování nesmí být opomenuto nebezpečí nekonzistence dat. Vzniká tehdy, když stav dat uložených v paměti neodpovídá stavu dle samotného zdroje. Toto riziko lze do jisté míry ošetřit uvedením životnosti dat v mezipaměti. Pak ovšem mohou být označena jako nevalidní i data, jejichž stav odpovídá realitě. Další možností je odstranit odpovídající položky z mezipaměti v případě změnové operace, ovšem to by vyžadovalo zjištění všech

cachovacích klíčů, které obsahují modifikovaná data. Lze to však zjednodušit rozdělením cache dle typů entit a v případě změnové operace vymazat vše, ale pouze z cache daného typu entity. Tento postup může být také doplněn o uvedení životnosti pro případ, že by zdrojová data byla změněna z jiného systému.

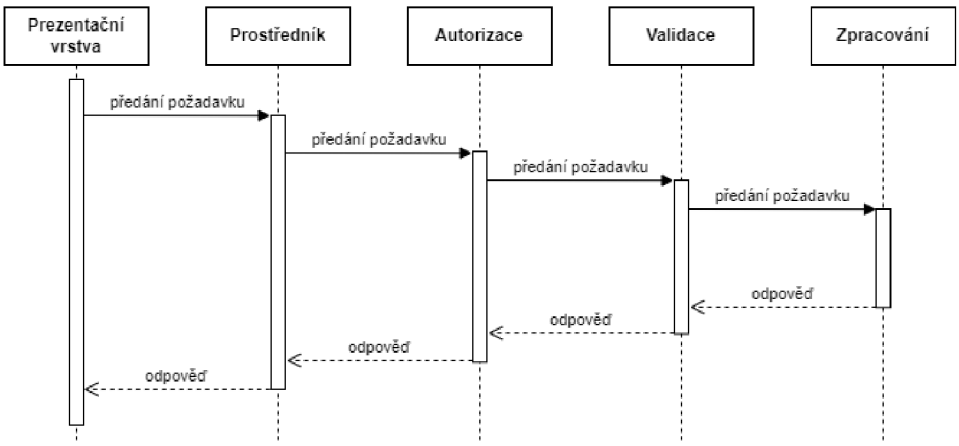
Důležité také je, aby cachování bylo volitelné, jelikož v některých případech je nežádoucí, aby výsledky pocházely z cache. Kupříkladu u přihlášení by hrozilo, že budou vyžadovány přihlašovací údaje, které již nejsou aktuální. Toto může být řešeno způsobem, že je ke standardní implementaci repozitáře vytvořen i cachovaný repozitář pomocí návrhového vzoru dekorátor. To ovšem způsobí, že pro každou novou metodu v repozitáři je nutné rovněž vytvořit metodu v dekorátoru. Jednodušší je, aby byla standardní implementace repozitáře parametrizována, zda má být využita cache či nikoliv.

Validace

Validační mechanismy jsou již integrovány ve frameworku ASP.NET, kde jsou využívány v podobě atributů a všechny příchozí požadavky jsou podle těchto pravidel automaticky kontrolovány. Tímto způsobem lze validaci požadavků snadno a přehledně implementovat. Nevýhodou však je, že se validační mechanismus sváže s prezentační vrstvou a v případě využití jiné prezentační vrstvy je nutné validace znovu implementovat. Při přenosu validačního mechanismu do vrstvy aplikační logiky je nutné zajistit automatickou kontrolu validačních pravidel, aby nemusela být volána explicitně v každé metodě. Zde lze využít kombinace návrhových vzorů prostředník a řetězec odpovědnosti. Princip fungování tedy je, že příchozí požadavky jsou předány prostředníkovi ve vrstvě aplikační logiky a před zpracováním je automaticky spuštěn validační proces.

Autorizace

Velmi podobná situace jako u validace je i u autorizačního mechanismu. Opět lze využít implementaci zabudovanou v ASP.NET, ovšem zůstává stejná nevýhoda jako v případě validace. Řešením je tedy zmíněná kombinace prostředníka a řetězce odpovědnosti, kdy před zpracováním požadavku je vedle validace provedena i autorizace. Sekvenční model zpracování požadavků je zachycen na obrázku 11.



Obrázek 11 – Sekvenční model zpracování požadavků ve vrstvě aplikační logiky

Ošetření chyb

V případě, že se při zpracovávání v aplikaci objeví chyba, je nutné na ni zareagovat. Tyto chyby mohou být očekávané (např. související s aplikační logikou) nebo neočekávané (např. nedostupná externí služba). Způsob, jakým aplikace na chybu zareaguje, ovlivňuje prezentační vrstva, jelikož webová aplikace reaguje jinak než desktopová aplikace. Je tedy nutné informaci o chybě dostat od zdroje chyby až k prezentační vrstvě. Jeden způsob je postupným předáváním této informace přes standardní řídící tok. To však vyžaduje upravit všechny návratové typy tak, aby umožňovaly předávat informaci o chybě. Jednodušším způsobem je však zachovat původní návratové typy a informaci o chybě předávat skrze výjimky, tedy tok výjimek (exception flow), přičemž prezentační vrstva tuto výjimku zachytí a patřičně na ni zareaguje.

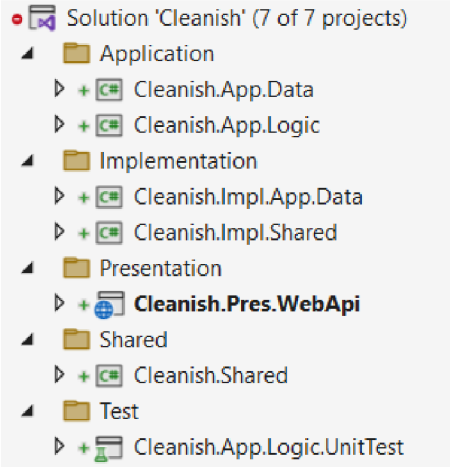
4.1.3 Implementace

Architektura bude dle uvedených požadavků a návrhu implementována jako šablona pro ASP.NET. Šablona umožní při zahájení nového projektu vygenerovat zdrojový kód společný pro jakékoliv řešení, tzv. boilerplate. Mimo to bude obsahovat logiku jednoduché aplikace pro správu úkolů, aby uživatel šablony pochopil, jak s architekturou pracovat a jak implementovat vlastní logiku. Šablona ponese název „Cleanish architecture“, aby vyjadřovala skutečnost, že vychází z clean architecture.

Struktura .NET řešení

Strukturou řešení je myšleno rozdělení na projekty, které je dáno vrstvami architektury. Výjimkou je ovšem vrstva implementace, která je rozdělena na dva projekty

podle toho, pro kterou vrstvu poskytuje konkrétní implementace. Prezentační vrstva je projekt typu ASP.NET Core Web API a ostatní projekty jsou typu Class library. Součástí řešení jsou také projekty obsahující jednotkové testy. Výsledná struktura je znázorněna na obrázku 12. Aby nebyly názvy příliš dlouhé, jsou některá slova zkrácena, například „Impl“ místo „Implementation“.



Obrázek 12 – Struktura .NET řešení šablony

Registrace závislostí a konfigurace

Vzhledem k tomu, že architektura je založena na obrácení závislostí, je využit vzor vkládání závislostí. Vytvoření a naplnění DI kontejneru je v odpovědnosti prezentační vrstvy, která také může registrovat své vlastní služby. Ostatní vrstvy však musí definovat, jaké služby mají být registrovány.

K tomu slouží soubory *DependencyInjection.cs*, ve kterých se nachází rozšiřující metoda pro rozhraní *IServiceCollection*. Jedná se o obecné rozhraní v .NET, které je využito DI kontejnerem zabudovaným v ASP.NET Core, ale také externími implementacemi. Závislost na tomto rozhraní tedy nevadí. Zmíněnou rozšiřující metodu obsahují všechny projekty, které poskytují konkrétní implementace, jedná se tedy o *Pres.WebApi*, *App.Logic*, *Impl.App.Data*, *Impl.Shared*.

Ve zdrojovém kódu 3 je uvedena část metody pro registraci služeb z vrstvy přístupu k datům, která registruje repozitáře a databázový kontext pro Entity Framework. Lze si všimnout, že metoda *AddApplicationData* vyžaduje argument typu *IConfiguration*. Opět se jedná o obecné rozhraní v .NET a slouží k poskytování hodnot konfigurace, v tomto případě umožňuje předat údaje pro připojení k databázi.

```
public static IServiceCollection AddApplicationData(
    this IServiceCollection services,
    IConfiguration configuration)
{
    services.AddDbContext<SqlDbContext>(opt =>
        opt.UseSqlServer(configuration.GetConnectionString("Sql")));
    services.AddDatabaseRepository<User, IUserRepository, UserRepository>();

    return services;
}
```

Zdrojový kód 3 – Metoda pro registraci služeb vrstvy přístupu k datům

Jak již bylo zmíněno, samotné vytvoření DI kontejneru a registraci služeb provádí prezentační vrstva. Ovšem prezentační vrstva zajišťuje i připojení konfiguračních souborů pro službu *IConfiguration*. Prezentační vrstvou je v této šabloně ASP.NET Core aplikace, jejíž vstupním bodem je třída *Program* a právě zde jsou tyto operace provedeny. Zdrojový kód 4 zachycuje registraci služeb ze všech vrstev a také připojení konfiguračních JSON souborů. Zdrojů konfigurací lze připojit více, ovšem je důležité pořadí, jelikož později připojené zdroje nahradí hodnoty klíčů, které již byly nastaveny. Z tohoto důvodu je JSON soubor závisející na prostředí připojen až po standardním, je tím například umožněno využívat jinou databázi při vývoji než při produkčním spuštění.

```
var builder = WebApplication.CreateBuilder(args);

builder.Configuration
    .AddJsonFile("appsettings.json)
    .AddJsonFile($"appsettings.{env.EnvironmentName}.json);

builder.Services
    .AddShared(builder.Configuration)
    .AddApplicationData(builder.Configuration)
    .AddApplicationLogic(builder.Configuration)
    .AddWebApi(builder.Configuration);
```

Zdrojový kód 4 – Registrace služeb všech vrstev a nastavení zdrojů konfigurace

Poskytování času

V případě, že bude nutné kdekoliv ve zdrojovém kódu získat aktuální čas, bude k dispozici služba pro poskytování času. Rozhraní této služby je k dispozici ve zdrojovém kódu 5. Standardní implementace této služby zajišťuje, aby metoda *Now* vracela aktuální čas dle UTC. Důvodem, proč získávat informace o čase pomocí této služby místo přímého volání metody na třídě *DateTime*, je především snadnější testování. V případě, že bude vyžadováno

mít pro potřeby testování jiný aktuální čas, než je systémový, tak stačí pouze poskytnout jinou implementaci dané služby. Dále zajišťuje konzistenci ve zdrojovém kódu, jelikož třída *DateTime* vystavuje metodu pro získání aktuální času dle lokálního časového pásma a dle UTC.

Tato služba muže být využita v jakékoliv vrstvě a není závislá na účelu aplikace, patří tedy do sdílené vrstvy. Rozhraní bude v projektu *Shared* a standardní implementace v projektu *Impl.Shared*.

```
public interface IClockProvider
{
    DateTime Now { get; }
}
```

Zdrojový kód 5 – Rozhraní pro službu poskytování času IClockProvider

Persistence dat a cachování

Základem persistence jsou entitní třídy, ty reprezentují strukturu dat a vztahy, které jsou ukládány. Tyto třídy patří do projektu *App.Data*, kde jsou zanořeny do adresářové struktury *Database/Entities*. Mimo entitní třídy sem patří i související struktury *enum*. Dále je zavedena společná abstraktní třída *BaseEntity* (viz zdrojový kód 6), která je vhodná pro společnou definici identifikátorů a údajů o čase vytvoření či změny. Abstraktní třída *BaseEntity* je také užitečná jako omezení generických tříd, jak lze vidět ve zdrojovém kódu 7.

```
public abstract class BaseEntity
{
    public Guid Id { get; set; }
    public DateTime Created { get; set; }
    public DateTime Updated { get; set; }
}
```

Zdrojový kód 6 – Abstraktní entitní třída BaseEntity

Dalším článkem persistence dat jsou repozitáře, které již byly popsány v kapitole 3.3.4 a také v návrhu architektury. Ve vrstvě přístupu k datům se však nachází pouze rozhraní repozitářů, jelikož konkrétní implementace jsou závislé na využívané technologii pro persistenci. Dle návrhu vzniknou repozitáře pro entitní třídy a také generický repozitář *IDatabaseRepository* (viz zdrojový kód 7), ze kterého budou ostatní repozitáře dědit.

```
public interface IDatabaseRepository<T> where T : BaseEntity
{
    Task<T> CreateAsync(T entity);
    Task<T> DeleteAsync(T entity);
    Task<T> UpdateAsync(T entity);
}
```

Zdrojový kód 7 – Rozhraní generického repozitáře IDatabaseRepository

Nyní již lze vytvořit konkrétní implementace pro repozitáře v implementační vrstvě. Zde již vzniká závislost na využívané technologii. V rámci této budované šablony je využit Entity Framework, ovšem uživatelé šablony mohou tuto implementaci přizpůsobit svým potřebám. Generický repozitář komunikuje s databází pomocí třídy *SqlDbContext*, kterou získá v konstruktoru skrze vkládání závislostí. Ve třídě databázového kontextu existuje generická metoda *Set<T>*, která vrací datový set relevantní pro daný typ entity. Dataset je poté využíván pro čtecí i zapisovací operace. Ve zdrojovém kódu 8 je uveden konstruktor generického repozitáře a implementace metody vytváření záznamu.

```
public DatabaseRepository(SqlDbContext dbContext)
{
    _dbContext = dbContext;
    _dbSet = _dbContext.Set<TEntity>();
}

public async Task<TEntity> CreateAsync(TEntity entity)
{
    TEntity createdEntity = _dbSet.Add(entity).Entity;
    await _dbContext.SaveChangesAsync();
    return createdEntity;
}
```

Zdrojový kód 8 – Implementace generického repozitáře DatabaseRepository

Nastavení entitních tříd pro Entity Framework lze provést dvěma způsoby. Prvním je využití atributů ve zdrojovém kódu, tedy přímo v entitní třídě. Tento postup je však nežádoucí, jelikož by vznikla závislost vrstvy datového přístup na Entity Frameworku. Vhodnější způsob je pomocí fluent API, kdy jsou konfigurace provedeny třídou databázového kontextu skrze reflexi. Aby bylo řešení přehlednější, má každá entitní třída svou vlastní konfigurační třídu (viz zdrojový kód 9), navíc existuje společný konfigurátor pro nadtřídu *BaseEntity*.

```
internal abstract class BaseEntityConfiguration<T>
    : IEntityTypeConfiguration<T> where T : BaseEntity
{
    public virtual void Configure(EntityTypeBuilder<T> builder)
    {
        builder.ToTable(typeof(T).Name);
        builder.HasKey(e => e.Id);
    }
}

internal class UserConfiguration : BaseEntityConfiguration<User>
{
    public override void Configure(EntityTypeBuilder<User> builder)
    {
        base.Configure(builder);
        builder.Property(i => i.Email).IsRequired();
        builder.HasIndex(i => i.Email).IsUnique();
    }
}
```

Zdrojový kód 9 – Konfigurace entitních tříd pro EntityFramework

Dále je potřeba implementovat čtecí operace pro konkrétní repozitáře. To je však zkomplikované požadavkem na cachování výsledků, které by dle návrhu mělo být automatické a volitelné. Aby bylo možné provést cachování automaticky, nemohou čtecí metody pracovat přímo s databázovým kontextem, musí být tedy vložen prvek mezi veřejnou metodu repozitáře a databázový kontext.

Toto lze realizovat vytvořením privátní metody *ReadDataAsync* v generickém repozitáři, které budou předány všechny parametry dotazu, dle kterých bude možné vytvořit cachovací klíč a případně provést čtecí operaci. Pro předání parametrů dotazu bude využita třída *Specification*, která obsahuje informace o řazení, filtrování, stránkování a spojování dat. Samotná specifikace však k provedení dotazu nestačí. Pro stanovení návratové hodnoty je nutné uvést operaci, například *CountAsync*, *SingleAsync* či *ToListAsync*. Výsledná podoba metody *ReadDataAsync* je ve zdrojovém kódu 10. Parametr *callerMethodName* je automaticky nastaven dle volající metody, tudíž není nutné název explicitně psát. Proměnná *valueProvider* obsahuje anonymní funkci pro získání dat dle specifikace a dle požadované operace. *SpecificationEvaluator* vytvoří LINQ dotaz pro daný dataset podle atributů předložené specifikace. Předtím než je sestaven cachovací klíč, je ověřeno, zda má být cache využita. Pokud ne, jsou data získána z datového zdroje.

Vytvoření cachovacího klíče spočívá ve spojení několika textových řetězců, kterými jsou: název typu entity, název volané metody repozitáře a seznam parametrů. Na výsledný řetězec je následně aplikována hashovací funkce MD5, která jeho délku podstatně zkrátí. Lze si všimnout, že v parametrech cachovacího klíče chybí atribut spojování dat. Je to z důvodu, že tento atribut má funkční závislost na kombinaci názvu typu entity a názvu volané metody, tím pádem jeho znalost neovlivní unikátnost cachovacího klíče. Problémem při vytváření cachovací klíče ze specifikace je, jak vytvořit textový řetězec z lambda funkcí. Algoritmus pro převedení libovolného výrazu na textový řetězec však zveřejnil Pete Montgomery ve svém blogu. [53] Mechanismus je tedy založen na jeho řešení, které je zapouzdřeno v metodě *ToEvaluatedString*.

```
private async Task<TResult> ReadDataAsync<TResult>(
    Specification<TEntity> specification,
    Func<IQueryable<TEntity>, Task<TResult>> queryOperation,
    [CallerMemberName] string callerMethodName = "")
{
    var valueProvider = () => queryOperation(
        SpecificationEvaluator.Evaluate(_dbSet, specification));

    if (CacheStrategy != CacheStrategy.Use) return await valueProvider();

    var cacheKey = new CacheKey(
        typeof(TEntity).Name,
        callerMethodName,
        specification.Filter?.ToEvaluatedString(),
        specification.OrderBy?.ToEvaluatedString(),
        specification.OrderByDesc?.ToEvaluatedString(),
        specification.Skip?.ToString(),
        specification.Take?.ToString()
    );

    return await _cacheService.UseCacheAsync(cacheKey, valueProvider);
}
```

Zdrojový kód 10 – Metoda ReadDataAsync pro cachované čtení dat

Ve zdrojovém kódu 11 je ukázka implementace metody repozitáře pro získání úkolu dle jeho identifikátoru. Pro vytváření instance třídy *Specification* je využit návrhový vzor stavitel, aby nebylo nutné vytvářet více přetížených konstruktorů.

```
public async Task<TodoItem> GetByIdAsync(Guid todoItemId)
{
    return await ReadDataAsync(
        Specification()
        .SetFilter(ti => ti.Id == todoItemId)
        .Build(),
        q => q.SingleOrDefaultAsync()
    );
}
```

Zdrojový kód 11 – Metoda GetById pro získání úkolu dle identifikátoru

Pro cachování je vytvořena služba *ICacheService*, jejíž standardní implementace využívá třídu *MemoryCache*, která je zabudována v ASP.NET. Pro využití cache je vystavena metoda *UseCache*, která vyžaduje v parametrech cachovací klíč a anonymní funkci pro získání dat z původního zdroje. Nejprve ověří, zda se pod cachovacím klíčem nachází platný výsledek. Pokud ano, tak jej vrátí. Pokud však výsledek není v cache, získá data pomocí anonymní funkce, výsledek uloží do cache a následně jej vrátí. Fungování cache, např. čas expirace záznamů, je ovlivněno konfigurací, která je přijímána jako *IOptions<CacheSettings>* v konstruktoru služby pro cachování.

Pro zajištění konzistence dat dle návrhu má být cache aktualizována při změnové operaci, ovšem mají být smazány pouze výsledky související s daným typem entity. *MemoryCache* je ale v DI kontejneru registrována jako jedináček, není tedy možné mít pro každý typ entity vlastní instanci *MemoryCache*.

Pro vyřazení záznamů z cache lze použít *CancellationToken*, který je předán spolu s ukládaným výsledkem. Cílem tedy je, aby měly všechny výsledky související s jedním typem entity přidružený stejný *CancellationToken*. Jakmile dojde ke změnové operaci, bude daný token zrušen a záznamy budou z cache odstraněny. *CacheService* musí být tedy převedena na generickou třídu a pro každý typ entity vznikne jedna instance, která bude mít uložený vlastní token. Důležité je, že tento token implementuje rozhraní *IDisposable*, tím pádem jej musí implementovat i *CacheService*, aby byly přidružené zdroje uvolněny, jakmile nejsou využívány. Zdrojový kód 12 obsahuje metodu, která zajistí invalidaci záznamů ve chvíli, kdy je v generickém repozitáři volána metoda změnové operace. Jakmile byla provedena invalidace, musí být zavolána metoda *Dispose* a následně vytvořen nový token.

```
public void Clear()
{
    if (_resetCacheToken != null)
    {
        if (!_resetCacheToken.IsCancellationRequested)
        {
            _resetCacheToken.Cancel();
        }
        _resetCacheToken.Dispose();
    }
    _resetCacheToken = new();
}
```

Zdrojový kód 12 – Metoda Clear pro invalidaci cachovaných záznamů

Posledním bodem persistence dat a cachování je registrace v DI kontejneru a umístění zdrojového kódu v řešení. U *CacheService* by se dalo argumentovat, že fungování není závislé na využití aplikace a mohlo by patřit do sdílené vrstvy. Ovšem cachování svou podstatou patří do přístupu k datům a zároveň na něm závisí pouze třídy z vrstvy datového přístupu, tudíž standardní implementace je v projektu *Impl.App.Data* v adresáři *Cache*. Při registraci repozitáře pro jeden typ entity musí být zaregistrovány tři služby. Jednou je generická *ICacheService* a dalšími dvěma službami jsou konkrétní repozitáře, ovšem pokaždé s jinou hodnotou atributu *CacheStrategy*, tedy jedna instance využije cache, a druhá nikoliv.

Aplikační logika

Pro zdrojový kód aplikační logiky a souvisejících prvků je určen projekt *App.Logic*. V návrhu architektury bylo stanoveno, že tato vrstva bude postavena na návrhových vzorech prostředník a řetězec odpovědnosti. NuGet balíček MediatR pomáhá tyto návrhové vzory implementovat. Lze jej zaregistrovat v DI kontejneru a následně využít službu *ISender*, která slouží k předání požadavků ke zpracování tzv. handlery. Požadavky představují data vstupující do aplikační logiky, např. údaje nově vytvářeného úkolu) a handlery definují, co má být s požadavkem provedeno, např. ověření maximálního počtu vytvořených úkolů, následnou delegaci repozitáři a vrácení výsledku.

Ve vrstvě aplikační logiky je nutné vytvořit třídy pro požadavky, které implementují rozhraní *IRequest* z balíčku MediatR, a také třídy handlerů implementující rozhraní *IRequestHandler*. Tyto třídy jsou organizovány v rámci projektu do adresářů dle zdroje a následně zanořeny v adresáři *UseCases*. Vzhledem k tomu, že požadavky jsou úzce

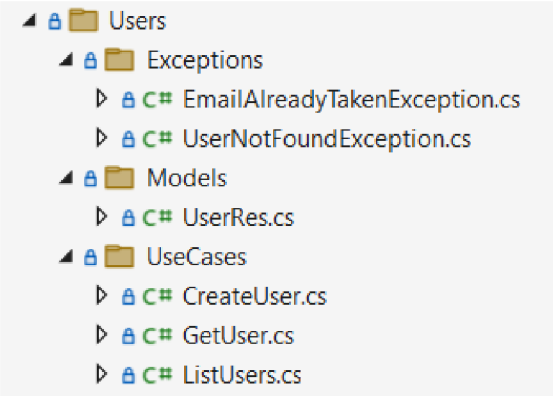
provázány s handlery, jsou uloženy ve společném souboru. MediatR při registraci objeví všechny vytvořené třídy požadavků a handlerů pomocí reflexe.

Pojem zdroj je v této kapitole nový, představuje předmět zájmu vytvářené aplikace. Množina zdrojů může odpovídat množině typů entit, ale nemusí tomu tak být. Například v databázi může existovat tabulka s adresami uživatelů, tudíž se jedná o typ entity, avšak nejedná se o zdroj, jelikož adresa samotná není předmětem zájmu, tím je pouze uživatel.

Zmíněný *IRequestHandler* a *IRequest* jsou generická rozhraní, která vyžadují uvést očekávaný návratový typ. Jako návratový typ jsou využity třídy DTO, které prezentační vrstvy odstiňují od podléhajícího datového modelu. Mohou zachovat původní podobu entitní třídy, ale také umožňují skrývat či přidávat atributy, nebo dokonce vytvářet zcela nové struktury. Třídy DTO jsou organizovány v adresáři *Models*, který je zanořen v adresáři příslušného zdroje. S využitím DTO se váže jeden problém a to, jak vytvářet instance z původních entitních tříd. Instance lze vytvářet manuálně a zdrojový kód umístit třeba do konstruktoru DTO. V případě, že je struktura DTO a entitní třídy podobná, tak se jedná o repetitivní činnost, kterou lze automatizovat pomocí NuGet balíčku Mapster. Mapster dokáže vyřešit přiřazení atributů se stejnými názvy a umožňuje mapování dále konfigurovat. Celý proces je proveden generickou metodou *Adapt* rozšiřující všechny objekty.

V návrhu vrstvy aplikační logiky je dále popsáno, že zpracování chyb bude realizováno pomocí toku výjimek místo standardního řídícího toku. Do chyb však patří i nevalidní situace v rámci aplikační logiky, například uživatel překročí maximální počet vytvořených úkolů. Pro každou takovou situaci je vytvořen nový typ výjimky, která je v případě výskytu zpracována prezentační vrstvou. Pro tento typ výjimek je ve sdílené vrstvě připravena nadtřída *BadLogicException*, která definuje vlastnost klíče zprávy pro možnost implementování lokalizace.

Výsledná adresářová struktura pro uživatele je na obrázku 13. Dále je ve zdrojovém kódu 13 uveden požadavek pro získání uživatele a příslušný handler. Handler ve svém konstruktoru vyžaduje cachovaný repozitář uživatelů skrze vkládání závislostí. Požadavek pro získání uživatele má jediný atribut, kterým je identifikátor uživatele, pokud není uživatel nalezen, handler vyhodí výjimku typu *UserNotFoundException*, která dědí z obecné výjimky *NotFoundException*. Nakonec je instance entitní třídy namapována na DTO a vrácena.



Obrázek 13 – Adresářová struktura pro zdroj uživatele

```
public class GetUserRequest : IRequest<UserRes>
{
    public Guid Id { get; set; }
}

internal class GetUserRequestHandler :
    IRequestHandler<GetUserRequest, UserRes>
{
    private readonly IUserRepository _userRepository;

    public GetUserRequestHandler(Cached<IUserRepository> userRepository)
    {
        _userRepository = userRepository.Value;
    }

    public async Task<UserRes> Handle(GetUserRequest request)
    {
        var user = await _userRepository.GetByIdAsync(request.Id);
        if (user == null) throw new UserNotFoundException();

        return user.Adapt<UserRes>();
    }
}
```

Zdrojový kód 13 – Požadavek a handler GetUser pro získání přihlášeného uživatele

Poskytování identity

Handlery ve vrstvě aplikační logiky mohou v některých případech vyžadovat znalost identity uživatele, aby mohly dokončit zpracování požadavku. Příkladem je vytvoření nového úkolu, kde je nutné uvést i autora vytvářeného úkolu. Tento mechanismus lze řešit explicitním předáním informace o identitě uživatele v požadavku, ovšem to způsobí značné znepřehlednění požadavků a také duplicitu zdrojového kódu.

Řešením je nová služba *ISecurityContextProvider*. Rozhraní této služby spadá do vrstvy aplikační logiky, ovšem implementace nikoliv, neboť závisí na způsobu, jakým prezentační vrstva udržuje kontext autentizace. Služba má vlastnost typu *SecurityContext* (viz zdrojový kód 14), která udržuje informace o přihlášeném uživateli. Pokud je hodnota této vlastnosti *null*, znamená to, že uživatel není ověřen.

```
public class SecurityContext
{
    public Guid SubjectId { get; init; }
    public UserRole Role { get; init; }
}
```

Zdrojový kód 14 – Třída SecurtiyContext pro poskytování identity

Implementace pro aplikaci ASP.NET je založena na získání informací o identitě ze služby *IHttpContextAccessor*, která je součástí frameworku a informace jsou nastaveny middlewarem *UseAuthentication*. Místo toho, aby vlastnost *SecurityContext* byla inicializována v konstruktoru služby, tak je využita opožděná inicializace. Tím je zajištěna optimalizace výkonu v případě, že i přes vyžádání služby, nedojde ke čtení informací o identitě. Celá implementace je ve zdrojovém kódu 15 a registrace služby probíhá v prezentační vrstvě. Metoda *BuildSecurityContext* zajišťuje parsování identifikátoru uživatele a uživatelské role, pokud parsování selže, je nastavena hodnota *null*.

```
internal class SecurityContextProvider : ISecurityContextProvider
{
    private readonly IHttpContextAccessor _httpContextAccessor;
    private readonly Lazy<SecurityContext> _securityContext;
    public SecurityContext SecurityContext => _securityContext.Value;

    public SecurityContextProvider(IHttpContextAccessor httpContextAccessor)
    {
        _httpContextAccessor = httpContextAccessor;
        _securityContext = new Lazy<SecurityContext>(BuildSecurityContext);
    }

    private SecurityContext BuildSecurityContext()
    {
        var claims = _httpContextAccessor.HttpContext?.User?.Claims;
        if (claims == null || !claims.Any()) return null;

        var subClaim = claims.FirstOrDefault(c => c.Type == "sub")?.Value;
        var roleClaim = claims.FirstOrDefault(c => c.Type == "role")?.Value;
```

```
        if (string.IsNullOrWhiteSpace(subClaim) ||
            !Guid.TryParse(subClaim, out Guid subjectId) ||
            string.IsNullOrWhiteSpace(roleClaim) ||
            !Enum.TryParse<UserRole>(roleClaim, out UserRole userRole))
        {
            return null;
        }
        return new SecurityContext(subjectId, userRole);
    }
}
```

Zdrojový kód 15 – Třída SecurityContextProvider pro poskytování identity

Autorizace

Pro splnění požadavku na automatickou autorizaci, která má dle návrhu probíhat ve vrstvě aplikační logiky, bude využito tzv. chování v knihovně MediatR. Chování umožňuje automaticky aplikovat další logiku na požadavky a odpovědi, implementuje tak návrhový vzor řetězec odpovědnosti. Autorizace má zajistit, aby určité požadavky směl prostředníkovi zaslat pouze ověřený uživatel, případně i s určitou uživatelskou rolí.

Pro implementaci mechanismu je nutné vytvořit autorizační atribut (viz zdrojový kód 16), kterým je možné označit zabezpečené třídy požadavků. Atributu lze předat kolekci uživatelských rolí, které jsou oprávněny daný požadavek zaslat.

```
[AttributeUsage(AttributeTargets.Class, AllowMultiple = false)]
internal class AuthorizeAttribute : Attribute
{
    public UserRole[] Roles { get; set; }
}
```

Zdrojový kód 16 – Autorizační atribut pro požadavky aplikační logiky

Třída *AuthorizationBehavior*, implementující rozhraní *IPipelineBehavior*, představuje zmíněné chování. Aplikovaná logika je zapsána v metodě *Handle* před voláním delegátu *next*, jelikož je nutné autorizaci provést při příchodu požadavku. Aby bylo možné ověřit, že uživatel má požadovanou roli, je nutné mít k dispozici instanci služby *ISecurityContextProvider*. V autorizačním chování je nejprve zkontrolováno, zda požadavek obsahuje autorizační atribut, pokud ne, je řízení předáno dalšímu chování v pipeline. Dále je zkontrolováno, zda *SecurityContext* není *null* a zda uživatel má požadovanou roli. Pokud jeden z těchto požadavků není splněn, pak je hozena odpovídající výjimka.

Autorizační atribut a autorizační chování jsou organizovány v adresáři *Common* v projektu *App.Logic*. Chování je konfigurováno při registraci knihovny MediatR v souboru

DependencyInjection (viz zdrojový kód 17). Pořadí spouštění chování je závislé na pořadí, ve kterém jsou konfigurovány.

```
services
    .AddMediatR(Assembly.GetExecutingAssembly())
    .AddTransient(
        typeof(IPipelineBehavior<,>), typeof(AuthorizationBehavior<,>));
```

Zdrojový kód 17 – Registrace služby MediatR a chování

Validace

Stejně jako autorizace, tak i validace má být dle návrhu automatický mechanismus ve vrstvě aplikační logiky. K implementaci bude opět využito chování z knihovny MediatR. Aby nebylo nutné psát celá validační pravidla, lze využít knihovnu FluentValidation, která poskytuje mnoho užitečných metod pro validaci. Použití je založeno na vytvoření nové třídy ke každé validované třídě. Validační třída je potomkem abstraktní třídy *AbstractValidator* a validační pravidla jsou definována v jejím konstruktoru (viz zdrojový kód 18).

Výhodou FluentValidation je podpora vkládání závislostí, navíc umožňuje automaticky zaregistrovat všechny validační třídy skrze reflexi. Chování si následně vyžádá odpovídající validační třídy pro příchozí požadavek, pokud validační třída existuje, spustí asynchronní kontrolu pomocí metody *ValidateAsync*.

Jelikož je cílem validovat požadavky ve vrstvě aplikační logiky, jsou součástí stejného souboru, ve kterém se již nachází třída požadavku a odpovídající handler. Vzhledem k tomu, že se nejedná o rozsáhlé třídy, nezpůsobí to nepřehlednost, naopak to umožní programátorovi pracovat rychleji.

Do souboru *DependencyInjection* je nutné doplnit konfiguraci validačního chování, a to za autorizační chování. Pokud tedy přijde neautorizovaný požadavek, bude zpracování ukončeno ještě před validací. Dále musí být přidána zmíněná automatická registrace validátorů metodou *AddValidatorsFromAssembly*.

```
internal class CreateUserRequestValidator
    : AbstractValidator<CreateUserRequest>
{
    public CreateUserRequestValidator()
    {
        RuleFor(x => x.Email).NotEmpty().EmailAddress();
        RuleFor(x => x.Password).NotEmpty().MinimumLength(8);
    }
}
```

Zdrojový kód 18 – Validační třída pro požadavek CreateUser

Přihlášení uživatele

Aby mohlo poskytování identity i autorizační mechanismus fungovat, musí být k dispozici možnost přihlášení uživatele. V šabloně architektury bude implementováno přihlášení pomocí lokálního účtu (email a heslo). V případě, že budou předložené údaje odpovídat uloženým hodnotám v databázi, bude vygenerován přístupový token typu JWT, který byl popsán v kapitole 3.6.

Přihlášení pomocí lokálního účtu bude realizovat handler pro požadavek *UserBasicLoginRequest*. Nejprve je nutné nalézt uživatele podle předložené emailové adresy, to zajistí nová metoda v repozitáři uživatelů s názvem *GetByEmailAsync*. Následně se ověří, zda předložené heslo odpovídá uložené hodnotě, která však představuje hash původního hesla. Tato logika bude delegována službě *ICryptographyService*, která již umožňuje vytvářet hash textového řetězce, nyní bude schopna hash i zpětně ověřit. Pokud je uživatel dle předloženého emailu nalezen a heslo odpovídá uložené hodnotě, je možné vygenerovat přístupový token.

Princip jedné odpovědnosti říká, že generaci přístupového tokenu má mít na starost jiná třída. Vznikne tedy nová služba *IWebTokenService*, která umožní generaci přístupového tokenu dle předaných parametrů. Parametry budou hodnoty uložené v tokenu (tzv. claims) a také životnost tokenu. Životnost je řízena konfiguračním zdrojem, kde bude hodnota uložena pod klíčem „Application:AccessTokenLifetime“, pro jednodušší zacházení je celá sekce „Application“ zaregistrována do DI kontejneru v souboru *DependencyInjection* (viz zdrojový kód 19). Generovaný JWT musí být podepsán klíčem, ten bude uložen v konfiguraci aplikace pod klíčem „WebToken:SigningKey“. Ve stejné sekci konfigurace bude uložen také zřizovatel (issuer) tokenu. Zdrojový kód 20 představuje implementaci metody pro generování JWT.

Služby *IWebTokenService* i *ICryptographyService* patří do sdílené vrstvy, jelikož mohou být využity jakoukoliv vrstvou a nejsou přímo svázány s doménou aplikace.

```
public static IServiceCollection AddApplicationLogic(
    this IServiceCollection services, IConfiguration configuration)
{
    services.Configure<ApplicationSettings>(configuration.GetSection(
        ApplicationSettings.CONFIG_KEY));
    // ...
}
```

Zdrojový kód 19 – Registrace sekce „Application“ z konfigurace

```
public string CreateWebToken(IEnumerable<Claim> claims, int expiresIn)
{
    JwtSecurityTokenHandler tokenHandler = new();
    SecurityTokenDescriptor tokenDescriptor = new()
    {
        Subject = new ClaimsIdentity(claims),
        Expires = _clockProvider.Now.AddMinutes(expiresIn),
        SigningCredentials = new SigningCredentials(
            _tokenSettings.GetSecurityKey(),
            SecurityAlgorithms.HmacSha256Signature),
        Issuer = _tokenSettings.Issuer
    };
    var token = tokenHandler.CreateToken(tokenDescriptor);
    return tokenHandler.WriteToken(token);
}
```

Zdrojový kód 20 – Metoda CreateWebToken pro generaci JWT

Lokalizace

V případě webového API se lokalizace týká především chybových zpráv v odpovědích. ASP.NET umožňuje využít zabudovanou službu pro lokalizaci. Nejprve musí být zaregistrována do DI kontejneru pomocí metody *AddLocalization* včetně konfigurace *RequestLocalizationOptions* pro lokalizaci požadavků. V konfiguraci lokalizace požadavků jsou nastaveny podporované jazyky a způsob určení využívaného jazyka (např. hlavička požadavku). Pro automatické rozpoznání jazyka dle požadavku musí být využit middleware pomocí metody *UseRequestLocalization*. Tento middleware musí být na začátku pipeline pro zpracování HTTP požadavků.

Slovníky hodnot textových řetězců jsou dle návrhu ve vrstvě přístupu k datům, aby bylo možné je využít v jakémkoliv typu aplikace. Tyto slovníky jsou realizovány pomocí tzv. resource souborů. Ke každému jazyku vznikne jeden slovníkový soubor, který obsahuje kolekci záznamů typu klíč-hodnota, kde klíč je identifikátor řetězce, který má být lokalizován. Aby lokalizační služba mohla se slovníky pracovat, musí existovat třída se stejným názvem jako slovníky. Pro lokalizaci do českého a anglického jazyka tedy vzniknou následující soubory:

- *Strings.cs,*
- *Strings.cs-cz.resx,*
- *Strings.en-us.resx.*

Lokalizační službu lze využít z DI kontejneru pomocí rozhraní *IStringLocalizer<Strings>*, které umožní přistupovat k lokalizovaným hodnotám skrze klíč.

Logování

Automaticky registrovaný mechanismus logování v ASP.NET implementuje rozhraní *ILogger*. Rozhraní však není vázané pouze na ASP.NET, může být využito v projektech jiného typu. Pokud je požadováno změnit chování logování, lze změnit nastavení registrace služby pomocí stavitele webové aplikace. Pro webové API je vhodné implementovat logování všech požadavků. Pro tento požadavek je využit vlastní middleware *HttpLoggingMiddleware* (viz zdrojový kód 21), který zaznamená informace o požadavku a stavovém kódu zpracování.

```
public async Task Invoke(HttpContext httpContext)
{
    await _next(httpContext);

    string method = httpContext.Request.Method;
    string path = httpContext.Request.Path.Value;
    int statusCode = httpContext.Response.StatusCode;
    string message = $"HTTP {method} {path} responded with {statusCode}";
    _logger.LogInformation(message);
}
```

Zdrojový kód 21 – Metoda Invoke pro automatické logování HTTP požadavků

Ošetření chyb

Dle návrhu má být v architektuře k předávání informací o chybách využit tok výjimek (exception flow). Tento návrh již byl ve výše implementovaných částech respektován hozením výjimky v případě nastání chyby. Každá chybná situace je reprezentována jednou třídou výjimky, která dědí ze třídy představující typ chyby, např. *UserNotFoundException* je podtřídou *NotFoundException*.

Výjimky musí být zachyceny v prezentační vrstvě, která rozhoduje, jak na ně zareagovat. V ASP.NET aplikaci je využit middleware, který volání následujícího middlewaru provede v *try-catch* bloku. Pokud se při zpracování objeví výjimka, je dle jejího typu určen stavový kód odpovědi, např. 404 pro typ *NotFoundException*. Dále je získána lokalizovaná hodnota chybové zprávy pomocí služby implementované v kapitole 4.1.3. Klíč pro lokalizaci zprávy je uložen v každé aplikační výjimce ve vlastnosti *MessageKey*. Klientovi je následně zaslána odpověď, jejíž tělo je definováno DTO třídou *ErrorRes* (viz zdrojový kód 22) nacházející se ve složce *Models* v prezentační vrstvě. V případě výskytu nestandardní chyby, neboli chyby na straně serveru, je navíc zaznamenána detailní informace o chybě pomocí logovací služby.

```
internal class ErrorRes
{
    public int StatusCode { get; set; }
    public string Message { get; set; }
}
```

Zdrojový kód 22 – DTO Třída ErrorRes pro chybové odpovědi

Koncové body

Koncové body jsou v ASP.NET aplikaci vytvořeny jako metody tzv. kontroleru. Pomocí atributů je určeno na jakou HTTP metodu reagují a také jaká je relativní cesta daného koncového bodu. Metody představující koncové body mají jedinou odpovědnost – předat přijatá data mediátoru a po zpracování vrátit klientovi odpověď. Třída kontroleru musí dědit ze třídy *ControllerBase* a musí být označena atributem *ApiController*. Instance mediátoru může být získána z DI kontejneru v konstruktoru, ovšem muselo by to být opakováno ve všech kontrolerech. Vhodnější je vytvořit vlastní nadtřídu *BaseApiController*, která dědí z *ControllerBase* a instanci mediátoru získá z vlastnosti *HttpContext.RequestServices*. Pro konkrétní kontrolery je vystavena metoda *Mediate*, která vložený mediátor využívá.

Příklad koncového bodu pro vytvoření úkolu je ve zdrojovém kódu 23. Metoda *Created* zajišťuje navrácení stavového kódu 201 a vytvořeného objektu v těle odpovědi. Lze si všimnout, že jako definice těla požadavku je využita třída *CreateTodoItemRequest*, která je rovnou předávána mediátoru, jelikož třída pochází z vrstvy aplikační logiky.

```
[HttpPost]
public async Task<ActionResult<TodoItemRes>> Create(
    CreateTodoItemRequest body)
{
    return Created(await Mediate(body));
}
```

Zdrojový kód 23 – Metoda koncového bodu pro vytvoření úkolu

Jednotkové testy

Poslední částí, kterou je nutné v šabloně implementovat, jsou jednotkové testy. Jelikož je architektura založena na vkládání závislostí, lze jednoduše a izolovaně testovat jednotlivé části aplikace. Pro jednotkové testy každé vrstvy vznikne nový projekt s příponou „UnitTest“ v názvu. Aby bylo možné testovat třídy s modifikátorem přístupnosti *internal*, musí být upravena konfigurace testovaného projektu (viz zdrojový kód 24).

```
<AssemblyAttribute Include="System.CompilerServices.InternalsVisibleTo">
    <_Parameter1>Cleanish.App.Logic.UnitTest</_Parameter1>
</AssemblyAttribute>
```

Zdrojový kód 24 – Nastavení přístupu ke třídám s modifikátorem internal

V .NET lze jednotkové testy vytvořit za pomocí různých frameworků, v této práci bude využit framework xUnit a struktura testu AAA (Arrange-Act-Assert). Konkrétní implementace závislých služeb budou mockovány (viz kapitola 3.3.1), k tomu poslouží knihovna Moq. Adresářová struktura v projektu jednotkových testů je podobná jako v projektu původní vrstvy. Například pro testování logiky získání informací o aktuálním uživateli je vytvořena třída *GetCurrentUserTest*, která je zanořena v adresáři *Users*.

Proces mockování v knihovně Moq začíná vytvořením instance generické třídy *Mock* s rozhraním mockované služby. Následně je na dané instanci zavolána metoda *Setup*, která umožňuje nastavit návratové hodnoty metod původního rozhraní. Ve zdrojovém kódu 25 je nastaveno mockování repozitáře uživatelů. Definuje, že po přijetí libovolné hodnoty typu *Guid* v parametru metody *GetByIdAsync* je vrácena hodnota *null*, tím je realizován testovací případ nenalezení uživatele. K samotné instanci mockované služby lze následně přistoupit pomocí vlastnosti *Object*. Stejným způsobem lze mockovat libovolnou službu.

Testované třídě *GetCurrentUserHandler* jsou mockované instance předány v konstruktoru, následně je volána metoda *Handle* a na závěr je ověřeno, že nastal požadovaný stav, v tomto případě výskyt výjimky *UserNotFoundException*.

```
var mockUserRepository = new Mock<IUserRepository>();
mockUserRepository
    .Setup(m => m.GetByIdAsync(It.IsAny<Guid>()))
    .ReturnsAsync(() => null);
```

Zdrojový kód 25 – Mockování repozitáře uživatelů

4.1.4 Sestavení šablony

Aby bylo možné využít vyvinuté řešení jako šablonu pro nová řešení, musí být k dispozici soubor *template.json* zanořený v adresáři *.template.config*. V tomto souboru jsou uvedeny následující informace o šabloně: celý název, identifikátor, popis, autor, klasifikace a klíčová slova. Dále je v souboru nastaveno vyřazení adresářů a souborů, které nemají být v novém řešení vytvořeny.

Celý název vytvořené šablony je „Cleanish architecture - Web API“ a je zařazena do kategorií web a obecné. Z řešení jsou vyřazeny struktury související s verzovacím systémem Git a adresáře obsahující výstupy sestavování řešení. Důležitou vlastností v konfiguraci

šablony je také *sourceName* s hodnotou „Cleanish“. Při vytváření nového řešení jsou všechny výskyty hodnoty *sourceName* nahrazeny novým názvem. V případě této šablony dojde například ke změně názvů jmenných prostorů a názvů projektů.

Po nastavení hodnot v souboru *template.json* lze šablonu nainstalovat. Instalace šablony je provedena příkazem *dotnet new install*. Po úspěšné instalaci lze šablonu využívat ve vývojovém prostředí Visual Studio i pomocí příkazu *dotnet new*.

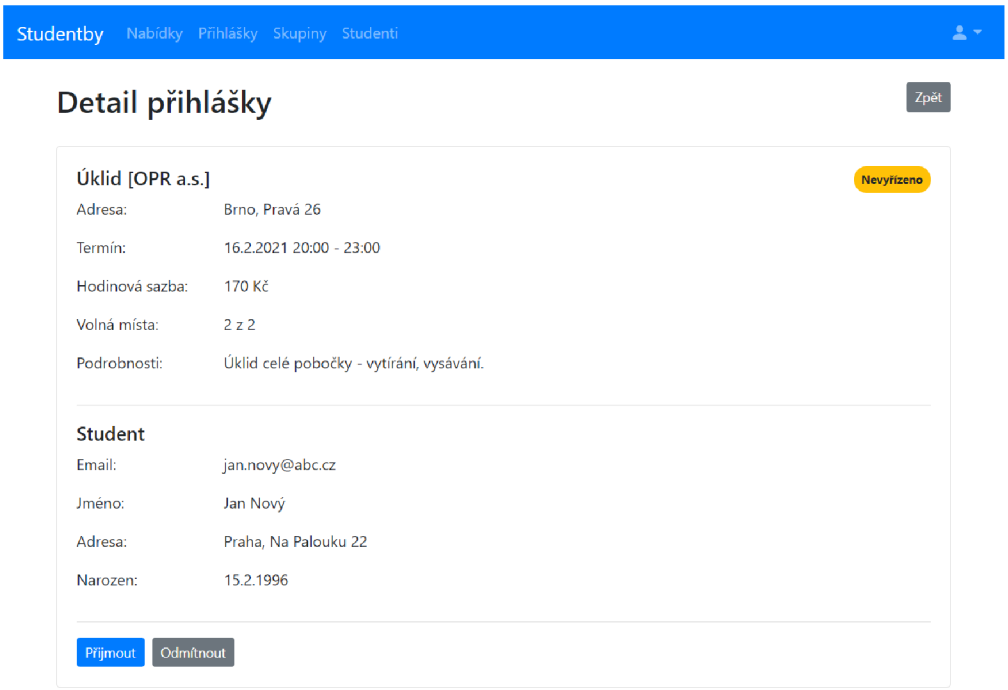
4.2 Aplikace pro vyhledávání a nabízení prací pro studenty

Dílčím cílem této práce je zavedení architektury v již existující aplikaci Studentby, která byla vytvořena v rámci bakalářské práce *Aplikace pro vyhledávání a nabízení prací pro studenty* [1], kterou tato práce rozšiřuje. Nejprve bude shrnut stav aplikace a budou stanoveny požadavky na rozšíření. Následně dojde k převedení aplikace na novou architekturu a k implementaci nových rozšíření. Důvodem pro zavádění architektury v řešení Studentby je odhalení možných nedostatků či omezení navržené architektury a také šablony.

Webová aplikace Studentby slouží ke sjednávání jednorázových prací pro studenty. Aplikaci využívají tři role uživatelů. Studentům je po vstupním pohovoru aktivován účet a mohou vytvářet přihlášky na existující pracovní nabídky. Nabídky jsou vytvářeny zaměstnavateli, kteří jsou reprezentovány tzv. skupinou, ve které může být více uživatelských účtů. Dále mají do aplikace přístup operátoři, kteří spravují skupiny zaměstnavatelů, aktivují studentské účty, schvalují přihlášky a také zaznamenávají docházku.

4.2.1 Existující řešení

Webová aplikace Studentby je tvořena serverovou a klientskou částí. Serverová část je realizována jako REST API, které je implementováno pomocí ASP.NET Core 3.0. Toto API využívá jednostránková aplikace vytvořená pomocí frameworků Vue.js 2 a Bootstrap. K persistenci dat slouží databázový systém Microsoft SQL Server a ORM nástroj Entity Framework. Na obrázku 14 je ukázka uživatelského rozhraní.

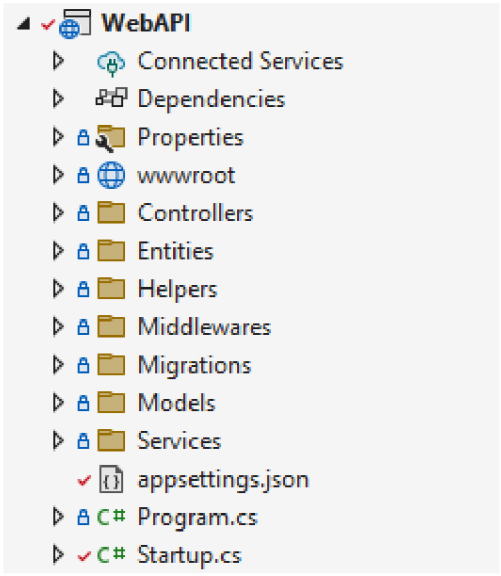


Obrázek 14 – Uživatelské rozhraní aplikace Studentby

Architektura serverové části

Webové API je implementováno jako monolitická ASP.NET aplikace typu all-in-one. Je tedy k dispozici pouze jeden .NET projekt a pro organizaci zdrojového kódu jsou využívány adresáře (viz obrázek 15). Adresář *Entities* obsahuje entitní třídy pro práci s databázovým systémem, obsah adresáře tedy představuje datový model řešení. Mimo entitní třídy však obsahuje i databázový kontext pro Entity Framework. Aplikační logika je zapouzdřena v tzv. službách (adresář *Services*), které jsou rozděleny podle entitních tříd. Strukturu dat poskytovaných webovým API určují DTO třídy, které se nacházení v adresáři *Models*. Všechny ostatní třídy jsou seskupeny do adresáře *Helpers*.

Problémy v současné architektuře jsou především nepřehlednost, úzká provázanost aplikační logiky s přístupem k datům, nemožnost znovupoužití v jiném typu aplikace a nedodržování principů vývoje softwaru.



Obrázek 15 – Struktura .NET řešení existující aplikcace Studentby

Implementované případy užití

V této části budou představeny případy užití, které jsou implementovány v již existujícím řešení Studentby. Vzhledem k tomu, že jsou funkce řešení podrobně rozvedeny v bakalářské práci *Aplikace pro vyhledávání a nabízení prací pro studenty* [1], budou zde uvedeny případy užití pouze ve zjednodušené struktuře (viz tabulka 1).

Případ užití	UC01 – Přihlášení uživatele
Aktér	Registrovaný uživatel

Případ užití	UC02 – Zobrazení uživatelského profilu
Aktér	Přihlášený uživatel

Případ užití	UC03 – Vytvoření nového uživatelského účtu
Aktér	Neregistrovaný student, neregistrovaný zaměstnavatel

Případ užití	UC04 – Vytvoření nové pracovní nabídky
Aktér	Přihlášený zaměstnavatel

Případ užití	UC05 – Zobrazení pracovních nabídek
Aktér	Přihlášený uživatel

Případ užití	UC06 – Vytvoření/zrušení přihlášky k pracovní nabídce
Aktér	Přihlášený aktivní student

Případ užití	UC07 – Zobrazení vlastních přihlášek k pracovním nabídkám
Aktér	Přihlášený aktivní student

Případ užití	UC08 – Vytváření skupin zaměstnavatelů
Aktér	Přihlášený operátor

Případ užití	UC09 – Aktivování/deaktivování studentského účtu
Aktér	Přihlášený operátor

Případ užití	UC10 – Schvalování přihlášek k pracovním nabídkám
Aktér	Přihlášený operátor

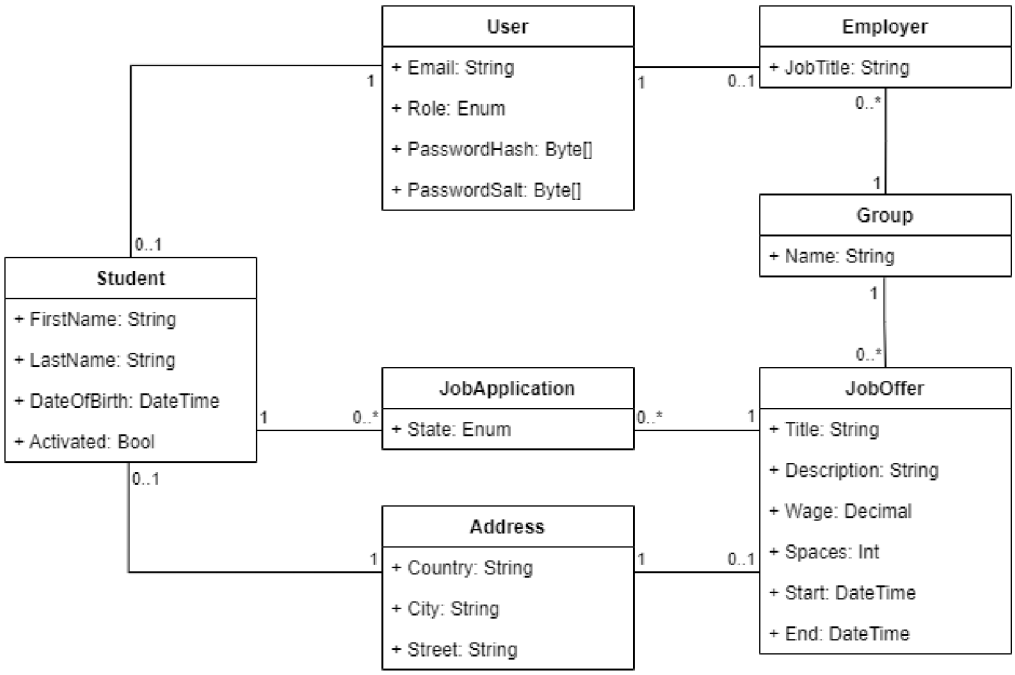
Případ užití	UC11 – Zadávání docházky u pracovních nabídek
Aktér	Přihlášený operátor

Tabulka 1 – Implementované případy užití v existujícím řešení Studentby

Datový model

Pro lepší porozumění následujících kapitol bude uveden datový model existující aplikace Studentby. Datový model je prezentován za pomocí UML diagramu tříd na obrázku 16. Třída *User* představuje obecného uživatele aplikace, pokud se jedná o studenta či zaměstnavatele je vytvořena i instance příslušné třídy s vazbou na instanci *User*. Tímto způsobem je například zajištěno, že vazbu na *JobApplication* může mít pouze uživatel s rolí studenta.

Datový model je podrobněji popsán v bakalářské práci *Aplikace pro vyhledávání a nabízení prací pro studenty* [1], v rámci které byl navržen.



Obrázek 16 – Datový model aplikace Studentby

4.2.2 Analýza nových požadavků

Vedle převedení serverové části aplikace Studentby na novou architekturu dojde i k rozšíření řešení o nové funkce. Nejprve musí být sestaven seznam nových požadavků. Je však nutné upozornit, že aplikace je rozšiřována především pro účely otestování navržené architektury a k ní vytvořené šablony. Vzhledem k této skutečnosti budou požadavky sestaveny pouze autorem a další strana nebude do vývoje zapojena.

R01 – Prodloužení doby přihlášení uživatele

Zabezpečení přístupu uživatelů je v aplikaci řešeno pomocí již popsaných JSON web tokenů. Nevýhodou těchto přístupových tokenů je, že musí mít krátkou dobu platnosti (30 – 60 minut). Jakmile token exspiruje, musí se uživatel znovu přihlásit. Je požadováno, aby uživatel zůstal přihlášený po delší dobu (v řádu jednotek dnů).

R02 – Stránkování pracovních nabídek

Při prohlížení pracovních nabídek jsou uživateli vráceny všechny záznamy z databáze. Požadavkem je rozšířit stávající koncový bod, aby podporoval stránkování. Při stránkování je vrácena pouze část záznamů dle zvolené stránky a její velikosti. Využívání stránkování vede ke zlepšení výkonu celé aplikace a také k nižšímu vytížení databázového systému.

R03 – Filtrování pracovních nabídek

Filtrování pracovních nabídek rozšiřuje předchozí požadavek na stránkování. Koncový bod musí být rozšířen tak, aby umožňoval filtrovat pracovní nabídky dle města, ve kterém bude probíhat a dle data konání. Cílem je umožnit uživatelům snadnější vyhledávání pracovních nabídek a také snížit vytížení databázového systému.

R04 – Cachování výsledků dotazů databáze

V rámci optimalizace výkonu aplikace je dále požadováno cachování dat získaných z databáze. Cachování má být zavedeno pro zobrazování pracovních nabídek a pracovních přihlášek.

R05 – Jednotkové testy

Dalším požadavkem na novou verzi aplikace je pokrytí zdrojového kódu jednotkovými testy. Důraz je kladen především na vytvoření testů pro vrstvy aplikační logiky. Jednotkové testy umožní zachytit možné chyby ještě před nasazením aplikace. Také sníží množství chyb vzniklých v již fungujících částech při zavádění dalších změn v aplikaci.

R06 – Nasazení do Microsoft Azure

Posledním požadavkem je připravit řešení k nasazení do Microsoft Azure a následně nasazení provést. K provozu samotné aplikace bude využita služba App Services. Dále musí být k dispozici i databázový systém skrze službu Azure SQL. Pro bezpečné uchování připojovacího řetězce a jiných tajemství bude využita služba Key Vault.

4.2.3 Implementace

Pro převedení původní aplikace na nově navrženou architekturu bude vytvořeno nové .NET řešení pomocí šablony vytvořené v rámci této práce. Jak již bylo uvedeno v kapitole 4.1.4, nové řešení lze ze šablony vytvořit příkazem *dotnet new cleanish-architecture -n Studentby*. Předání názvu řešení zajistí, že budou v šabloně upraveny názvy projektů a jmenných prostorů.

Vzhledem k tomu, že šablona obsahuje ukázkovou implementaci aplikace pro správu úkolů, musí být tato část odstraněna. Jedná se tedy o odstranění entitních tříd, repozitářů, aplikační logiky a koncových bodů. Zde dochází k prvnímu ověření architektury, kdy při úpravě související s aplikační doménou není měněna sdílená vrstva. Implementační vrstvy

nevyžadují další změny, jelikož hlavní využité technologie zůstávají stejné, tedy Entity Framework a Microsoft SQL Server. V konfiguračním souboru je však nutné nastavit připojovací řetězec k databázi a klíč pro podepisování přístupových tokenů. Po provedení těchto úvodních úprav je ověřeno, že se řešení úspěšně sestaví.

Převedení původního řešení

Ve chvíli, kdy je nové řešení připraveno, lze začít převádět jednotlivé části z původní aplikace. Bude postupováno směrem od datového modelu k prezentační vrstvě, tedy s ohledem na tok závislostí. Tím je zajištěno, že bude možné řešení průběžně sestavovat a včas odhalovat vzniklé chyby. Vzhledem k tomu, že kapitola 4.1.3, popisující implementaci šablony, byla doplněna mnoha ukázkami zdrojového kódu, nebudou v této části uvedeny znovu, jelikož jediný rozdíl je, že se týkají jiné aplikační domény. Místo toho bude popsán postup převádění komponent původní aplikace na novou architekturu s důrazem na rozdíly mezi řešeními.

Entitní třídy z původního adresáře *Entities* jsou přesunuty do stejnojmenného adresáře ve vrstvě přístupu k datům. První změnou je zavedení nadtřídy *BaseEntity*, která k entitním třídám přidává identifikátor a údaje o čase vytvoření a změny. Z původních tříd je tedy možné odstranit atribut *Id*. Dále je nutné upravit konfiguraci entitních tříd pro Entity Framework, která byla provedena pomocí atributů přímo z balíčku Entity Framework. V šabloně je toto řešeno pomocí fluent API. Pro každý využitý atribut existuje příslušná metoda, například místo atributu *Required* je metoda *IsRequired*. Díky využití nadtřídy *BaseEntity* není nutné konfigurovat společné atributy pro každou entitní třídu zvlášť, stejně tak název tabulky v databázi. Je přesunut také databázový kontext pro ORM, který byl původně ve stejném adresáři jako entitní třídy. Taková organizace je nepřehledná, dle architektury se nově nachází v implementační vrstvě, stejně jako třídy obsahující konfigurace entit.

Po entitních třídách lze převést na novou architekturu adresář *Services*. V tomto adresáři se nachází téměř celá aplikační logika aplikace, která je rozdělena do tříd dle zdrojů, kterých se týká, například *StudentService*. Každému koncovému bodu, a většinou i případu užití, poté odpovídá jedna metoda v takovéto třídě. Přenos aplikační logiky tedy spočívá ve vytvoření dvou nových tříd ke každé metodě v původní službě, jelikož je využíván návrhový vzor prostředník. První třída reprezentuje požadavek a druhá handler, který obsahuje

přenášenou metodu. Tyto třídy jsou organizovány do adresářů dle odpovídajícího zdroje, např. *Students/UseCases/CreateStudent.cs*.

Dále je nutné provést další úpravy, jelikož nová architektura již neumožňuje přímou práci s ORM nástrojem ve vrstvě aplikační logiky. Pro každý typ operace s databází je nutné vytvořit novou metodu v repozitářích. Celkem musí vzniknout šest repozitářů, tedy pro každou entitní třídu jeden, kromě *Address*, pro kterou není repozitář potřeba. Po dokončení musí být všechny repozitáře zaregistrovány do DI kontejneru v implementační vrstvě.

Ve vrstvě aplikační logiky musí také vzniknout třídy výjimek, jelikož pro předávání chyb je využíván tok výjimek místo původního standardního toku. Původně bylo nenalezení záznamu realizováno vrácením hodnoty *null*, nově je v takových situacích hozena výjimka typu *NotFoundException*. Podstatným zjednodušením je také využití automatického mapování místo explicitního převádění mezi entitními třídy a DTO.

Posledními rozdíly, které je třeba při přenosu aplikační logiky překonat, jsou autorizace a validace. V původním řešení jsou obě tyto funkce realizovány prezentační vrstvou pomocí atributů u požadavků a koncových bodů. Nově je autorizace nastavena pomocí atributů u požadavků ve vrstvě aplikační logiky a validace pomocí validačních tříd balíčku FluentValidation.

Po úspěšném přenosu aplikační logiky je možné vytvořit nové kontrolery pro koncové body. Metody kontrolerů mají nově pouze jedinou odpovědnost – předat prostředníkovi požadavek a následně klientovi vrátit odpověď. V původním řešení také zajišťovali získání identifikátoru uživatele a rozhodovali o vrácení stavového kódu 404 v případě, že záznam nebyl nalezen.

Prodloužení doby přihlášení uživatele

Prodloužení doby přihlášení uživatele (dle požadavku R01) lze realizovat zavedením refresh tokenů. Refresh token je náhodně vygenerovaný řetězec, který autorizační služba odesílá spolu s přístupovým tokenem. Na rozdíl od přístupového tokenu je refresh token uložený v databázi a má mnohem delší životnost. Jakmile přístupový token exspiruje, lze službě odeslat platný refresh token a v odpovědi je vrácen nový přístupový i refresh token. Díky skutečnosti, že je refresh token uložen v databázi, je možné jej odvolat při odhlášení uživatele nebo při podezření na odcizení. Po odvolání refresh tokenu jej již nelze využít pro získání nových přístupových tokenů.

Nejprve musí být vytvořena nová entitní třída *RefreshToken*, kterou bude možné zapojit do kontextu Entity Frameworku. Třída je uvedena ve zdrojovém kódu 26. Mimo atribut se samotným tokenem, obsahuje i informaci o odvolání, datum platnosti a vazbu na uživatele.

```
public class RefreshToken : BaseEntity
{
    public string Token { get; set; }
    public bool Revoked { get; set; } = false;
    public DateTime Expiration { get; set; }
    public Guid UserId { get; set; }

    public User User { get; set; }
}
```

Zdrojový kód 26 – Entitní třída RefreshToken

Dále je nutné se přesunout do vrstvy aplikační logiky, konkrétně do souboru *UserBasicLogin.cs*, kde se nachází definice požadavku a příslušného handleru. Zde pozměnit přímo handler, tak aby po vygenerování přístupového tokenu vytvořil a uložil refresh token. K vytvoření samotného tokenu poslouží služba *ICryptographyService* s metodou *GetRandomString*. Vzhledem k tomu, že je vhodné, aby exspirace tokenu byla parametrizována, musí být rozšířena třída *ApplicationSettings* a také zdrojový soubor konfigurací o vlastnost *RefreshTokenLifetime*. Hodnota vlastnosti bude nastavena na 10 080 minut (7 dní). Handler pomocí této konfigurace a služby *IClockProvider* vypočítá datum exspirace tokenu (viz zdrojový kód 27). Posledním krokem je přidání vlastnosti *RefreshToken* do DTO třídy *AuthRes*.

```
RefreshToken refreshToken = new(
    _cryptographyService.GetRandomString(32),
    _clockProvider.Now.AddMinutes(_appSettings.RefreshTokenLifetime),
    user.Id);
```

Zdrojový kód 27 – Vytvoření refresh tokenu

Po otestování, že generace refresh tokenů probíhá v pořádku, je možné implementovat i samotnou možnost obnovy přístupových tokenů. Zde se již nejedená o pouhou změnu existujících tříd. Místo toho je vystaven nový koncový bod, požadavek i související handler. Algoritmus implementovaný v handleru je následující:

1. ověřit, zda je předložený refresh token uložen v databázi,
2. dle záznamu v databázi ověřit, zda je refresh token platný,
3. vrátit nový přístupový i refresh token.

Pozornost vyžaduje poslední krok, jelikož tato operace je již prováděna v handleru pro přihlášení uživatele. Kvůli principu DRY bude zdrojový kód přesunut do statické metody ve třídě *AuthCommonLogic* a tuto metodu budou oba handlery využívat.

Poslední chybějící článek je odvolání refresh tokenu, pro které opět musí vzniknout nový koncový bod. Zpracování požadavku je však velmi jednoduché. Nejprve musí být načten refresh token z databáze a pokud je stále platný, je možné nastavit vlastnost *Revoked* na hodnotu *true*. Důležité je přidat k požadavku atribut *Authorize*, jelikož odvolat refresh token smí pouze přihlášený uživatel.

Stránkování pracovních nabídek

Prvním krokem implementace stránkování pracovních nabídek dle požadavku R02 je rozšíření třídy *ListJobOfferRequest* o vlastnosti *Page* a *PageSize*. V odpovídajícím koncovém bodu musí být typ požadavku označen atributem *FromQuery*, aby byly vstupní parametry získány z dotazu URI řetězce. Pro nové vlastnosti požadavku musí být nastavena validační pravidla, aby hodnoty byly větší než nula. Definice takových pravidel je znázorněna ve zdrojovém kódu 28.

```
RuleFor(x => x.Page).GreaterThan(0);
RuleFor(x => x.PageSize).GreaterThan(0);
```

Zdrojový kód 28 – Validační pravidla pro stránkování

Dané parametry je potřeba předat až do vrstvy přístupu k datům. Metoda *ListAsync* ve třídě *JobOfferRepository* musí být rozšířena tak, aby očekávala parametry stránkování. V *ListAsync* je místo původní metody *ReadDataAsync* použita metoda *ReadPaginatedDataAsync*, která zajistí správné cachování stránkovaných výsledků. Tato metoda ovšem nemá jako návratový typ *IEnumerable*, ale třídu *PaginatedList*, která mimo vrácených dat obsahuje také informace o celkovém počtu stránek a položek. *PaginatedList* musí být nastaven jako návratový typ také ve vrstvě aplikační logiky i v prezentační vrstvě.

Filtrování pracovních nabídek

Implementace filtrování pracovních nabídek dle požadavku R03 navazuje na předchozí požadavek R02. Stačí pouze rozšířit *ListJobOfferRequest* o vlastnosti *City* a *Date*. Obě vlastnosti jsou však nepovinné, což znamená, že *Date* musí být explicitně nastaveno jako nullovatelné. Opět musí být rozšířena metoda *ListAsync* ve vrstvě přístupu k datům, aby tyto hodnoty přijímala jako argumenty. Aby byl název více popisný je metoda přejmenována

na *FiltredListAsync*. Do těla metody je přidán zdrojový kód 29, který zajistí filtrování v případě, že hodnoty nejsou *null*.

```
if (city != null)
{
    specificationBuilder.SetFilter(jo => jo.Address.City == city);
}
if (date != null)
{
    specificationBuilder.SetFilter(jo => jo.Start.Date == date.Value.Date);
}
```

Zdrojový kód 29 – Nepovinné filtrování v repozitáři

Cachování výsledků dotazů databáze

Cachování dat získaných z databáze lze do aplikace zavést velmi snadno, jelikož tato funkce je již součástí šablony a je popsána v kapitole 4.1.3. V handlerech vybraných požadavků stačí pouze nahradit typ požadované služby v konstruktoru. V *GetJobOfferRequest* je služba *IJobOfferRepository* vnořena do generického typu *Cached*, ze kterého lze instanci repozitáře získat skrze vlastnost *Value* (viz zdrojový kód 30).

```
private readonly IJobOfferRepository _cachedJobOfferRepository;

public ListJobOffersRequestHandler(
    Cached<IJobOfferRepository> cachedJobOfferRepository)
{
    _cachedJobOfferRepository = cachedJobOfferRepository.Value;
}
```

Zdrojový kód 30 – Vyžádání cachované verze repozitáře

Jednotkové testy

Pro vytvoření jednotkových testů aplikační logiky aplikace existuje v řešení projekt *App.Logic.UnitTest*, zde jsou testy organizovány stejným způsobem jako požadavky a jejich handlery. V této části bude pro zjednodušení popsán pouze jeden testovací případ, a tím je vytváření přihlášky na již obsazenou pracovní nabídku.

Vznikne tedy třída *CreateJobApplicationTest* a v ní metoda *Handle_JobOfferFull*. Jelikož se jednotkový test má týkat pouze handleru, jsou všechny závislosti mockovány, jedná se o repozitáře, službu pro získání času a službu pro poskytování identity. Při mockování je však nutné nové implementace správně nastavit, aby jejich metody vracely správná data. Metoda *GetByIdAsync* v repozitáři pracovních nabídek musí vrátit obsazenou pracovní nabídku, například celkový počet míst je roven jedné a zároveň existuje jedna

schválená přihláška. Jakmile jsou všechny závislosti připraveny, lze vytvořit handler a v metodě *Handle* předat příslušný požadavek. Vzhledem k tomu, že očekávaný stav zpracování je výskyt výjimky *JobOfferFullException*, musí být volání handleru obaleno anonymní funkcí, která je následně testována. Toto je uvedeno i ve zdrojovém kódu 31.

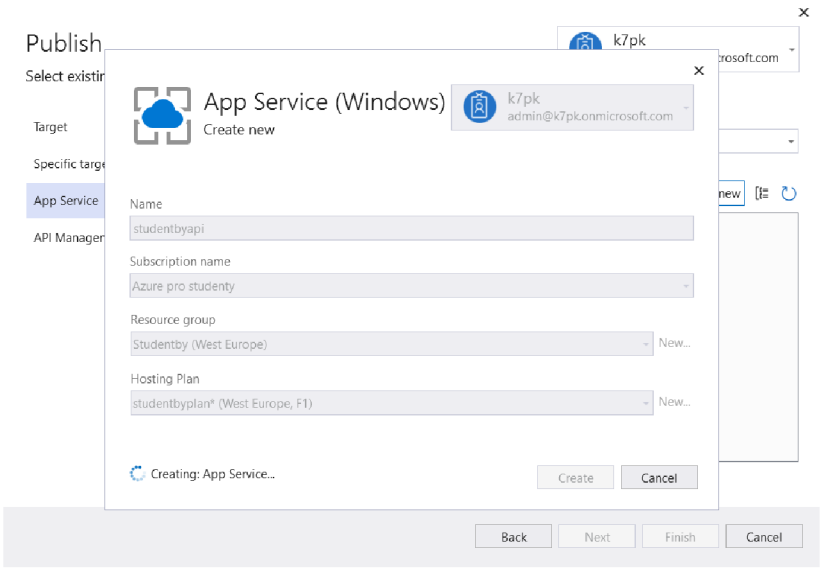
```
CreateJobApplicationRequestHandler handler = new(
    mockJobOfferRepository.Object,
    // ...
);
CreateJobApplicationRequest request = new()
{
    JobOfferId = jobOffer.Id
};

// Act
async Task<JobApplicationRes> act() =>
    await handler.Handle(request, CancellationToken.None);

// Assert
await Assert.ThrowsAsync<JobOfferFullException>(act);
```

Zdrojový kód 31 – Jednotkový test ověřující výskyt výjimky

4.2.4 Nasazení

Nasazení nové verze aplikace do cloudové platformy Microsoft Azure bude provedeno ve vývojovém prostředí Visual Studio. Služby uvedené v požadavku R06 se budou nacházet v jedné skupině zdrojů. Tato skupina bude mít přiřazenou subskripci Azure pro studenty.

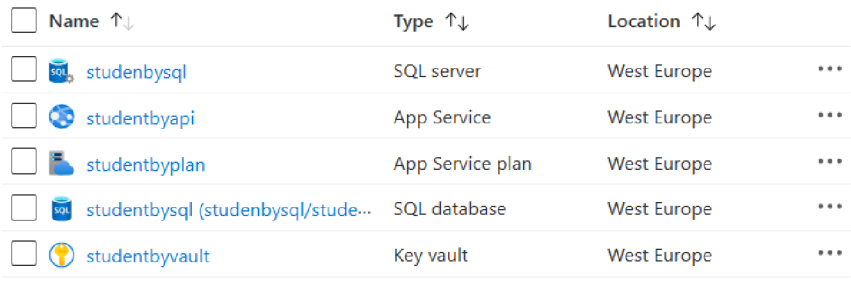
Nasazení ve Visual Studiu spočívá ve vytvoření publikačního profilu, který je v případě Azure založen na ARM šablonách. Publikační profil lze vytvořit pomocí tlačítka *Publish* v kontextové nabídce projektu prezentační vrstvy. V dialogovém okně lze následně zvolit požadované parametry vytvářeného zdroje, v tomto případě Azure App Service. Snímek obrazovky při vytváření požadovaného zdroje je uveden na obrázku 17.



Obrázek 17 – Vytvoření zdroje App Service, snímek obrazovky ve Visual Studiu

Dále musí publikační profil obsahovat definice dalších zdrojů, kterými jsou Azure SQL Database a Azure Key Vault. Tyto zdroje lze definovat v sekci *Service Dependencies* v publikačním profilu. Po nastavení parametrů databázového zdroje je zobrazena možnost uložení připojovacího řetězce do služby Azure Key Vault, která bude automaticky vytvořena. Po úspěšném vytvoření vygeneruje Visual Studio zdrojový kód pro nastavení tohoto uložiště jako konfiguračního zdroje.

Jakmile jsou všechny definice hotové, lze spustit publikaci dle vytvořeného profilu. Výsledkem publikace je vytvoření zdrojů v Microsoft Azure (viz obrázek 18). Posledním krokem je aplikování migračního skriptu databáze. Tento skript zajistí vytvoření potřebných tabulek v produkční databázi dle Entity Framework migrací. Skript lze vygenerovat příkazem *Script-Migration -Idempotent* a spustit jej lze například ve webové aplikaci Azure Portal.



Obrázek 18 – Vytvořené zdroje v Azure, snímek obrazovky v aplikaci Azure Portal

5 Výsledky a diskuse

Na základě autorem stanovených požadavků byla v této práci navržena softwarová architektura webové aplikace. Bylo požadováno, aby architektura byla vhodná pro středně velké aplikace a pro individuální vývojáře či malé týmy. Dalšími požadavky bylo zajištění nezávislosti na využívaných technologiích, respektování principů při vývoji softwaru a také možnost nezávislého testování jednotlivých komponent.

Navržená architektura vychází ze základů clean architecture, což je vrstvená monolitická architektura využívající obrácení závislostí. Nová architektura přidává navíc jednu vrstvu a také mění odpovědnosti a názvy některých původních vrstev. Přidána je sdílená vrstva, která je dostupná ve všech úrovních řešení a obsahuje společné komponenty nezávislé na účelu aplikace. Změnami prochází především vrstva domény, která se mění na vrstvu přístupu k datům a také se mění vymezení vrstvy aplikační logiky.

Dále byla vytvořena šablona pro nové ASP.NET aplikace, která vygeneruje zdrojový kód dle navržené architektury. Vygenerovaný kód obsahuje mechanismy pro automatické cachování ve vrstvě přístupu k datům, ošetření chybových situací, lokalizaci, validaci a autorizaci. Tyto mechanismy jsou implementovány tak, aby byly zcela nezávislé na konkrétním využití aplikace.

Aby byl mechanismus pro validaci a autorizaci požadavků nezávislý na prezentační vrstvě, nemohly být využity služby integrované do frameworku ASP.NET. Místo toho byla využita kombinace návrhových vzorů prostředník a řetězec odpovědnosti ve vrstvě aplikační logiky.

Problematickou částí byla implementace cachování ve vrstvě přístupu k datům. Cílem bylo umožnit vyžádání cachovaného repozitáře, který pro jakoukoliv čtecí operaci nejprve ověří, zda je hodnota již uložena v mezipaměti a pokud ne, získá hodnotu z datového zdroje a následně ji do mezipaměti uloží. Zároveň bude zajišťovat, aby v případě změny byla data z mezipaměti odstraněna. Zveřejněná řešení automatického cachování často spočívají ve vytvoření dekorátoru pro repozitář. To ovšem není dobré řešení, jelikož je nutné cachování každé operace explicitně definovat. V práci bylo cachování navrženo a implementováno lepším způsobem. Nový přístup spočívá v zamezení přímého dotazování datového zdroje ve veřejných metodách repozitáře. Místo toho je využívána privátní metoda generického repozitáře, která zajistí vytvoření cachovacího klíče dle předaných parametrů a čtení dat deleguje cachovací službě. Pro cachovací službu je zaregistrováno více instancí dle počtu

entitních tříd. Tento přístup umožňuje evikci pouze části záznamů místo vymazání celé mezipaměti.

Vytvořená šablona byla využita při migraci existující aplikace pro vyhledávání prací pro studenty, která využívá all-in-one architekturu. Převedení již zhotoveného řešení na navrženou architekturu bylo provedeno za účelem otestování architektury i šablony především z hlediska vývojářské zkušenosti. Níže jsou shrnuty výsledky tohoto testování.

V první řadě je nutné zmínit, že nová architektura je podstatně komplexnější než původní all-in-one. Bez vyšší komplexity by však nebylo možné zajistit naplnění stanovených požadavků. Implementace jednoho případu užití je v nové architektuře výrazně složitější. Je nutné vytvořit metodu kontroleru, tři třídy ve vrstvě aplikační logiky (požadavek, handler, validátor) a metodu repozitáře. Zdrojový kód celého případu užití je tedy rozčleněn do mnoha souborů, což může prodloužit dobu vývoje, avšak výhodou je větší přehlednost a dodržování principů vývoje softwaru. Zde je nutné zmínit, že by bylo vhodné rozdělit třídy požadavku, handleru a validátoru do samostatných souborů, neboť u větších případů užití se v jednom souboru hůře orientuje.

Mechanismus pro cachování přístupu k datům byl velmi dobře navržen a implementován. Vývojáři stačí pouze vytvořit metodu repozitáře a následně repozitář využít v aplikační logice. Přepínání mezi repozitářem bez cachování a s cachováním pomocí generického typu *Cached* je rovněž jednoduché. Vrstva přístupu k datům by však mohla být rozšířena, aby repozitáře byly registrovány do DI kontejneru automaticky, například skrze reflexi. Nyní musí programátor všechny nové repozitáře registrovat manuálně.

Velmi užitečným prvkem je také služba pro získání identity ve vrstvě aplikační logiky. Bez této služby by bylo nutné rozšiřovat všechny požadavky, které pracují s identitou uživatele, o uživatelský identifikátor. Místo toho lze k získání identity aktuálně přihlášeného uživatele využít službu, jejíž implementace je definována prezentační vrstvou, jelikož závisí na typu aplikace.

Výhodou ošetření očekávaných chybových situací pomocí toku výjimek je především jednoduchost a přehlednost. Avšak jsou zde i nevýhody. Mezi ně patří dopad na výkon aplikace, ale také obtížnější hledání chyb, než v případě standardního toku. Dále by bylo vhodné zjednodušit strukturu výjimek tak, aby pro lokalizaci chybových zpráv nebyla využívána hodnota atributu, ale název výjimky získaný pomocí reflexe.

Navržená architektura umožnila vytvořit jednotkové testy aplikační logiky bez závislostí na prezentační vrstvě a vrstvě přístupu k datům. Toho bylo dosaženo pomocí

obrácení závislostí a mockování. Některé případy užití však závisí na mnoha službách a příprava testovacích implementací těchto služeb je časově náročná. To může vývojáře odrazovat od pokrytí jednotkovými testy.

Vytvořená šablona kromě popsaných mechanismů obsahuje zdrojový kód jednoduché aplikace pro správu úkolů. To vývojářům umožňuje seznámit se s principy architektury na konkrétním případě. Tuto část zdrojového kódu lze navíc velmi snadno odebrat. Bylo by ovšem vhodné rozšířit šablonu o parametr, který by umožňoval vygenerovat zdrojový kód bez této části. Stejně tak by mohl vzniknout další parametr, který by ovlivňovat vygenerování zdrojového kódu pro autentizaci a autorizaci. Neboť není časté, aby si aplikace tyto mechanismy zajišťovaly. Většinou jsou využívány externí autorizační služby.

I přes větší komplexitu nové architektury nebylo převedení existující aplikace časově náročné. Celý proces, včetně nasazení do cloudové platformu Microsoft Azure, si vyžádal pracnost 2,5 MD. Tento údaj je pochopitelně ovlivněn tím, že vývojář provádějící migraci a autor architektury je tatáž osoba. Lze však očekávat, že u vývojářů seznámených s clean architecture by pracnost byla podobná kvůli stejnému základu obou architektur.

Zdrojový kód vytvořené šablony a nové verze webové aplikace Studentby je k dispozici v příloze této práce.

6 Závěr

Cílem této práce bylo navrhnout architekturu serverové části webové aplikace a následně vytvořit šablonu pro ASP.NET aplikace dle nově navržené architektury. Šablona měla být dále využita k migraci aplikace pro sjednávání prací pro studenty a její následné rozšíření. Tato aplikace byla vytvořena v bakalářské práce *Aplikace pro vyhledávání a nabízení prací pro studenty* [1], kterou tato práce rozšiřuje. Převedení již zhotoveného řešení na novou architekturu mělo být provedeno za účelem otestování z pohledu vývojářské zkušenosti.

Pro splnění stanovených cílů byly využity teoretické základy uvedené v první části práce. Byly popsány základní principy při vývoji softwaru, návrhové vzory, architektury a typy webových aplikací. Dále byly představeny konkrétní technologie využité v praktické části, například ASP.NET, Microsoft Azure a Entity Framework.

V praktické části práce byly nejprve autorem stanoveny požadavky na novou architekturu a poté byl vytvořen návrh vrstev, principů i obecných funkcí architektury. Následně byl návrh implementován jako šablona pro ASP.NET aplikace, která umožňuje generovat zdrojový kód pro nové projekty. Část obsahující implementaci šablony je doplněna ukázkami zdrojového kódu.

Poté byla představena aplikace pro sjednávání prací pro studenty, zejména implementované případy užití a datový model. Byly však uvedeny i požadavky na další rozšíření aplikace. Po vytvoření nového projektu pomocí zmíněné šablony byla aplikace postupně převedena na novou architekturu, dále byly implementovány nové funkce a na závěr byla aplikace nasazena do cloudové platformy Microsoft Azure.

V kapitole 5 byly nejprve shrnuty aspekty navržené architektury a problémy při implementaci. Dále byly uvedeny výhody a nevýhody šablony i architektury z pohledu vývojářské zkušenosti, které byly objeveny při migraci již existující aplikace. Vyvinutá architektura a šablona splňují všechny stanovené požadavky. Při další práci se šablonou nebyly objeveny žádné závažné nedostatky. Pouze bylo poukázáno na možnosti, jak by bylo možné některé části řešení zlepšit. Tím byly splněny všechny cíle této diplomové práce.

7 Seznam použitých zdrojů

[1] HROUDA, Adam. *Aplikace pro vyhledávání a nabízení prací pro studenty*. 2021. Bakalářská práce. Česká zemědělská univerzita v Praze.

[2] CLARK, Dan. *Beginning C# Object-Oriented Programming*. 2013. ISBN 9781430249351.

[3] Introduction to Object Oriented Programming. In: *The University of Texas - Computer Science* [online]. [cit. 2022-07-02]. Dostupné z: https://www.cs.utexas.edu/users/mitra/csSpring2017/cs303/lectures/oop.html

[4] ČÁPKA, David. Úvod do objektově orientovaného programování v C#. In: *Itnetwork.cz* [online]. [cit. 2022-07-02]. Dostupné z: https://www.itnetwork.cz/csharp/oop/c-sharp-tutorial-uvod-do-objektove-orientovaneho-programovani

[5] Principles of Object-Oriented Design. In: *The University of Texas at San Antonio: Department of Computer Science* [online]. [cit. 2022-07-02]. Dostupné z: http://www.cs.utsa.edu/~cs3443/notes/designPrinciples/designPrinciples.html

[6] MARTIN, Robert C. *Agile Software Development, Principles, Patterns, and Practices*. 2003. ISBN 9780135974445.

[7] FOWLER, Martin. CodeSmell. In: *MartinFowler.com* [online]. [cit. 2022-07-11]. Dostupné z: https://martinfowler.com/bliki/CodeSmell.html

[8] PODKAMENNYI, Valentin. Principles of Software Engineering: SOLID DRY KISS — What does it mean?. In: *Medium* [online]. [cit. 2022-07-11]. Dostupné z: https://vpodk.medium.com/principles-of-software-engineering-6b702faf74a6

[9] MARTIN, Robert C. *Clean architecture: a craftsman's guide to software structure and design*. 2018. ISBN 978-0134494166.

[10] JOSHI, Bipin. *Beginning SOLID Principles and Design Patterns for ASP. NET Developers*. 2016. ISBN 9781484218471.

[11] What's a design pattern?. In: *Refactoring.Guru* [online]. [cit. 2022-07-11]. Dostupné z: https://refactoring.guru/design-patterns/what-is-pattern

[12] GAMMA, Erich, Richard HELM, Ralph JOHNSON a John VLISSIDES. *Design patterns: Elements of Reusable Object-Oriented Software*. 1995. ISBN 978-020-1633-610.

[13] SEEMANN, Mark a Steven VAN DEURSEN. *Dependency Injection Principles, practices and patterns*. 2019. ISBN 978-1617294730.

[14] SMITH, Steve. *Architecting Modern Web Applications with ASP.NET Core and Microsoft Azure*. Microsoft Developer Division.

[15] FOWLER, Martin. Inversion of Control Containers and the Dependency Injection pattern. In: *MartinFowler.com* [online]. [cit. 2022-07-13]. Dostupné z: https://martinfowler.com/articles/injection.html

[16] FOWLER, Martin. *Patterns of enterprise application architecture*. 2003. The Addison-Wesley Signature Series. ISBN 03-211-2742-0.

[17] FOWLER, Martin. Fail Fast. In: *MartinFowler.com* [online]. [cit. 2022-07-13]. Dostupné z: https://www.martinfowler.com/ieeeSoftware/failFast.pdf

[18] Guard Clause. In: *DevIQ* [online]. [cit. 2022-07-13]. Dostupné z: https://deviq.com/design-patterns/guard-clause

[19] HTTP. In: *MDN* [online]. [cit. 2022-08-23]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/HTTP

[20] FIELDING, Roy, IRVINE a W3C. Hypertext Transfer Protocol - HTTP/1.1. In: *RFC* [online]. [cit. 2022-08-23]. Dostupné z: https://www.rfc-editor.org/rfc/rfc2616

[21] Safe (HTTP Methods). In: *MDN* [online]. [cit. 2022-08-23]. Dostupné z: https://developer.mozilla.org/en-US/docs/Glossary/Safe/HTTP

[22] Idempotent. In: *MDN* [online]. [cit. 2022-08-23]. Dostupné z: https://developer.mozilla.org/en-US/docs/Glossary/Idempotent

[23] What Is An API?. In: *AWS* [online]. [cit. 2022-08-23]. Dostupné z: https://aws.amazon.com/what-is/api/

[24] FIELDING, Thomas Roy. *Architectural Styles and the Design of Network-based Software Architectures*. 2000. DISSERTATION. UNIVERSITY OF CALIFORNIA.

[25] GraphQL. In: *GraphQL Specification Versions* [online]. [cit. 2022-08-23]. Dostupné z: https://spec.graphql.org/October2021/

[26] OWASP API Security Project. In: *OWASP* [online]. [cit. 2022-08-23]. Dostupné z: https://owasp.org/www-project-api-security/

[27] Cross-Origin Resource Sharing (CORS). In: *MDN* [online]. [cit. 2022-08-23]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS

[28] HARGUINDEGUY, Bernard. Everything You Need to Know about API Security. In: *PingIdentity* [online]. [cit. 2022-08-23]. Dostupné z: https://www.pingidentity.com/en/resources/blog/post/complete-guide-to-api-security.html

[29] JSON Web Tokens. In: *OAuth0* [online]. [cit. 2022-08-23]. Dostupné z: https://auth0.com/docs/secure/tokens/json-web-tokens

[30] Different Types of Web Application Development. In: *Allcode* [online]. [cit. 2022-09-02]. Dostupné z: https://allcode.com/different-types-of-web-application-development/

[31] STRAUSS, Dirk. *Creating ASP.NET Core Web Applications: Proven Approaches to Application Design and Development*. 2021. ISBN 9781484268278.

[32] ANSARI, Rijwan. What Is Clean Architecture. In: *C# Corner* [online]. 2022 [cit. 2022-09-02]. Dostupné z: https://www.c-sharpcorner.com/article/what-is-clean-architecture/

[33] HARRIS, Chandler. Microservices vs. monolithic architecture. In: *Atlassian* [online]. [cit. 2022-09-02]. Dostupné z: https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith

[34] PALERMO, Jeffrey. *The Onion Architecture* [online]. In: . [cit. 2022-09-02]. Dostupné z: https://jeffreypalermo.com/2008/07/the-onion-architecture-part-1/

[35] HARVEY, Cynthia. What is Microsoft Azure Cloud & What is It Used for?. In: *Datamation* [online]. [cit. 2022-10-21]. Dostupné z: https://www.datamation.com/cloud/microsoft-azure-cloud/

[36] Azure products. In: *Microsoft Azure* [online]. [cit. 2022-10-21]. Dostupné z: https://azure.microsoft.com/en-us/services/

[37] ZAAL, Sjoukje. *Microsoft Azure Architect Technologies: Exam Guide AZ-300: A guide to preparing for the AZ-300 Microsoft Azure Architect Technologies certification exam*. 2020. ISBN 978-1-838-55353-1.

[38] What is Azure Active Directory?. In: *Microsoft Learn* [online]. [cit. 2022-10-21]. Dostupné z: https://learn.microsoft.com/en-us/azure/active-directory/fundamentals/active-directory-whatis

[39] What is Azure SQL Database?. In: *Microsoft Learn* [online]. [cit. 2022-10-21]. Dostupné z: https://learn.microsoft.com/en-us/azure/azure-sql/database/sql-database-paas-overview?view=azuresql

[40] What is Azure Resource Manager?. In: *Microsoft Docs* [online]. [cit. 2022-10-21]. Dostupné z: https://docs.microsoft.com/en-us/azure/azure-resource-manager/management/overview

[41] Azure portal overview. In: *Microsoft Docs* [online]. [cit. 2022-10-21]. Dostupné z: https://docs.microsoft.com/en-us/azure/azure-portal/azure-portal-overview

[42] Cloud Shell. In: *Microsoft Azure* [online]. [cit. 2022-10-21]. Dostupné z: https://azure.microsoft.com/en-us/features/cloud-shell/#overview

[43] Azure SDK Releases. In: *Azure SDKs* [online]. [cit. 2022-10-21]. Dostupné z: https://azure.github.io/azure-sdk/releases/latest/index.html

[44] R, Dave. ARM Templates Or Azure Bicep — What Should I Use?. In: *Medium* [online]. [cit. 2022-10-21]. Dostupné z: https://medium.com/codex/arm-templates-or-azure-bicep-what-should-i-use-14e8662d3f27

[45] What is .NET? Introduction and overview. In: *Microsoft Learn* [online]. [cit. 2022-11-30]. Dostupné z: https://learn.microsoft.com/en-us/dotnet/core/introduction

[46] PRICE, Mark J. *C# 9 and .NET 5 – Modern Cross-Platform Development*. 2020. ISBN 9781800568105.

[47] Language Integrated Query (LINQ) (C#). In: *Microsoft Learn* [online]. [cit. 2022-11-30]. Dostupné z: https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/

[48] ASP.NET Core Middleware. In: *Microsoft Learn* [online]. [cit. 2022-11-30]. Dostupné z: https://learn.microsoft.com/en-us/aspnet/core/fundamentals/middleware/?view=aspnetcore-7.0

[49] Configuration in ASP.NET Core. In: *Microsoft Learn* [online]. [cit. 2022-11-30]. Dostupné z: https://learn.microsoft.com/en-us/aspnet/core/fundamentals/configuration

[50] Entity Framework Core. In: *Microsoft Learn* [online]. [cit. 2022-11-30]. Dostupné z: https://learn.microsoft.com/en-us/ef/core/

[51] GLABAZŇA, Tomáš. Úvod do Vue a první aplikace. In: *Itnetwork.cz* [online]. [cit. 2022-12-07]. Dostupné z: https://www.itnetwork.cz/javascript/vuejs/uvod-do-vuejs-a-prvni-aplikace

[52] Introduction. In: *Vue.js* [online]. [cit. 2022-12-07]. Dostupné z: https://vuejs.org/guide/introduction.html

[53] MONTGOMERY, Pete. Caching the results of LINQ queries. In: *Monty's Gush* [online]. [cit. 2023-01-09]. Dostupné z: https://petemontgomery.wordpress.com/2008/08/07/caching-the-results-of-linq-queries/

8 Seznam obrázků, tabulek, grafů a zkratek

8.1 Seznam obrázků

8.2 Seznam zdrojových kódů

8.3 Seznam tabulek

8.4 Seznam použitých zkratek

API – Application Programming Interface
CSS – Cascading Style Sheets
DI – Dependency Injection
DTO – Data Transfer Object
HTML – Hypertext Markup Language
HTTP – Hypertext Transfer Protocol
JSON – JavaScript Object Notation
JWT – JSON Web Token
ORM – Object Relational Mapping
REST – Representational State Transfer
SPA – Single Page Application

Přílohy

Na přiloženém CD se nachází zdrojový kód vytvořené šablony Cleanish architecture a také zdrojový kód aplikace Studentby, která byla v rámci této práce převedena na novou architekturu a následně rozšířena.