Středoškolská technika 2025

Setkání a prezentace prací středoškolských studentů na ČVUT

Scoolib – digitalizace seznamů maturitní četby

Adam Hojer, Vojtěch Glejdura

Střední škola informatiky a finančních služeb, Plzeň,

Klatovská 200 G, 301 00 Plzeň

Prohlášení

Prohlašujeme, že jsme naši práci SOČ vypracovali samostatně a použili jsme pouze prameny a literaturu uvedené v seznamu bibliografických záznamů.

Prohlašujeme, že tištěná verze a elektronická verze soutěžní práce SOČ jsou shodné.

Nemáme závažný důvod proti zpřístupňování této práce v souladu se zákonem č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších předpisů.

V Plzni dne 10. 3. 2025 Vojtěch Glejdura ……

V Plzni dne 10. 3. 2025 Adam Hojer ……

ANOTACE

Tato práce se zabývá přípravou žáků k rozboru literárních děl během ústní části maturitní zkoušky z českého jazyka a literatury. Žáci musí během přípravy podniknout mnoho kroků. Ještě před samotným čtením knížek a přípravou rozborů si musí projít seznam dostupných knih a z něj vybrat určitý počet a zkombinovat ho takovým způsobem, aby došlo ke splnění požadavků stanovených školou. Seznam knih je u většiny škol veden ve formě dokumentu s názvy jednotlivých děl, takový seznam ale moc úsilí žákovi neušetří.

Cílem této práce bylo vytvoření webové aplikace, jež by byla maturujícím žákům nápomocná během procesu přípravy na takovou zkoušku. Před samotným začátkem vývoje proběhl ze strany autorů mezi vrstevníky průzkum týkající se přípravy na maturitní zkoušku, a dle výsledků si následně přesně definovali, jaké funkce budou chtít do aplikace zakomponovat. Při tvorbě použili moderní přístupy vývoje a pomocí technologií React a C# Web API vytvořili funkční řešení. Výstupní aplikace správně implementuje definované cíle a je připravena ke spuštění pilotní verze na některé ze škol.

Při vývoji aplikace se autoři zaměřili na její snadnou použitelnost a přehlednost. Pro její cílové uživatele tak bude představovat účinný prostředek k získání přehledu o nabídce knih. Ke každé knize bude k dispozici velká sada informací, čímž bude omezena potřeba dohledávat takové detaily na internetu. Díky integrované možnosti skládání seznamu četby získají její uživatelé rychlý přehled o stavu splnění jejich kritérií. Aby byla hodnota aplikace jako informačního zdroje ještě více posílena, zavedli zde vývojáři možnost vytvářet k daným knihám rozbory, jež lze dokonce sdílet mezi ostatní žáky.

Hlavní hodnotou aplikace je fakt, že ať už se žáci nacházejí v jakékoliv části procesu přípravy ke zkoušce, tato aplikace jim vždy bude v tomto kroku nápomocná.

KLÍČOVÁ SLOVA

Maturitní zkouška; digitalizace; webová aplikace; vzdělávání; informační zdroj; knihy; autoři; rozbory; seznam četby

ANOTATION

This work addresses the preparation of students for the analysis of literary works during the oral part of the final exam in Czech language and literature. Students must undertake several steps as part of their preparation. Before they even begin reading the books and creating analyses, they need to review the list of available books, select a specified number of them, and combine them in a way that fulfills the requirements set by the school. In the majority of schools, the list of books is provided as a document containing the titles of the books, but this does not help the student that much.

The goal of this work was to create a web application that would assist graduating students during the preparation process for such an exam. Before starting the development process, the

authors conducted a survey among their classmates regarding the preparation for the final exam, and according to the results, they defined exactly what features they wanted to include in the application. They used modern development approaches and created a functional solution using React and C# Web API technologies. The output application properly implements the defined goals and is ready to be released in a pilot version at one of the schools.

During the development of the application, the authors focused on its ease of use and clarity. For its target users, it will be an effective tool to get an overview of the book selection. For each book, a large set of information will be available, which will reduce the need to search for such details on the internet. Thanks to the integrated reading list composition feature, its users will get a quick overview of the progress in meeting their criteria. To further enhance the application's value as an information resource, the developers have introduced the ability to create analyses for books, which can even be shared among other students.

The main value of the app is the fact that no matter where students are in the exam preparation process, the app will always be there to help them through this step.

KEYWORDS

Maturita exam; digitalization; web application; education; information resource; books; authors; analyses; reading list

Obsah

1 Úvod

Součástí každé maturitní zkoušky z českého jazyka a literatury je i ústní část. K její stěžejní části patří povinný rozbor literárního díla před komisí. Každá škola má svoji vlastní nabídku literárních děl. Žáci si z ní musí na základě specifikovaných kritérií určitý počet vybrat a jejich seznam následně odevzdat vedení školy. Před komisí si v den zkoušky žák vylosuje jedno z vybraných děl a na základě výňatku koná rozbor. Většina žáků se připravuje zejména čtením vybraných knih a následným vypracováním rozborů.

Pro žáky tedy celý postup představuje, zejména ze začátku, určité zatížení. Je potřeba se zorientovat v kompletní nabídce děl a vybrat taková, která budou sedět preferencím žáka. Žáci se při výběru orientují dle určitých vlastností knih, které si ale musejí hledat sami na internetu. Chybí tedy jednotný informační zdroj, kde by měli k dispozici všechny potřebné informace na jednom místě. Spolu s výběrem žádaných knih musí dbát i na dodržení stanovených kritérií.

U některých středních škol je vidět snaha alespoň částečně tento proces zjednodušit zpřístupněním online nástroje, který po zadání vybraných knih požadavky sám kontroluje. Možnosti zefektivnění jsou ale mnohem větší.

Cílem této práce je stanovit další funkcionality, jak žákům přípravu na tuto část maturitní zkoušky zjednodušit. Výstupem práce bude webová aplikace zaměřená na práci se seznamem maturitní četby. Její základní funkcionalitou bude prezentace knih nabízených školou. Z nich bude možné skládat vlastní seznam četby, v němž se aplikace postará o automatickou kontrolu kritérií. Ke knihám budou moci žáci vytvářet přímo v prostředí naší aplikace rozbory, jež bude možné sdílet mezi ostatní uživatele. Definováním možných vylepšení bude navržena a vytvořena s dalšími nástroji a dostupnými informacemi, aby byla pro žáky během přípravy co nejvíce přínosná.

2 TEORETICKÁ ČÁST

2.1 Výzkum mezi žáky

Abychom dokázali definovat stěžejní funkcionality, které by žáci ocenili, potřebovali jsme získat názor maturantů na formu jejich přípravy a způsoby ulehčení. Rozhodli jsme se udělat krátký výzkum v našem blízkém okolí.

Formou neformálního rozhovoru se spolužáky a žáky přidružených ročníků na Střední škole informatiky a finančních služeb jsme zjistili několik zásadních informací, které pro nás budou v rámci implementace funkcí podnětné.

Pro žáky představuje největší problém právě orientace v seznamu nabízené četby. Ten je na této škole zpracován v podobě tabulky. Díla jsou zde rozdělena dle časového zařazení a u každé knihy je uveden autor a také doporučené vydání. Další informace, včetně těch důležitých pro splnění kritérií, si musí žáci dohledávat sami. To může vést k určitým nepřesnostem a rozdílům na základě použitého informačního zdroje.

Při výběru konkrétních děl do žákovského seznamu četby upřednostňují žáci knihy s určitými parametry. Značnou roli hraje počet stran a také téma knihy. Pro žáky je důležité obsahu rozumět, a knihám s určitým, třeba politickým, kontextem se záměrně vyhýbají. Mnozí jsou klidnější, pokud je děj knihy i zfilmován. Značnou výhodou je i dostupnost audioknihy.

Někteří dotazovaní uvedli, že doporučené vydání je matoucí. Stává se, že se jedná o titul dlouhodobě nedostupný ke koupi. Důležitá je tedy včasná aktualizace vydání. Velkou výhodu by prý přineslo zpřístupnění digitální verze díla.

2.2 Cíloví uživatelé

Naše aplikace je primárně cílena na žáky středních škol připravující se k maturitní zkoušce.

Aby byla aplikace pro cílové uživatele užitečná, bude potřeba každému žákovi zpřístupnit seznam dostupných literárních děl na základě jeho školy.

Aplikaci tedy nebudeme nabízet přímo žákům, nýbrž našimi zákazníky budou právě **střední školy.** Těm, které se do projektu zapojí, pomůžeme s digitalizací jejich seznamu. Také z naší strany proběhne zpřístupnění administračního prostředí, kde budou moci učitelé spravovat knihy školy, studenty a jejich seznamy. Teprve po domluvě se školou a kompletaci knih dostanou do aplikace přístup i její žáci.

Ke vstupu do aplikace bude potřeba mít založený a ověřený účet. Během registrace si žák vybere školu, následně vyplní svoji školní e-mailovou adresu a tam mu přijde ověřovací zpráva. Každá škola bude mít přiřazenou svou emailovou doménu, pod kterou se lze registrovat. Tím bude zaručeno, že žák skutečně navštěvuje zadanou školu. Například, pokud žák navštěvuje *Střední školu informatiky a finančních služeb*, ověření bude potřeba provést z emailu končícím na její doméně, tedy *@infis.cz*. Aplikace nabídne také alternativní možnosti ověření, a to skrze jednorázové registrační tokeny, jež budou moci generovat učitelé z administračního prostředí.

Klíčem k úspěchu je samozřejmě mít o knížkách co nejvíce informací, ideálně včetně dostupných digitálních verzí. O informace se musí někdo starat a udržovat je aktuální. Naše prvotní úsilí bude směřovat ke spuštění aplikace s informacemi, které do databáze vložíme sami. Jak se ale bude seznam škol rozšiřovat, už zajisté nebude jen v našich silách zajistit aktuálnost informací. Právě z toho důvodu budou mít školy k dispozici administrační prostředí, s jehož pomocí budou moci nejčastější a jednodušší úkony provádět samotní učitelé. Ti zde budou moci také vkládat k jednotlivým knihám výukové materiály, či odkazy na externí výukové zdroje.

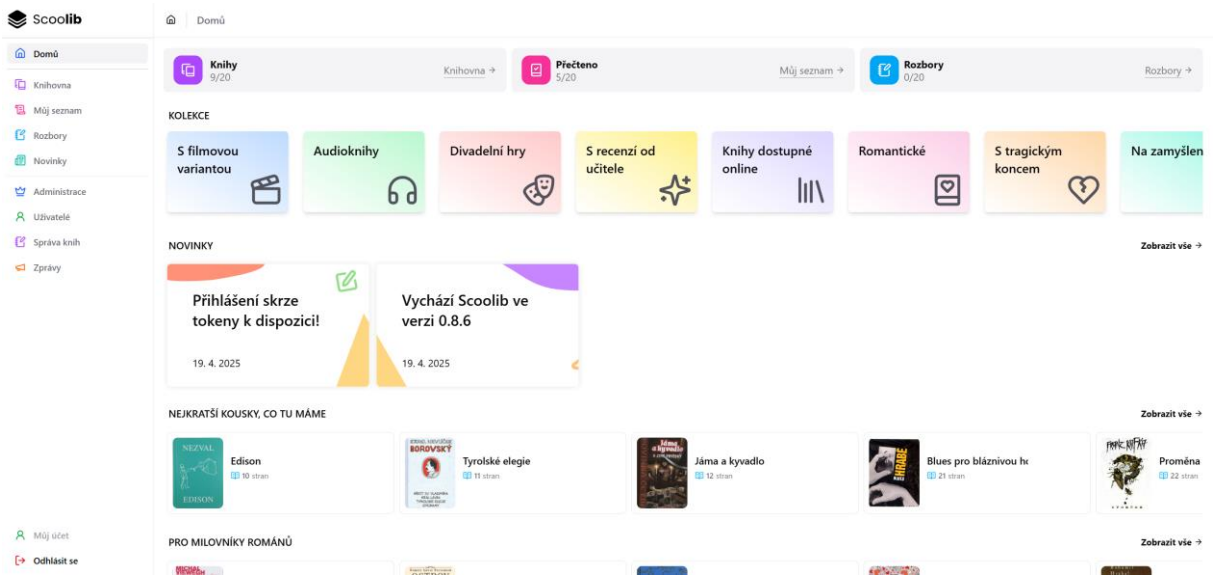
Pro pilotní provoz plánujeme aplikaci spustit na jedné ze středních škol a během krátkého období nasbírat zpětnou vazbu, tu do aplikace zapracovat, a postupně ji rozšířit do dalších středních škol.

2.3 Části aplikace

Na základě získaných informací z výzkumu jsme definovali čtyři hlavní pohledy aplikace, z nichž každý bude mít specifický účel užití. Tyto pohledy jsme následně designově navrhli a implementovali do výsledného projektu.

2.3.1 Uživatelský přehled

Tento pohled je při vstupu do aplikace výchozí. Žák zde může vidět stručný přehled svého seznamu četby – kolik knížek je již obsaženo a kolik do něj zbývá dodat. Dále v horním pruhu najde počet přečtených knížek a součet těch, ke kterým má připraven rozbor. Všechny tyto informace si může žák evidovat v dalších modulech, které budou popsány v dalších podkapitolách.



Obrázek 1: Uživatelský přehled

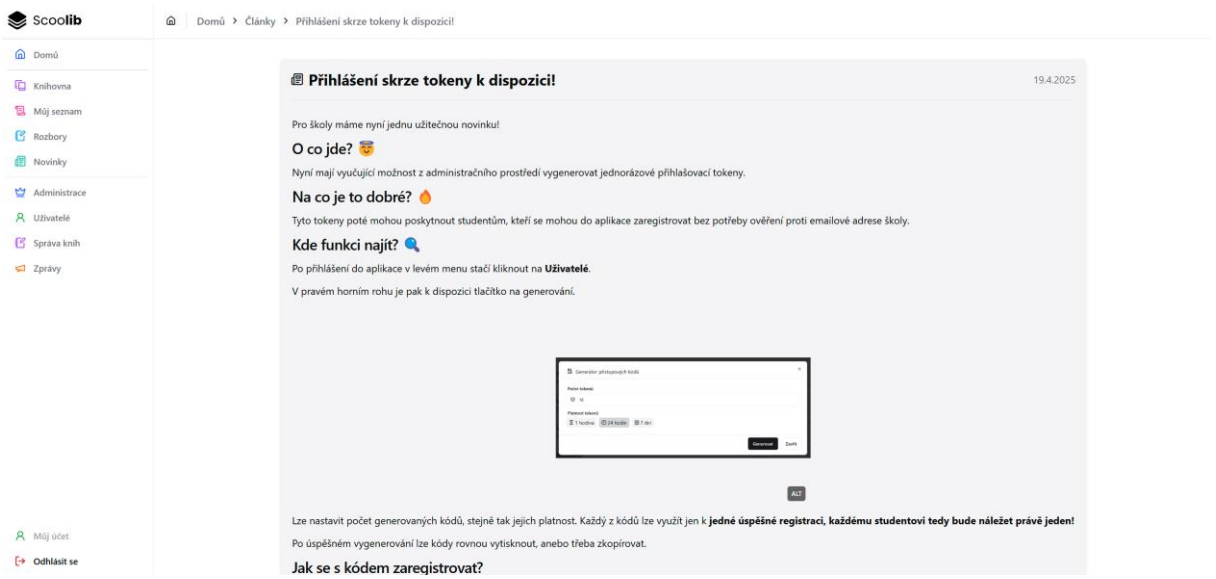
Dále zde je prezentováno také doporučení k četbě, a to prostřednictvím kolekcí. Ty budou zobrazovat knihy dle různých kritérií a řazení – např. *jen knihy s dostupnou audioknihou*. Chceme, aby naše aplikace do jisté míry fungovala jako kurátor knih. Rádi bychom tak přímo oslovili žáky a inspirovali je ke čtení a výběru určitých děl, které by je mohly zaujmout. Kolekce a jejich kritéria budou primárně nadefinovány námi – vývojáři aplikace. Do budoucna chystáme i pro samotné školy možnost si v případě zájmu vytvořit v administračním prostředí kolekce vlastní. Pro cílové uživatele jsou kolekce představovány v tomto modulu skrze dva různé typy komponent, a to pro větší diverzitu uživatelského prostředí.

Nejprve vkládáme celý jeden řádek boxů pouze s názvem a ikonou kolekce. K zobrazení obsažených knih je potřeba na box kliknout. Tento druh zobrazení je vhodný pro kolekce, jež

obsahují mnoho knih a pro snadnou orientaci je lepší rozprostřít knihy na větším prostoru – samostatném pohledu.

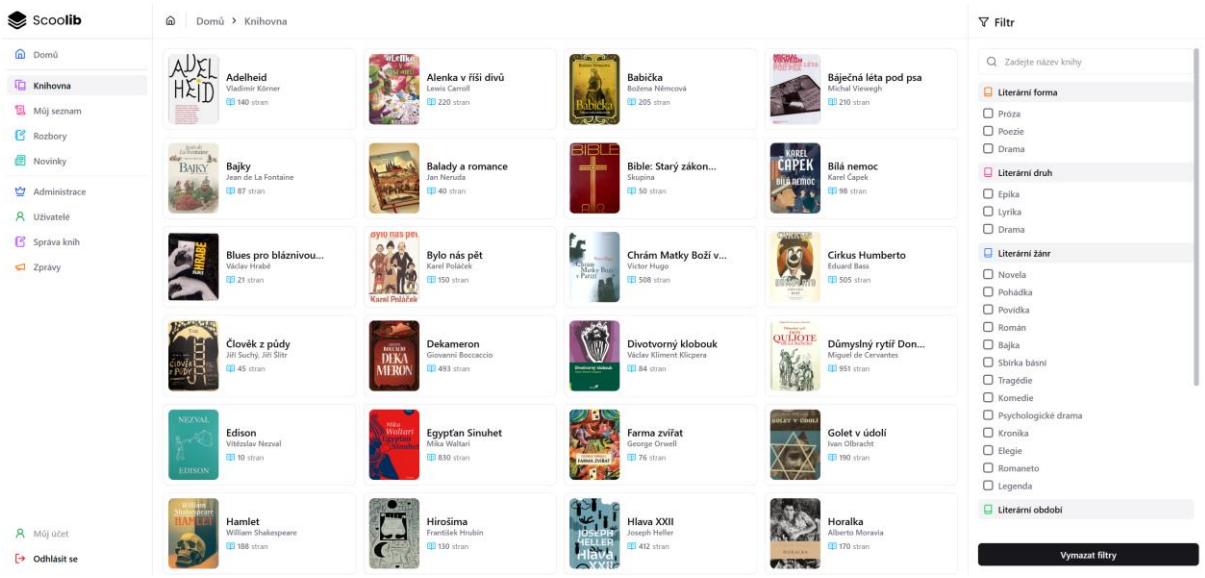
Oproti tomu druhý typ zobrazení prezentuje knihy rovnou, bez nutnosti měnit pohled v aplikaci. Funkčnost je zařízena prostřednictvím posuvného vertikálního seznamu – podobně jako je u streamovacích platforem *Netflix* nebo *Max* předkládána uživatelům nabídka videotitulů. Pro tyto kolekce je příhodné omezit maximální počet obsažených knih v nich. Prozatím u všech nastavujeme omezení na maximálně 8 titulů.

Další užitnou hodnotou přehledu je zobrazení novinek, a to jak od vývojářů aplikace, tak těch vložených školou. Po rozkliknutí dochází k otevření celého obsahu novinky. Školy mohou ke vkládání využít své administrační prostředí.



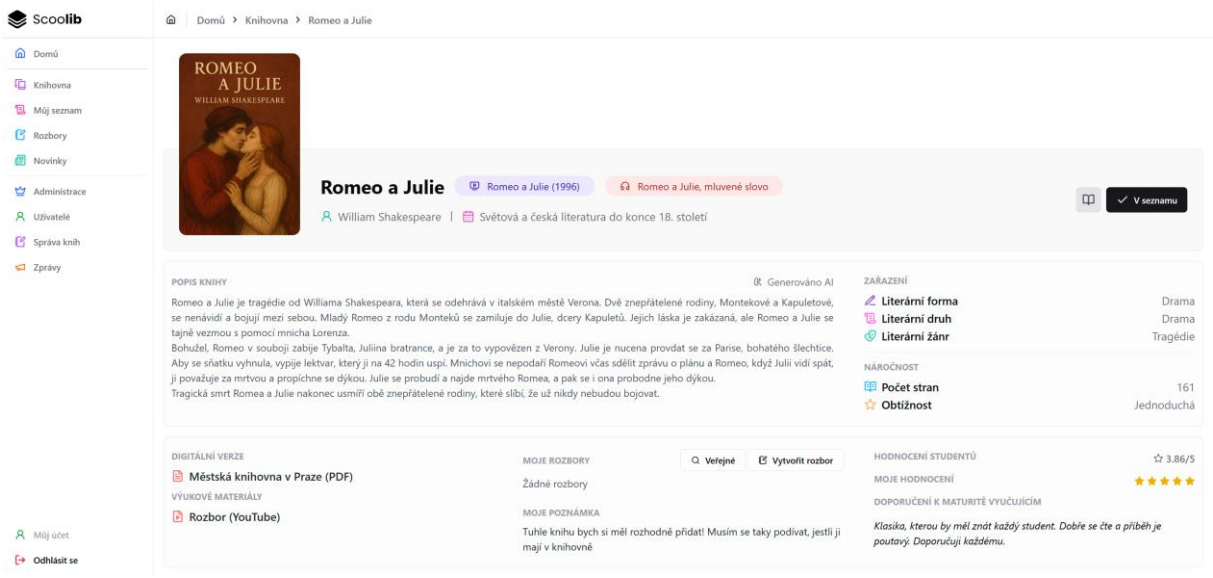
Obrázek 2: Pohled na celou novinku

2.3.2 Knihovna



Obrázek 3: Pohled na přehled knih v knihovně

Pohled knihovny obsahuje kompletní nabídku knih ze školního seznamu. Knihy jsou prezentovány formou dlaždic s jejich grafickou obálkou. Zobrazené položky lze filtrovat dle nejdůležitějších kritérií – žánru, časového zařazení a také počtu stran. Každá dlaždice s knihou jde také rozkliknout, čímž dojde k zobrazení všech informací o knize.



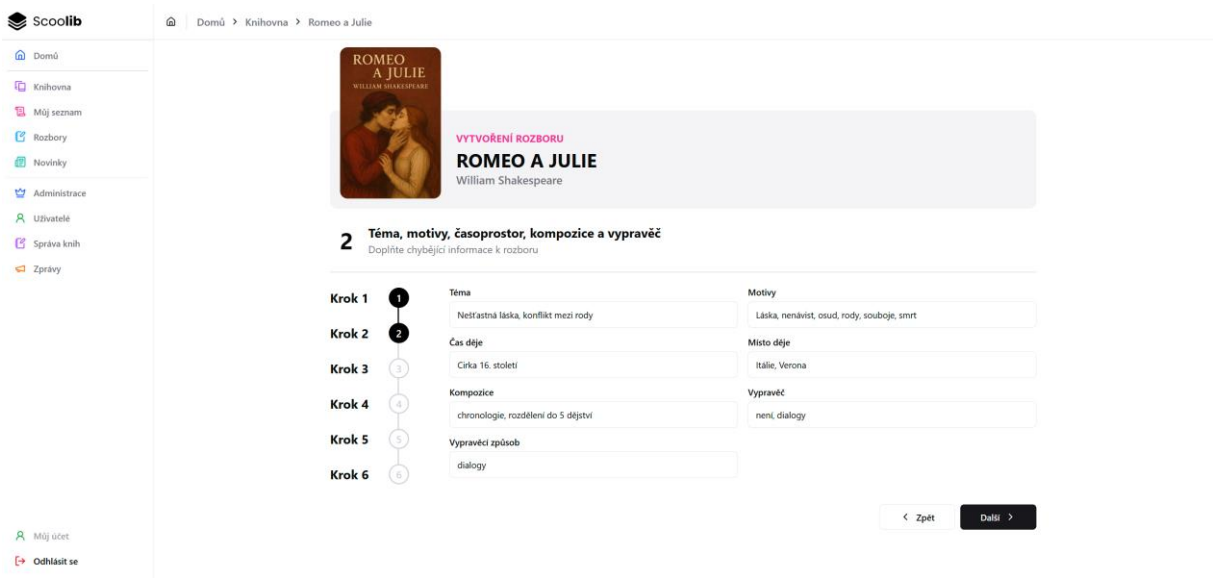
Obrázek 4: Přehled jedné knihy

Kromě názvu knihy a náhledové obálky zde uživatelé vidí informace o časovém zařazení, žánru, formě a literárního druhu knihy, dále indikátor počtu stran a také stručné shrnutí děje.

Z akčních tlačítek je možné knihu přidat na seznam četby, nebo ji označit za přečtenou. Pokud má kniha zfilmovanou podobu, či je k dispozici audiokniha, uvádíme zde odkaz.

V dolní části přehledu lze najít digitální verze díla, pokud jsou na internetu volně přístupná a šířena pod otevřenou licencí. Kromě toho jsou zde vyjmenovány výukové materiály. Ty mohou učitelé vkládat ze svého administračního prostředí, a to dvěma způsoby – prostřednictvím odkazu na externí obsah, nebo vložením souboru v podporovaném formátu.

Aplikace nabízí uživatelům možnost tvorby rozborů a jejich případné sdílení. V přehledu jednotlivých knih může žák prohlížet rozbory vytvořené ostatními a přidávat si je do své knihovny. Také je zde k dispozici tlačítko pro přechod do interaktivního editoru pro tvorbu rozboru vlastního. Tam je možné nejen využít našeho předpřipraveného průvodce tvorbou (obrázek č. 5), ale umožňujeme i nahrávat již hotové soubory z počítače, třeba ve formátu *.pdf* a *.docx*. Naše aplikace poté slouží jako úložiště takových souborů a uživateli je umožněno si je stáhnout odkudkoliv, kde bude mít k naší aplikaci přístup. Touto funkcí chceme opět přispět k našemu cíli všestrannosti Scoolibu – aby dokázal být vždy výchozím bodem pro všechny činnosti přípravy k maturitě bez potřeby využívání externích služeb.



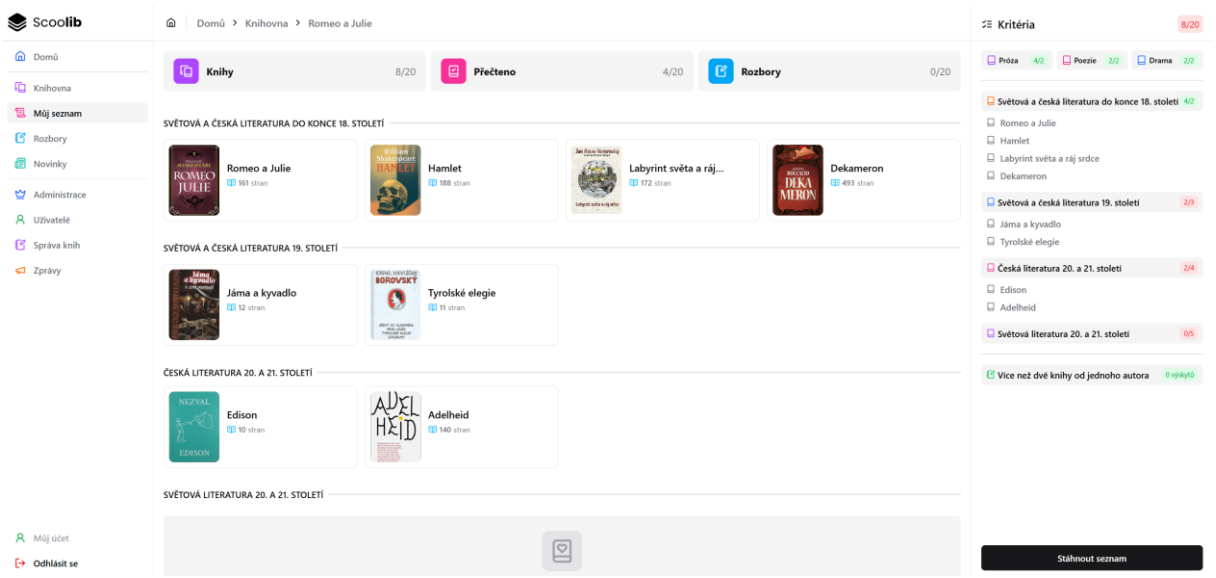
Obrázek 5: Editor pro tvorbu vlastního rozboru

V přehledu je dále ještě možnost zapsat si ke knize rychle vlastní poznámku, ohodnotit ji, či se podívat na hodnocení od ostatních studentů. Pokud učitel dané školy zadal do aplikace komentář ke knize, zobrazí se v pravém dolním rohu.

2.3.3 Můj seznam

Třetím hlavním pohledem v aplikaci je modul Můj seznam. Zde jsou vyjmenovány všechny knihy, které už má uživatel přidané v seznamu své četby. Stěžejní funkcionalitou je okamžitá

kontrola splnění požadavků pro takový seznam. Uživatel tak ihned uvidí, jaké kroky má podniknout pro správnost všech kritérií.



Obrázek 6: Žákův seznam s kontrolou kritérií

Pro zajištění jednotnosti zpracování žákovských seznamů četby je k dispozici nástroj pro jeho generování. Výstupem bude PDF soubor určený ke stažení do počítače. Žák následně může soubor poslat zodpovědným osobám ve škole, případně vytisknout a přinést do školy. Vzhled takového souboru a jeho uspořádání bude definováno striktní šablonou pro danou školu. Ve spolupráci se školou můžeme vytvořit originální šablonu přímo pro danou instituci, prozatím však musíme takový úkon udělat ručně v kódu. Využitím šablon bychom rádi stanovili jednotnou podobu seznamů četby pro školy, což zajisté usnadní orientaci v nich.

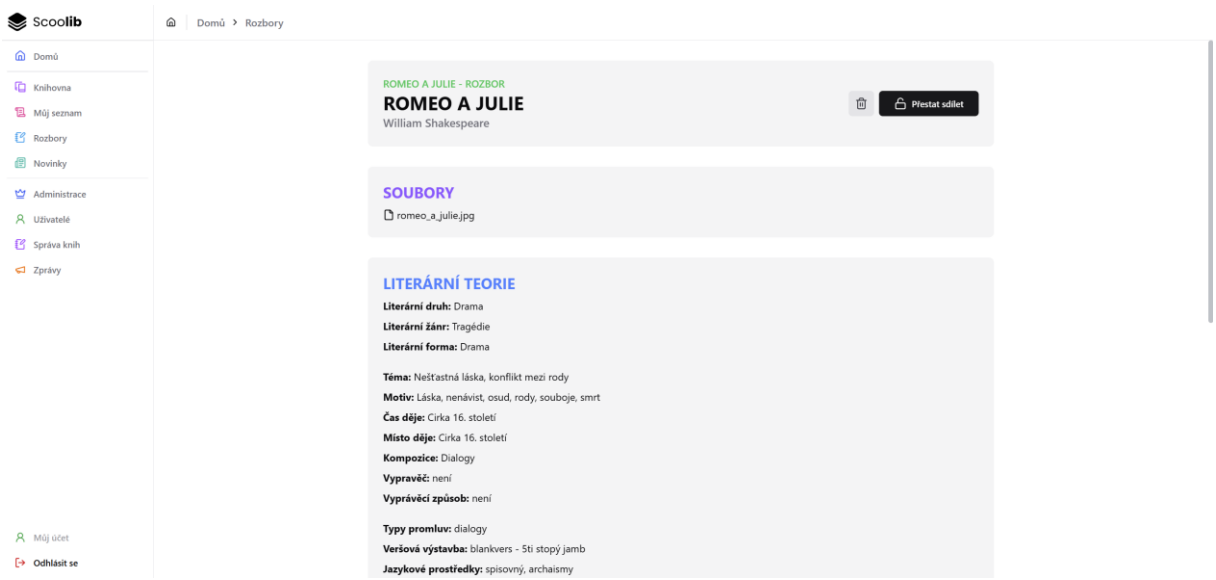
PLÁNOVANÁ FUNKCE: K seznamu četby svých žáků budou mít v budoucí verzi aplikace přístup ze svého administračního prostředí i učitelé. Budou si moci seznam prohlédnout, podívat se na stav splnění kritérií a seznam vyexportovat. Pro účely organizace maturitních zkoušek bude možný hromadný export seznamů četby, a to dokonce i u vícero žáků naráz. Aplikace bude umět výsledný soubor vytvořit v mnoha rozložení, včetně funkce zahrnutí automaticky generovaných čísel pro následný los u zkoušky.

2.3.4 Rozbory

Posledním pohledem je seznam rozborů. Zde žák vidí své rozbory a může je moci upravovat a prohlížet. Každý rozbor se vztahuje ke specifické knize a jeho tvorbu je možné vyvolat z pohledu dané knihy. Pro tvorbu rozboru je připravený vlastní interaktivní editor uvnitř naší aplikace, blíže popsaný v kapitole *2.3.2*. Zobrazení takto hotového rozboru pak probíhá do naší předpřipravené šablony.

Při tvorbě rozborů může uživatel rozhodnout, zda takový rozbor zanechat jako soukromý, nebo ho sdílet mezi ostatní žáky dané školy. V případě sdílení bude rozbor k dispozici všem pouze

pro prohlížení, nikoliv editaci. Nastavení sdílení rozboru je možné kdykoliv změnit v jeho přehledu, stejně tak soubor trvale odstranit.

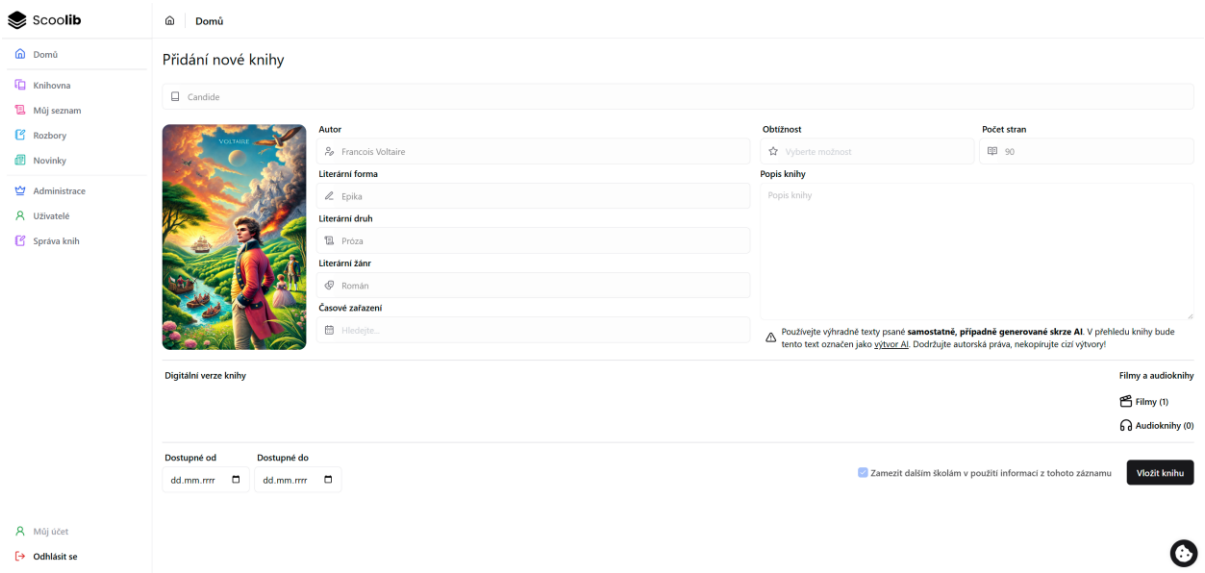


Obrázek 7: Ukázka hotového rozboru a jeho zobrazení

PLÁNOVANÁ FUNKCE: Rozbory vytvořené prostřednictvím našeho interaktivního editoru bude možné v jedné z budoucích verzích stáhnout do počítače ve formátu *.pdf*.

2.3.5 Administrace

Tento speciální pohled je k dispozici jen pro vyučující. Mohou zde spravovat knihy ve školním seznamu a v případě potřeby přidávat nové, nebo editovat existující. Dále zde také naleznou informace o všech žácích školy, včetně jejich seznamů. Seznamy četby budou moci v následujících verzích aplikace vyučující exportovat, a to i hromadně.



Obrázek 8: Proces přidání knihy skrze administraci

V aktuální verzi aplikace je tento pohled v testovacím prostředí a zajišťuje jen základní činnosti (přehled žáků, správa knih, správa informací o škole). Do dalších verzí plánujeme jeho rozšíření o další nastavení a funkce.

2.4 Název a vizuální identita

Jako název jsme pro naši aplikaci zvolili pilotně ”Scoolib”. Jde o jakousi složeninu ze slov *school*, *cool* a anglického *lib*, což je zkratka pro knihovnu. Tímto jsme tedy chtěli prezentovat, že naše aplikace je určená pro studenty a souvisí právě s knihami. Ačkoliv jsme na počátku nebyli s názvem plně ztotožněni a plánovali ještě jeho projednání a změnu, nakonec jsme se s ním v průběhu vývoje sžili a neplánujeme ho nahrazovat jiným.

Vizuální identita významně ovlivňuje přitažlivost projektu pro jeho uživatele. Je důležité umět ji přizpůsobit cílové skupině. V našem případě půjde zejména o mladé lidi. Při jejím návrhu tedy plánujeme využít minimalistický design v čele s černobílým barevným schématem, přičemž primární barvou bude bílá. Rozhraní bude definováno moderními uživatelskými prvky se střídmými animacemi. Významnou součástí designu budou ikony, které využijeme z jednotné sady pro potřebnou konzistenci.

2.5 Autorský zákon

Značnou překážkou během vývoje může představovat autorský zákon a jeho požadavky. Naše aplikace pracuje s informacemi, které musí být čerpány z existujících informačních zdrojů. Postupným zajišťováním všech zdrojů dat budeme muset řádně ošetřit, že jsou tyto zdroje pro použití bezpečné a nedochází k porušování autorských práv třetí osoby.

Autorskými právy se zabývá zákon *č. 121/2000 Sb*. Bude teda třeba pochopit jeho interpretaci autorských práv a definici děl, na která se autorský zákon přímo vztahuje.

Shromažďované informace pro funkčnost naší aplikace můžeme rozdělit do třech oblastí.

2.5.1 Textové informace o knihách

V naší aplikaci budeme zobrazovat u jednotlivých knih relevantní informace (jako třeba literární druh, obsah, nebo počet stran).

Základní definice ze zákona uvádí: „Předmětem práva autorského je dílo literární a jiné dílo umělecké a dílo vědecké, které je jedinečným výsledkem tvůrčí činnosti autora a je vyjádřeno v jakékoli objektivně vnímatelné podobě včetně podoby elektronické, trvale nebo dočasně, bez ohledu na jeho rozsah, účel nebo význam (dále jen "dílo").“ [1]

Autorský zákon chrání literární díla jako taková, ale z definice předmětů práv nelze považovat obecné informace o díle jako autorské dílo. Základní specifikace knihy lze tedy využít volně. Popis děje už by ale mohl naplňovat podstatu autorského díla, a z toho důvodu by bylo jistě bezpečnější použít k jeho tvorbě generativní technologie. Problematika autorských práv AI je popsána níže.

2.5.2 Grafické náhledy knih

V naší aplikaci bychom rádi zobrazovali u knih i jejich obálku. Na obálku knihy se ovšem vztahuje autorský zákon, a bylo by tak nutné zajistit licenci u vlastníka práv. To by bylo vzhledem k obsáhlosti a různorodosti nabídky knih nemožné.

Speciálně pro obálky knih existuje v autorském zákoně výjimka. Paragraf 37 v autorském zákoně umožňuje knihovnám a neziskovým vzdělávacím institucím použít obálku pro propagaci knihy, bez potřeby explicitně vlastnit licenci [1]. Podmínkou je uvedení autora,

pokud to možnosti dovolují. Bohužel, náš projekt není zaštiťován knihovnou, či jinou vzdělávací institucí, a z toho důvodu tuto cestu využít nemůžeme.

Nejvhodnější volbou tak zůstává generování náhledových obrázků pomocí AI.

2.5.3 Digitální verze knih

Významnou funkcí v naší aplikaci bude odkazování na digitální verze knih, pokud budou na internetu k dispozici. V tomto případě budeme hledat jen takové zdroje, které jsou licencovány některou z otevřených licencí.

Jak definuje autorský zákon v paragrafu 27, autorská práva na dílo zanikají 70 let po smrti původního autora. Pokud jde o dílo vícero autorů, rozhodujícím je rok smrti nejdéle žijícího. V případě vícero dílů/svazků se počítá tato hodnota pro každé dílo zvlášť. [1]

Jakmile zaniknou autorská práva, dílo lze označit jako *volné dílo (public domain)*. O publikaci takových děl v digitální podobě se v České republice stará třeba Městská knihovna v Praze.

V našem případě bychom tedy měli právo takové dílo přímo využít v naší aplikaci (například stáhnout a pracovat s obsahem). Prozatím ale plánujeme na zdroje digitálních verzí knih jen odkazovat.

2.5.4 Problematika autorských práv u AI

Během procesu přípravy informací do naší databáze budeme často využívat generativních technologií. Je tedy potřeba správně pochopit, jakým způsobem jsou její výtvory v dnešním světe vnímány z pohledu autorského práva.

Tvorba umělé inteligence (AI) nepochází od člověka, z podstaty věci se jí tedy nemohou autorská práva dotýkat, jak rozhodlo i soudní řízení v kauze „Thaler v. Perlmutter“ [2]

Oblíbený nástroj ChatGPT od společnosti OpenAI má naproti tomuto soudnímu řízení uvedeno v podmínkách, že vlastníkem autorských práv se po vygenerování stává uživatel.

„**Ownership of content.** As between you and OpenAI, and to the extent permitted by applicable law, you (a) retain your ownership rights in Input and (b) own the Output. We hereby assign to you all our right, title, and interest, if any, in and to Output.“ [12]

Naše zákony, stejně jako většina zákonů evropských zemích, ještě není na řešení autorských práv v souvislosti s AI připravena a přizpůsobena.

Bude jistě přibývat soudních řízení, kdy se budou podobné otázky řešit, a jistě brzy dojde i na zákonné regulace. Do té doby se ale generovaná díla nacházejí v právní nejistotě. Není jasné, zda je vlastníkem práv autor promptu, nebo se práv nemůže dožadovat nikdo.

Jasné ale je, že užitím samostatně generovaných výstupů nedochází k porušování práv třetí osoby, což je pro nás nejhlavnější fakt, a díky tomu je můžeme bez obav využít.

2.6 Přínos aplikace

2.6.1 Přínos pro uživatele

Cíloví učitelé, jimiž jsou žáci středních škol, získají díky naší aplikaci jednotné místo, kde hledat informace o maturitní četbě bez ohledu na to, v jaké části přípravného procesu se nacházejí. Aplikace je koncipována tak, aby nabídla užitnou hodnotu pro všechny typy studentů. Nezáleží, zda si chce daný žák jen rychle svůj seznam poskládat, anebo se důkladně věnovat přípravě přečtením všech knih – pro všechny je v naší aplikaci místo. Díky jednoduchému prostředí a lákavé prezentaci knih lze očekávat, že dokážeme studenty více motivovat k četbě a kvalitnější přípravě, a to díky sjednocení všech potřebných informací na jedno místo. Zpřístupňování digitální verze děl vnímáme jako významný krok k podpoření zájmu studentů o čtení děl. Nemusí totiž knihu složitě shánět, ale mají ji k dispozici na jedno kliknutí.

2.6.2 Přínos pro školy

Správce školy může skrze aplikaci vytvořit účty pro své vyučující. Ti poté mají po přihlášení k dispozici administrační prostředí, jež nabízí několik užitečných funckionalit. Lze zde spravovat literární díla – knihy přidávat, odebírat, či editovat. Ke každé knize lze přidávat výukové materiály, které se poté zobrazí v aplikaci studentům. Dále je možné vydávat oznámení, které se též ukáže studentům v jejich klientském pohledu. Učitelé mají díky administračnímu prostředí také přehled nad seznamy jednotlivých žáků a mohou v případě zájmu kontrolovat jejich správnost. Přítomna bude také možnost pro export jak jednotlivých seznamů, tak hromadných přehledů přímo k účelům použití pro komisi u maturitní zkoušky.

Aplikaci je možné zakomponovat přímo do výuky literatury. Učitelé mohou aplikaci a díla v ní obsažená používat při výkladu jako podpůrnou pomůcku. Díky možnosti evidence učebních materiálů je pak jednoduše při hodinách najdou rychle na jednom místě, nehledě na fakt, že je tak rovnou zpřístupní i samotným studentům, kteří v jejich vypracování mohou pokračovat třeba po skončení výuky.

2.7 Praktická ukázka

Aplikace bude pro všechny školy a její uživatele k dispozici na jednotné webové adrese v podobě webové aplikace. Odkaz je následující: https://scoolib.cz/.

Pro kompletní přístup do aplikace je potřeba mít ověřený uživatelský účet spárovaný s určitou školou.

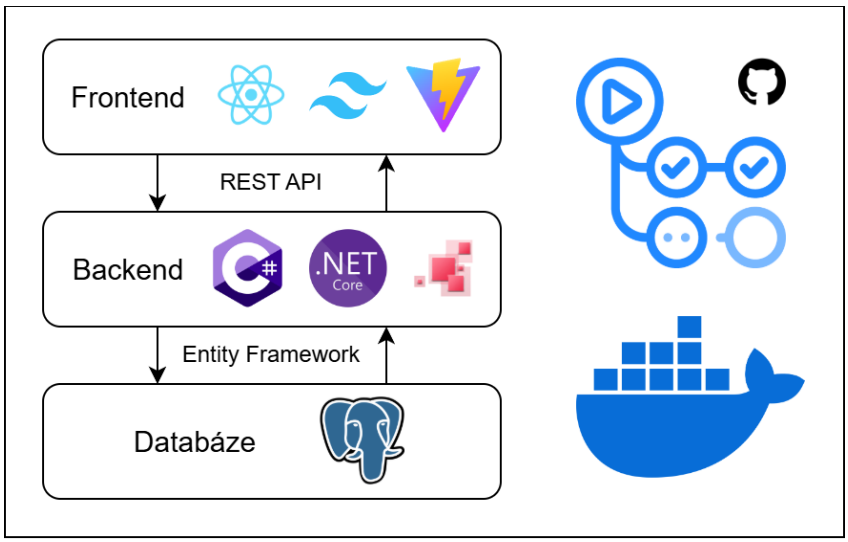
Registrace je nyní omezena jen na studenty vybraných škol, k přístupu lze využít vytvořený testovací účet s následujícími údaji:

- uživatelský email: **scoolib@adamhojer.cz**
- heslo: **Scoolib2024**

Aplikace je stále v průběhu vývoje. Testovací účet obsahuje přístup do testovací verze. Funkce obsažené v aplikaci a dostupná díla se mohou lišit dle použitého účtu k přihlášení.

3 PRAKTICKÁ ČÁST – VÝVOJ APLIKACE

Vývoj naší aplikace zahrnuje práci na třech hlavních pilířích – databázi, backendu a frontendu. Součástí vývoje bude i zajištění zpřístupnění webové aplikace na internetu.



Obrázek 9: Schéma aplikace s vybranými použitými technologiemi

Každá webová stránka má svou strukturu, styly a funkcionalitu. Této části říkáme frontend. Je to ta část, kterou cíloví uživatelé při návštěvě stránky vidí a se kterou interagují.

Webové stránky ovšem na určitých místech zobrazují odlišné informace na základě určitých vstupů (může jít třeba o přihlášeného uživatele, vybraný produkt na e-shopu, nebo třeba polohu). Aby bylo možné dosáhnout odlišného obsahu a chování webových stránek, je k dispozici backend. Ten je schopen do jisté míry, ovlivnit obsah a vzhled stránky dodáním relevantních dat.

Webové stránky zobrazují mnoho informací, které musí vývojáři aplikace dlouhodobě ukládat. K takovým účelům slouží databáze.

Aby bylo možné aplikaci vystavit na internet, je potřeba provést její nasazení na server – počítač přístupný z internetu.

V průběhu práce budou použity termíny *klient* a *server*. V našem kontextu *klient* zasílá požadavky, *server* tyto požadavky zpracovává a vrací výsledky. Jde v podstatě o podobnou analogii jako srovnání frontendu a backendu.

Všechny tyto části vývoje si přímo na aplikaci Scoolib popíšeme v jednotlivých podkapitolách.

3.1 Databáze

Databáze slouží pro dlouhodobé ukládání dat aplikace. Databázi spravuje její engine – kus softwaru. Ten nejenže zajišťuje přístup k databázi, ale zároveň nabízí i metody pro efektivní práci s jejími daty – může nativně podporovat například hledání v datech, jejich filtrování atd.

Databáze můžeme rozdělit na *relační* a *nerelační*.

Relační databáze ukládají data do tabulek, každý nový záznam znamená jeden řádek tabulky. Schéma je pevné a nemění se, tabulka má jasně definované sloupce a každý z nich pracuje s určitým datovým typem. Relační databáze nativně podporují práci s relacemi, data tedy na sobě mohou vzájemně záviset, odkazovat na sebe. Relační databáze používají pro přístup a práci s daty vlastní jazyk *SQL*. Populárními zástupci této kategorie je například *MySQL*, *Microsoft SQL*, *PostgreSQL*.

Nerelační databáze nemají pro ukládaná data jasně danou strukturu a nemusí být striktně uspořádány. Zatímco u relačních databází je možné škálovat kapacitu vertikálně (přidáváním řádků tabulky), nerelační databáze jsou škálovatelné horizontálně, protože postrádají strukturu tabulky. Data mohou nabývat různých formátů. Pro přístup k datům se používají různé metody, schémata či jazyky. Typickým příkladem nerelačních databází je *MongoDB*, *AstraDB*, *ScyllaDB*.

3.1.1 Výběr databáze

Pro všechna data používaná naší aplikací dokážeme definovat jejich jasnou strukturu, zároveň bude mezi informacemi vznikat mnoho vazeb, takže pro nás byla logickou volbou některá z relačních databází.

Při výběru konkrétního databázového řešení jsme se soustředili na několik kritérií. Stěžejní pro nás byla kompatibilita s *Entity Framework*, včetně podpory LINQ a práce s relacemi. Tyto přístupy jsme totiž plánovali využít při tvorbě backendu, bylo tedy důležité těmto požadavkům přizpůsobit už výběr databáze. Důležitým aspektem byla také možnost snadného nasazení na systém GNU/Linux, na kterém poběží náš produkční server. Ideálním kandidátem se jevilo řešení *Microsoft SQL*, to ale bohužel není open-source[1] a jeho bezplatná verze *Express* byla velmi omezující. Proto nakonec volba padla na *PostgreSQL*.

3.1.2 Nasazení databáze

Databázi PostgreSQL jsme nainstalovali na náš server pomocí oficiálního návodu. Aby byla databáze přístupná pro naše lokální vývojové prostředí a také pro databázové klienty určené pro

[1] Software s otevřeným zdrojovým kódem, jeho užití je zpravidla bezplatné

správu databáze, upravili jsme konfiguraci PostgreSQL a aktualizovali nastavení firewallu[2]. Díky tomu byla nyní databáze přístupná i mimo server, z internetu.

3.1.3 Práce s databází

K práci s databází (prohlížení a editace dat) jsme používali software DataGrip od společnosti JetBrains. Aplikace samotná poté s databází pracuje pomocí *Entity frameworku*, který popíšeme v kapitole *Backend*.

[2] Software řídící síťovou komunikaci

3.2 Backend

Backend je velmi široký pojem, který může mít mnoho podob a forem. Základním úkolem backendu v kontextu aplikací je většinou práce s daty a jejich zajišťování frontendu[3]. Má tedy v gesci zpravidla i hlavní funkcionalitu aplikace, kterou u sebe provádí z bezpečnostních, či výkonnostních důvodů. Backend část webové aplikace běží zpravidla na vyhrazeném serveru a cílový uživatel s ní nepřijde přímo do styku, pomyslným mostem mezi uživatelem a backendem je tedy frontend popsaný v kapitole *3.3*.

3.2.1 Výběr technologie pro backend

Způsobů, jak k aplikaci napsat potřebnou backendovou část, je dnes celá řada. V závislosti na použitém přístupu se výrazně mění i architektura celé aplikace a způsob implementace frontendu.

Při výběru byl v našem případě klíčovým parametrem způsob, jakým je backend funkčně s frontendem spjat. Konkrétně nás zajímalo, jestli se backend stará o funkcionalitu samotnou, anebo s funkcionalitou stránky rovnou i vykresluje. Popišme si tyto dva odlišné scénáře.

Pokud se backend stará i o vykreslení stránky, říkáme tomuto jevu SSR (server-side rendering). V praxi to znamená, že jakmile uživatel zažádá o webovou stránku, server přijme požadavek, provede potřebné úkony (třeba získání dat z databáze), ty rovnou vloží do HTML [4]struktury, a na klientské zařízení putuje zpět již kompletní webová stránka, kterou prohlížeč jen zobrazí. Tento přístup má zajisté výhodu v rychlosti načítání webu, protože odpadá nutnost dodatečné manipulace s obsahem na zařízení klienta. Také je zpravidla o něco bezpečnější, protože programátor zde nemá takových možností vystavit důvěrné informace mimo server. Na druhou stranu, kód je v těchto případech velmi komplexní, někdy i nekonzistentní, a použité backend technologie často limitují možnosti samotného frontendu. Na zařízení cílového uživatele je omezena možnost interaktivity, protože zásadní změny ve struktuře webu je třeba řešit serverovým požadavkem na nový kód stránky. Populárním zastáncem své doby byl *ASP.NET Web Forms*.

V poslední době je stále oblíbenější volbou backend od frontendu separovat. Backend se tedy o frontend vůbec nestará, jen předloží data v určité datové struktuře, a zpracování je na frontendu samotném. V praxi se na cílové zařízení uživatele nejdříve načte frontend část aplikace (tedy struktura webu, styly a frontend funkcionalita). Po načtení základní struktury teprve probíhá dotazování backendu na potřebná dynamická data, která se po obdržení na stránku doplní. Na efektivitu tohoto řešení je možné pohlížet ze dvou pohledů. Jelikož nejdříve dochází k načtení a zobrazení základní struktury a až posléze se doplňují dynamická data, je časový interval potřebný pro načtení kompletní webové stránky skutečně delší. Na druhou stranu je ale uživateli předložena základní struktura bez čekání na funkcionální úkony, takže

[3] Část aplikace sloužící k reprezentaci dat cílovému uživateli
[4] Značkovací jazyk, který reprezentuje webové stránky

prvotní interakce s webem může proběhnout dříve. Získávání potřebných dat je možné rozdělit na vícero menších požadavků, takže data se poté doplňují nezávisle na sobě. Velkou výhodou použití tohoto přístupu je nezávislost na podobě frontendu. Nezáleží, zdali data vyžaduje web, mobilní aplikace, nebo třeba IoT zařízení[5], což činí tento přístup univerzálním. Pro tento typ backendu se často využívá například *Express.js* nebo *ASP.NET Core* implementující Web API funkcionalitu.

Druhý zmíněný přístup v poslední době nalézá stále více využití. Mnoho významných společností staví své webové aplikace tak, aby prvně proběhlo vykreslení základní struktury a další data se načetla posléze.

Samozřejmě, je častou praxí oba přístupy kombinovat, a využívat tak výhody z obou táborů, toho užívá třeba i populární framework[6] *Next.js*. To by ale pro naše účely bylo zbytečné. Abychom měli nad každou částí aplikace kompletní dohled, zvolili jsme nakonec metodu odděleného backend a frontend prostředí. Backend tedy bude v našem případě jen zpracovávat a připravovat data.

Technologií a jazyků, které bychom mohli pro takovýto separátní backend použít, je celá řada. Od starších možností jako je PHP, až po modernější řešení v čele s jazyky Go, Python nebo JavaScript.

Po detailním prozkoumání nabídky nás nejvíce zaujalo využití jazyka C#, a konkrétně jeho frameworku *ASP.NET Core*.

ASP.NET Core je open-source webový framework vyvinutý společností Microsoft Corporation. Je spustitelný pod všemi hlavními desktopovými platformami, to znamená Windows, Mac OS X a GNU/Linux. Zahrnuje podporu technologií *Razor* a *Blazor*. Razor slouží pro server-side rendering webových stránek, Blazor k tomu přidává i možnost psaní frontend funkcionality skrze C#. [11]

3.2.2 Způsob implementace (API)

Jak už bylo zmíněno, backend bude fungovat separátně od frontendu. Frontend bude zasílat požadavky na získání nebo manipulaci s daty, a backend bude po provedení náležitých procedur odpovídat výsledkem. Pro definování konkrétní podoby takových výměn informací se využívá

[5] Zařízení chytré domácnosti
[6] Sada nástrojů rozšiřující možnosti programovacího jazyka

pojem *API*. API je v kontextu webových aplikací souhrn pravidel, kterými se řídí vývojář při implementaci výměny dat.

Pro definici takových pravidel se zpravidla využívají už existující rozhraní, jedním z nejvyužívanějších je *REST API*. To jsme ve větší míře použili i my, řídili jsme se tedy některými jeho pravidly [13].

REST API reprezentuje každý datový zdroj backendu jedinečným identifikátorem (URI). V kontextu webových aplikací je tak většinou každý přístupový bod definován vlastní URL adresou, např. *http://localhost/api/users/*. REST API má vlastní metodiku pro skládání takových URL adres, dle ní jsme se ale spíše neřídili, protože pro nás práce s adresami dle pravidel REST nebyla pohodlná.

Pravidla dále stanovují, že pro definici konkrétní akce s daty by měli vývojáři využívat *CRUD* operace. Protokol *http*, který se stará o přenos dat po internetu, má definované metody jako GET, POST, PUT, UPDATE, DELETE aj. Implementací těchto metod a jejich použitím jsou jasně stanovené dopady dotazů. Tento přístup při vývoji využíváme.

V rámci *http* protokolu je také doporučeno správně využívat jeho návratové typy. Kromě případných dat se totiž s každou odpovědí na požadavek vrací i stavový kód reprezentovaný trojicí čísel – např. 200 OK, 400 Bad Request, 500 Server Error. Toto označení jasně vyjadřuje stav dotazu a případně dokáže specifikovat důvod chyby a její zařazení. Tento přístup také využíváme.

Klíčovou součástí definice REST je také užití HATEOAS (Hypermedia as the Engine of Application State). Toto pravidlo uvádí, že součástí odpovědi na požadavek by správně měly být i odkazy na dostupné akce s daným prostředkem. Pokud bychom například vznesli dotaz na konkrétní knihu, kromě informací by měl backend vrátit i URL adresy akcí přidružených k ní (např. přidání do seznamu četby, označení za přečtenou aj.). Dokonce by měly být už samotné akce filtrovány a vráceny jen ty dostupné – např. pokud kniha už bude v uživatelově seznamu, součástí odkazů by byl pouze endpoint na její odebrání ze seznamu. Tento přístup prozatím není v komunitě vývojářů hojně implementován, a ani my jsme neměli důvod ho zahrnovat do naší aplikace.

REST API je definice nezávislá na využité platformě, neuvádí tedy přesně datovou strukturu, v níž má být přenos zajištěn. My jsme si pro naše účely vybrali JSON, jelikož jde o přehlednou formu reprezentace dat a pracuje se s ní pohodlně ve všech hlavních programovacích jazycích.

3.2.3 Příprava projektu a jeho struktura

Při tvorbě projektu ve frameworku *ASP.NET Core* má vývojář mnoho možností, jak bude jeho aplikace fungovat. Pokud se zaměříme na aplikace webové, což byl i náš případ, měli jsme

možnost využít technologie Razor, Blazor, nebo napsat aplikaci s Web API. Vybrali jsme Web API, protože Razor i Blazor pracují s frontendem v backend kódu, což pro nás nebylo účelné.

Každý požadavek je na serveru definován určitou funkcionalitou, sledem instrukcí. Web API má dva způsoby zápisu jednotlivých požadavků, a to buď skrze *kontrolery*, anebo *Minimal API*. Přístup skrze kontrolery vychází ze starých standardů, zatímco práce s *Minimal API* je efektivní, protože zápis se snaží být co nejvíce abstraktní a zjednodušený. Vybrali jsme si tedy *Minimal API*, protože jde mj. o modernější způsob zápisu [3]

Struktura našeho backend projektu vypadá následovně.

```
.
├── Context/                        databázové kontexty aplikace
├── Endpoints/              funkcionalita jednotlivých požadavků
├── Enums/                                        enum objekty
├── Migrations/                       migrace tvořené EF Core
├── Models/                                      datové modely
├── Resources/                      mediální soubory aplikace
├── Services/               pomocné třídy pro speciální činnosti
├── appsettings.json                        nastavení aplikace
├── Dockerfile                   konfigurační soubor pro Docker
└── Program.cs                             vstupní bod aplikace
```

3.2.4 Definice endpointu

Koncový bod API se nazývá endpoint. Za ním se skrývá určitá funkcionalita, která by měla vrátit zpět odpověď, ideálně určitá data.

V architektuře naší aplikace rozlišujeme endpointy pro jednotlivé druhy objektů, se kterými pracujeme. Máme samostatné endpointy pro knihy, seznam četby, uživatele, rozbory atd. Každou takovou skupinu endpointů máme v samostatné třídě. Třída je statická a obsahuje vždy jedinou statickou metodu pro registraci takových endpointů. Ta přijímá jako parametr interface *IEndpointRouteBuilder*, nad kterým se poté registrují jednotlivé endpointy. Parametr je vložen pomocí *dependency injection*. Ta zajišťuje, že instance takového objektu je dostupná globálně v celé aplikaci a při použití jako parametru je v metodě dostupná bez potřeby definovat objekt při volání metody (jak bude vidět v ukázce kódu č. 2)

Jednoduchý endpoint vracející textový řetězec ze vstupu, je pro ukázku definován v souboru *Status.cs* ve složce *Endpoints* (kód č. 1). Každý jednotlivý endpoint definujeme metodou *MapGet()*, resp. její alternativou v závislosti na použité http metodě. Jako první parametr přebírá metoda cestu k samotnému endpointu ve formě textového řetězce. V našem případě uvádíme vždy jeho koncovou část, zbytek cesty definujeme skrze metody *MapGroup()* výše v hierarchii. Jako druhý parametr je třeba vložit metodu s funkcionalitou endpointu, v našem

případě vytváříme takovou metodu přímo uvnitř parametru skrze *lambda*[7] výraz, tedy anonymní[8] vnořenou funkci. Jako parametry funkce zadáme vstupní data, která jsou připojena k požadavku (např. může jít o data z registračního formuláře, nebo ID žádané knihy) a naopak navrácená data z pravé strany výrazu API vrátí zpět ke klientovi.

```
public static class Status
{
    public static void RegisterStatusProcedures(this IEndpointRouteBuilder routes)
    {
        routes.MapGet("/returnText", (String in) => Results.Text('Text: '+in));
    }
}
```

Kód 1: Ukázka třídy pro registraci endpointů

Ke zprovoznění musí ještě proběhnout registrace celé skupiny ve vstupním bodu aplikace, souboru Program.cs (kód č. 2)

```
app.MapGroup("/status").WithTags("Status").RegisterStatusProcedures();
```

Kód 2: Registrace jednotlivých tříd v Program.cs

3.2.5 Práce s databází

Aby naše aplikace mohla pracovat s databází, využíváme řešení *Entity Framework*. Jde o nástroj od společnosti Microsoft Corporation, který zajišťuje komunikaci s databází. Vývojáři při práci s databází nepoužívají její nativní jazyk (v našem případě SQL), ale volají určité metody nad Entity Frameworkem, který se poté o překlad do příkazu pro databázi postará. Jde o velmi efektivní řešení, protože se tím zvyšuje míra abstrakce a lze jasněji zapsat, jaká data potřebujeme, a to přímo v syntaxi C#.

Entity Framework, mimo jiné, spravuje schéma databáze a předává jej do projektu. Vývojáři poté mohou při psaní kódu přistupovat ke konkrétním databázovým objektům a mají přehled o jejich uspořádání. Nemůže se tak stát, že by nevědomě přistupovali k neexistující entitě, nebo pracovali se špatnými datovými typy. K dosažení synchronizace stavu databáze s modely nabízí Entity Framework (dále jen EF) dva přístupy: *Code-first* a *Databse-first* [4]. První z nich, Code-first, je modernější. Vývojář v tomto případě definuje datovou strukturu databáze ve svém kódu. EF se následně postará o její aplikaci do databáze skrze tzv. *migrace*. Oproti tomu

[7] Zápis anonymních funkcí, odleva od operátoru '=>' jsou uvedené parametry, na pravé straně pak funkcionalita
[8] Funkce definována beze jména, umožňuje okamžité užití na konkrétním místě

Database-first se využívá, pokud už je databáze hotová a teprve poté se tvoří samotná aplikace. V tom případě si EF namapuje její strukturu a vytvoří odpovídající datové modely v projektu.

Jelikož jsme aplikaci i databázi tvořili naráz, vybrali jsme modernější způsob *Code-first*.

Provedli jsme počáteční nastavení spojení s databází v souboru *Program.cs* a následně bylo potřeba vytvořit databázový kontext. Databázový kontext reprezentuje v projektu právě jednu databázi a její obsah. Definuje také některé úkony při práci s databází. Skrze kontext vývojáři k datům v databázi přistupují.

Využíváme jen jednu databázi, takže jsme vytvořili *ScoolibContext* třídu, kterou jsme i zaregistrovali v *Program.cs*.

Každá tabulka v databázi je v kontextu reprezentovaná svým *DbSet* objektem (kód 3), její sloupce (resp. poté i struktura každého řádku) jsou definovány samostatnou třídou (kód 4).

```
public partial class ScoolibContext {

       …

       public virtual DbSet<Book> Books { get; set; }

       …

}
```

Kód 3: Definice tabulky skrze DbSet

```
public class Book
{
    [Key, DatabaseGenerated(DatabaseGeneratedOption.Identity)]
    [SwaggerSchema(ReadOnly = true)]
    public int Id { get; set; }
    [MaxLength(100)]
    public string? Author { get; set; }
    …

}
```

Kód 4: Definice sloupců tabulky Books

Jakmile máme takové schéma vytvořené, propíšeme ho do databáze pomocí vytvořením migrace, EF se sám postará o adekvátní aktualizaci.

K datům z databáze poté v těle samotných endpointů přistupujeme skrze vytvořený databázový kontext (*ScoolibContext)*, nad kterým definujeme kolekci DbSet (*Books)* a následně využíváme metod EF. V ukázce z každého záznamu v tabulce vytvoříme nový objekt jen s vybranými

vlastnostmi (Select). Celý výsledek poté převedeme do Listu, aby s ním bylo možné pracovat jako s polem. Parametr *ScoolibContext* opět využívá funkcionality dependency injection.

EF převede tuto reprezentaci do SQL příkazu, který následně pošle na databázový server.

```
public static class Books
{
    public static void RegisterBooksProcedures(this IEndpointRouteBuilder routes)
    {

       routes.MapGet("/all/simple", (ScoolibContext db) => db.Books.Select(s => new
       { s.Name, s.Author, s.Thumbnail }).ToList());

    }

}
```

Kód 5: Přístup k databázi v těle endpointu

V databázové struktuře často používáme relace pro vyjádření vztahu mezi různými entitami (například každý uživatel má přiřazenou konkrétní školu). Tento vztah je nejdříve nutné definovat v modelu. V kontextu databází existují různé varianty vztahů, dle nich se také liší implementace.

Nejvyužívanější v naší aplikaci je relace *1:N*. Ta vyjadřuje, že k jednomu záznamu je přiřazeno více záznamů jiné entity. V naší aplikaci je takovým případem třeba entita knihy, k jejíž záznamu může být přiřazeno vícero filmů.

Musíme změnit model knihy i filmu (ukázka kódu č. 7). U knihy uvádíme novou vlastnost *Films*, list filmů. V modelu filmu uvádíme vlastnosti *BookId* a *Book*, ale nevyužíváme je. Slouží nám pro spárování vlastností při incializaci *ScoolibContext*. Relaci je ještě nutné definovat v nastavení konkrétního kontextu (ukázka kódu č. 6). Pro získání seznamu filmů nám tímto odpadá nutnost dalšího dotazu do databáze. Pokud v těle endpointu využijeme funkci *Include()*,

EF pak vloží do SQL dotazu adekvátní *JOIN* příkaz sloužící právě pro získání navázaných dat z relací, vše se provede v jednom příkazu.

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Book>().HasMany(b => b.Films).WithOne(f =>
f.Book).HasForeignKey(f => f.BookId).IsRequired(false);
}
```

Kód 6: Příprava relace při incializaci kontextu

```
public class Book
{
    …
    public ICollection<BookFilms> Films { get; } = new List<BookFilms>();
    …
}


public class BookFilms
{
    …
    [JsonIgnore]
    public int? BookId { get; set; }
    [JsonIgnore]
    public Book? Book { get; set; }
    …
}
```

Kód 7: Přidání vlastnosti School v modelu uživatele

3.2.6 Uživatelské účty a jejich relace

Naše aplikace z podstaty fungování vyžaduje funkcionalitu uživatelských účtů. Mnoho informací se totiž vztahuje k jednotlivým uživatelům, jako je například seznam vybraných knih. Přítomností účtů je také zajištěn přístup pouze k datům týkajících se konkrétní školy, na které uživatel studuje. Účty tedy hrají zásadní roli, a je třeba je, spolu s metodami pro jejich správu, implementovat.

Pro tyto účely jsme použili knihovnu *Microsoft Identity* od společnosti Microsoft Corporation. Ta je napsána v jazyce C# a poskytuje obrovskou míru abstrakce nad implementací funkcionality právě uživatelských účtů. Sama v pozadí spravuje samotná data, uživatelské relace, i tokeny, a pro práci s jejich stavem poskytuje několik metod. Její užití nám ušetřilo

mnoho času, který by se jinak musel věnovat detailnímu studiu fungování takových relací, a zejména zajištění jejich správného zabezpečení.

Knihovna Microsoft Identity není ve výchozím nastavení součástí projektu, proto jsme ji doinstalovali skrze balíček NuGet[9] a nakonfigurovali její základní parametry v souboru *Program.cs*.

Pro události související s uživatelskými účty jsme vytvořili samostatnou třídu *Auth*. Ta je zpřístupňována v rámci celé aplikace jako *dependency injection*, bylo tedy nutné třídu registrovat jako službu v *Program.cs*.

Jednotlivé metody třídy *Auth* jsou volány obvykle v jednotlivých endpointech. Jejich funkcionalita tedy mohla být implementována rovnou do těla endpointů, avšak kvůli obsáhlosti některých procedur jsme se rozhodli pro oddělení do samostatné třídy. Navíc jsou metody definované zde použitelné univerzálně a mnohdy například dochází k jejím volání z jiných míst aplikace, definice skrze endpoint by tedy nedávala příliš smysl.

V případě registrace nového uživatele definujeme ve třídě *Auth* metodu *RegisterNewUser*, která přebírá jako parametry objekt *user* – obsahující údaje z registračního formuláře – a také *db* pro přístup k databázi (ukázka č. 9). Nejdříve dojde k vytvoření objektu typu *User*, se kterým už dokáže Microsoft Identity pracovat. Následně dochází k ověření, zdali je vybrána nějaká škola, a jestli zadaný email odpovídá požadavkům dané školy. Pokud je vše splněno, dochází už skrze metodu *CreateAsync* k samotné tvorbě profilu. EF v pozadí zajistí zapsání nového uživatele do databáze a vygenerování potřebných tokenů. Pokud se vše povedlo, dojde k zavolání metody pro zaslání ověřovacího emailu. Vždy dojde na vrácení výsledku operace.

Výsledek operace vrací knihovna Microsoft Identity ve svém standardizovaném objektu *IdentityResult*. Tato třída se používá pro popis výsledků většiny procedur knihovny, výjimkou je však přihlášení. Při pokusu o přihlášení se totiž vrací objekt typu *SignInResult*, nikoliv *IdentityResult*. Jelikož jsme chtěli předejít nekonzistentnosti kódu při zpracovávání takových výsledků na frontendu, potřebovali jsme jednotný formát odpovědí. Zároveň jsme potřebovali strukturu výsledků doplnit o parametr *ModuleType*, který nám definuje, v jakém konkrétním údaji nastala chyba (např. chyba v emailové adrese).

Z toho důvodu jsme si vytvořili vlastní třídu *UserEndpointResult*, jež obsahuje seznam chyb prezentovaných další vlastní třídou *UserEndpointError*. K převodům na tento objekt slouží pak naše statické třídy *UserEndpointErrorHandler* a *SignInErrorHandler*.

Abychom dokázali definovat chyby doplněné o *ModuleType* parametr a zároveň vepsat český popis chyby, vytvořili jsme si třídu *IdentityErrorHandler* dědící od třídy *IdentityErrorDescriber*, která je přímo definovaná knihovnou Microsoft Identity a slouží právě

[9] Správce balíčků (knihoven) specifický pro .NET platformu

pro případ implementace vlastních datových struktur pro znázornění chyby [5]. Tu bylo potřeba řádně zaregistrovat v souboru *Program.cs*.

Do naší aplikace budou mít přístup různí uživatelé – někteří z nich budou studenti, jiní učitelé a jindy to zas budeme přímo my, vývojáři. Každý z takových uživatelů by měl mít specifická přístupová práva k jednotlivým endpointům. K rozlišení takových práv jsme využili *role*. Pro potřeby aplikace jsme si nadefinovali tři: *Admin, School, User*. Ty poté využíváme při definici přístupových skupin: *Admins, Schools* a *Everyone*.

Po správné implementaci knihovny a jejich hlavních funkcionalit je poté možné zabezpečit jednotlivé endpointy proti neoprávněnému přístupu. K takovému řízení slouží metoda *RequireAuthorization()*, jež se zapisuje při registraci endpointu (ukázka č. 8). Použití bez parametru dovolí užít takový endpoint každému přihlášenému uživateli. Pokud bychom však potřebovali omezit přístup jen určitým rolím, lze je do parametru vložit v jejich textové podobě. V ukázce je také vidět, že pomocí parametru typu *ClaimsPrincipal* lze získat přístup k instanci uživatelského účtu, se kterým backend právě pracuje.

```
routes.MapGet("/schoolMail",
    (ClaimsPrincipal user) => user.Identity.Name).RequireAuthorization();

routes.MapGet("/changeSchoolMail",
    (…) => {…}).RequireAuthorization(EAuthorizationRoles.Admins.ToString());
```

Kód 8: Zabezpečení přístupu k endpointu

Velkou výhodou použití knihovny Microsoft Identity je také fakt, že se vývojář nemusí na straně frontendu starat o implementaci funkcionality autorizace a autentifikace. Během přihlášení je serverem vytvořen token, jež je navrácen a uložen jako cookie[10] do zařízení uživatele. S každým dalším požadavkem je poté knihovnou tento token ověřen.

Díky velké abstrakci poskytnuté knihovnou Microsoft Identity pro nás nebyl problém integrovat také možnost přihlášení a registrace skrze existující uživatelské účty u jiných služeb. Aktuálně podporujeme přihlášení skrze účty u společností Google a Microsoft. Uživatel si tak nemusí vytvářet separátní účet jen kvůli naší aplikaci. Jednoduše se přihlásí skrze účet externí služby a ta nám předá informace o uživateli (jméno, příjmení a email). My následně ověříme, jestli už uživatele známe, a navštěvovanou školu tak už evidujeme, případně jestli ji nemůžeme

[10] Textový soubor uchovávající určitou informaci na zařízení uživatele po delší dobu

ověřit z předané emailové adresy. Pokud toto možné není a uživatel se přihlašuje poprvé, proběhne ještě žádost o ověření emailové adresy u vzdělávací instituce.

```
public async Task<IdentityResult> RegisterNewUser(UserRequest.RegisterUser user,
ScoolibContext db)
{
    var identityUser = new User()
    {
        Email = user.Email,
        UserName = user.Email,
        SchoolId = user.SchoolId
    };

    if(user.SchoolId == 0)
    {
        return IdentityResult.Failed(new UserEndpointError()
            {
                …
            }
        );
    }

    if(db.Schools.Where(w => w.Id.Equals(user.SchoolId)).Select(s =>
s.MailSuffix.Domain).FirstOrDefault() != user.Email.Split('@')[1])
    {
        return IdentityResult.Failed(new UserEndpointError()
            {
                …
            }
        );
    }

    var result = await _userManager.CreateAsync(identityUser, user.Password);

    if (result.Succeeded)
    {
        var email = await SendVerificationEmail(identityUser.Email, identityUser);
        return email;
    }
    return result;
}
```

Kód 9: Registrace nového uživatele

3.2.7 Další funkce backendu

V souvislosti s potřebou implementovat další funkce, specifické pro naši aplikaci, jsme museli využít několika dalších knihoven, které nejsou přímo ve frameworku *ASP.NET Core* dostupné.

Většinu jsme definovali odděleně, ve složce *Services* a vytvořili si tak vlastní třídy pro práci s těmito knihovnami.

Abychom naplnili bezpečnostní politiku naší aplikace, jež spočívá například v ověření účtů skrze emailové adresy, museli jsme zprovoznit odesílání emailů. Ačkoliv nativně je ve frameworku třída pro práci s emaily dostupná, její užití není v dnešní době doporučeno. Jako NuGet balíček jsme tedy nainstalovali udržovanou knihovnu MailKit. Tu využíváme ve vlastní třídě *Mail*, která skrývá většinu technické funkcionality a umožňuje odesílání pošty jen vložením dat do parametru instance třídy a zavolání metody *SendMail()*.

```
public async Task<IdentityResult> SendVerificationEmail(string email, User user)
{
    …
    var result = new Mail()
    {
        Subject = "Scoolib - ověření registrace",
        To = new MailboxAddress("", email),
        HtmlBody = mailBody
    }.SendMail();
    return result ? IdentityResult.Success : IdentityResult.Failed(new
IdentityErrorHandler().UnsentEmail());

}
```

Kód 10: Odeslání emailu skrze třídu Mail

Klíčovou funkcí naší aplikace je generování souborů PDF se seznamem děl k maturitě. Samotné generování jsme se rozhodli zajistit knihovnou *MigraDoc*, která staví nad základy knihovny *PdfSharp* a poskytuje nad ní větší míru abstrakce během tvorby dokumentů [6]. Obě knihovny jsou open-source, díky čemuž jsme je zvolili namísto alternativy *iTextSharp*. Značnou překážkou během tvorby dokumentů je potřeba tvořit jeho obsah pomocí vestavěných objektů, metod a parametrů (kód č. 11).

V naší aplikaci jsme se rozhodli pro reprezentaci některých hodnot použít tzv. *enumy*. Enumy vyjadřují uzavřený výčet hodnot přiřazenou číselnou hodnotou. Číselná reprezentace nejenže je efektivnější při ukládání dat, ale zároveň poskytuje lepší možnosti pro dlouhodobou správu hodnot, a především jejich úpravu [7]. Časové zařazení u každé knihy tvoří poměrně dlouhý textový řetězec, např. ”Česká literatura 20. a 21. století” Ukládat takový řetězec u každé knihy je neefektivní, zvláště, když dochází k jeho opakování velmi často. Namísto toho jsme tedy definovali enum *EBookPeriod*, v němž třeba tento textový řetězec představuje hodnotu *3*. Do databáze je následně vložena právě tato číselná hodnota. Abychom ji byli schopni jednoduše

při dotazování na data převést zpět na textový řetězec, používáme knihovnu *AutoMapper*. Díky ní se nám zpětný převod značně zjednodušil.

```
public static byte[] CreateAnalysis(String logo, IQueryable<Book> books)
{
    Document document = new Document();
    …
    Section page = new Section();
    …
    Table listInfo = new Table();
    listInfo.Borders.Width = 0.75;

    double tableUsableWidth = usableWidth - listInfo.LeftPadding.Point -
listInfo.RightPadding.Point;

    listInfo.AddColumn(tableUsableWidth/2);
    listInfo.AddColumn(tableUsableWidth/2);
    listInfo.Format.Font.Size = 14;

    Row rowName = listInfo.AddRow();
    rowName.Cells[0].AddParagraph("Jméno a příjmení žáka");
    rowName.Cells[1].AddParagraph("Adam Hojer");

    Row rowClass = listInfo.AddRow();
    rowClass.Cells[0].AddParagraph("Třída");
    rowClass.Cells[1].AddParagraph("4.F");

    Row rowSchool = listInfo.AddRow();
    rowSchool.Cells[0].AddParagraph("Školní rok");
    rowSchool.Cells[1].AddParagraph("2024/2025");

    page.Add(listInfo);
    …
    document.Sections.Add(page);

    PdfDocumentRenderer renderer = new PdfDocumentRenderer(true)
    {
        Document = document
    };
    renderer.RenderDocument();

    MemoryStream ms = new MemoryStream();

    renderer.PdfDocument.Save(ms, true);

    return ms.ToArray();
}
```

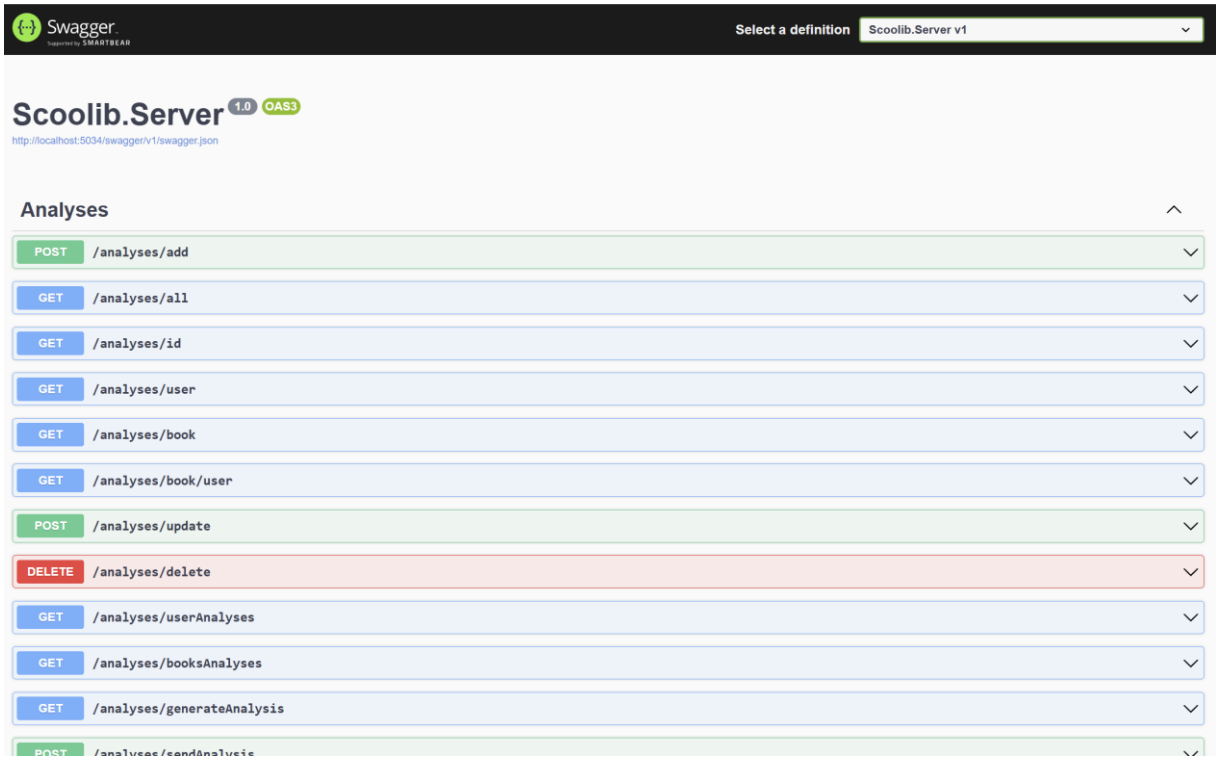
Kód 11: Přidání tabulky s informacemi o žákovi

3.2.8 Testování API

Abychom mohli funkčnost jednotlivých endpointů otestovat, využíváme v projektu nástroj *Swagger*. Ten při každém spuštění projektu zanalyzuje databázové kontexty, vytvoří si přehled o endpointech, jejich parametrech i návratových hodnotách a spustí webové prostředí pro jejich reprezentaci (obrázek č. 2). V něm je lze i otestovat. Jelikož jsou ve Swaggeru vidět všechny dostupné endpointy, jejich typy metod a datová struktura, není ideální vystavit toto prostředí i v produkční verzi. Z toho důvodu inicializujeme Swagger jen během vývoje (ukázka kódu č. 12).

```
if (app.Environment.IsDevelopment())
{
    app.UseSwagger(); //analyzace projektu
    app.UseSwaggerUI(); //zapnutí webového prostředí
}
```

Kód 12: Implementace nástroje Swagger



Obrázek 10: Swagger pro projekt Scoolib.Server

3.3 Frontend

Frontend je část aplikace, se kterou interaguje cílový uživatel aplikace, běží tedy zpravidla právě na zařízení cílového uživatele. Do vývoje frontendu spadá návrh rozhraní aplikace (design) a jeho implementace, včetně zajištění funkcionality. Je potřeba sladit UI (vzhled) a UX (uživatelskou přívětivost), což je často obtížné.

Hlavní úlohou frontendu je odprezentovat informace a nabízet možnosti interakce s nimi. Vývoj frontendu, který zahrnuje nejen samotnou implementaci uživatelského rozhraní, začíná už ve fázi návrhu aplikace. Tento návrh je důležitou součástí, protože definuje, jak bude aplikace vypadat, jak se bude chovat a jak budou uživatelé s aplikací interagovat. Právě proto jsme věnovali značné úsilí samotnému návrhu, kde jsme definovali strukturu, rozložení a celkový vzhled aplikace ještě před samotným programováním.

3.3.1 Návrh designu

Prvním krokem při vytváření aplikace bylo rozvržení její podoby. Začali jsme stanovením hlavních funkcí, které má aplikace nabízet. Po jasném určení účelu jsme mohli navrhnout strukturu stránek a vytvořit celkový návrh.

Na papír jsme nejdříve načrtli design jednotlivých stránek. Přitom jsme se soustředili na to, aby měli uživatelé vše potřebné na jednom místě a mohli se jednoduše orientovat. Rozmístili jsme klíčové komponenty tak, aby bylo jejich používání příjemné a vizuálně atraktivní. Zároveň jsme mysleli na responzivitu – jak bude rozložení vypadat v desktopové, tabletové a mobilní verzi.

Po vytvoření kostry aplikace jsme přešli do nástroje Figma, kde jsme mohli náš návrh plně realizovat do konkrétní designové podoby.

Jako první věc, kterou jsme při vytváření návrhu řešili, byla volba barev aplikace. Rozhodli jsme se pro čistě černou jako primární barvu, přičemž u ikonek a dalších prvků jsme využili pestrou škálu dostupných barev.

Jako další věc jsme řešili typografii, tudíž jaký font použijeme pro to, aby byl čitelný a zároveň se hodil k estetice naší aplikace. U typografie také řešíme, jaká bude velikost písma v daných sekcích, jako jsou třeba nadpisy, podnadpisy a nebo běžné texty.

Také musíme přizpůsobit velikost písma pro výše zmíněná zařízení, tudíž zmenšíme velikost pro mobilní vzhled aplikace, aby byla plně responzivní a text byl dostatečně malý, aby byla aplikace vizuálně přívětivá a zároveň aby bylo písmo čitelné.

Barvu textu přizpůsobujeme tak, aby uživatelé pochopili, co je více důležité a co je naopak méně důležité. Důležitější texty mají černou barvu a pro podnadpisy či obyčejné texty snížíme světlost, tudíž bude písmo šedivé.

Také se držíme osvědčených barev – např. u kontroly kritérií na stránce "Můj seznam" používáme zelenou pro splněná kritéria a červenou pro nesplněná kritéria.

Dále jsme navrhli, jak budou vypadat tlačítka. Pro různé případy, které můžou nastat, potřebujeme jiné variace tlačítek. Navrhli jsme tlačítko "primární", které bude mít černé pozadí a bílý text – slouží jako hlavní tlačítko a uživatel si ho kvůli primární barvě všimne většinou jako první. Pak máme "sekundární" tlačítko, které má šedou barvu pozadí a černý text. Když zvýšíme světlost pozadí tlačítka, mozek člověka to automaticky bude brát jako druhou možnost a pro hlavní akci bude vědět, že má použít primární tlačítko. Dále máme "obrysovou" variaci tlačítka, která má pouze šedý obrys a transparentní pozadí s černým textem uvnitř. Tento typ tlačítka obsahuje akce, které jsou důležité, ale nejsou primární akcí v dané situaci. Obrysová tlačítka se dobře párují s primárními tlačítky, protože označují alternativní akci oproti primárnímu tlačítku. Dále v aplikaci používáme "výstrahová" tlačítka, která mají červené pozadí a bílý text. Tato tlačítka značí to, že se chystáme udělat nějakou nebezpečnou akci, jako například odstranit rozbor či odstranit účet. Z výstrahového tlačítka tímto způsobem uděláme primární tlačítko a v aplikaci ho kombinujeme s výstražným dialogem, který slouží k tomu, abychom si danou akci rozmysleli. Vedle výstražného tlačítka používáme obrysovou variaci, abychom zrušili danou akci.

Na začátku při vytváření projektu jsme si stanovili technologie, které využijeme při vývoji aplikace. Mezi klíčové nástroje patří Shadcn/ui knihovna, která nabízí předpřipravené komponenty pro snadné použití v naší aplikaci.

Vzhledem k rozhodnutí použít Shadcn jsme tomu přizpůsobili i náš designový návrh. Při práci ve Figmě jsme se inspirovali vzhledem komponent této UI knihovny a navrhli je tak, aby přesně

odpovídaly Shadcn stylu. Tento přístup významně zefektivnil naši práci, protože jsme nemuseli vyvíjet komponenty od základu.

Mezi komponenty, které jsme takto implementovali, patří například dříve zmíněná tlačítka, vyskakovací dialogy a vstupní textová pole. Díky této strategii jsme mohli rychleji postupovat ve vývoji a zároveň zachovat konzistentní design napříč celou aplikací.

Úvodní stránka je ta první věc, kterou uživatel vidí při navštívení samotné aplikace. V našem případě byla stránka předělána, protože původní verze nevystihovala klíčové body, které by měly uživatele nalákat k tomu, aby začali aplikaci používat. Současná verze se skládá z navigačního menu, které je vždy viditelné v horní části úvodní stránky. Obsahuje tři nejdůležitější věci, a těmi je logo a dvě CTA tlačítka pro přihlášení a registraci.

Pod navigačním menu se nachází úvodní sekce, kde je slogan velkým tučným písmem a pod tímto textem se nachází obyčejný text, který velmi stručně popisuje, co se dá v aplikaci dělat. Hned pod textem jsou dvě stejná CTA tlačítka, která můžeme vidět v navigačním menu. Aby nebyla tato sekce prázdná, tak jsou kolem úvodního textu barevné ikony, které souvisí s tématem aplikace. Poslední věc, kterou zde můžeme vidět, je ukázka vnitřku aplikace, protože tímto ukážeme uživatelům, co mají očekávat ještě před tím, než se vůbec zaregistrují, a zvyšujeme tak šance na to, že uživatel aplikaci použije.

Pod úvodní sekcí jsou zobrazené informační karty, ve kterých mohou uživatelé vidět, jakým způsobem mohou aplikaci využít. Tímto dáváme budoucím uživatelům příležitost ukázat, co jim může aplikace nabídnout.

Dále je sekce s konkrétními pohledy aplikace a možnosti jejich využití. Napadlo nás použít stručný nadpis s detailnějším textem, který podrobně popíše jednotlivé stránky. Vedle textu se nachází obrázek přímo z aplikace, aby si mohli daný text ověřit a lépe představit.

Těmito sekcemi jsme dali příležitost uživatelům vidět, co může aplikace nabídnout, a proto další sekce navazuje tím, že opět vyzve uživatele k používání aplikace. Nachází se zde text se stručným popisem aplikace a CTA tlačítkem pro registraci.

Poslední dvě sekce jsou sekce s častými dotazy, kde uživatelé najdou potřebné odpovědi na otázky a kontakt sekce s e-mailem.

Po přihlášení do aplikace může uživatel vidět levý boční panel, který mu slouží k navigaci mezi jednotlivými stránkami. Také se tam nachází logo aplikace, tlačítko pro dialog s nastavením a tlačítko pro odhlášení z účtu. Jednotlivá tlačítka jsou seskupeny podle jejich funkčnosti, aby to uživatele nezmátlo. Také jsou stránky v bočním panelu poskládany od shora dolů systematicky za sebou tak, aby uživatel věděl, jak aplikaci používat, a to tedy tak, že otevře knihovnu (stránka

Knihovna), zde si knihu přidá do seznamu (stránka Můj seznam), a po přidání knihy do seznamu si vytvoří k dané knize rozbor (stránka Rozbory).

Stránka "Domů" slouží jako hlavní přehledová stránka aplikace, kde uživatelé najdou všechny klíčové informace a rychlý přístup k nejdůležitějším funkcím. Design této stránky je navržen tak, aby poskytoval přehled o stavu čtení a nabízel snadný přístup ke knihám.

V horní části stránky jsou umístěny tři hlavní informační karty, které poskytují rychlý přehled o postupu uživatele. První karta "Knihy" zobrazuje celkový počet knih v seznamu, druhá karta "Přečteno" ukazuje počet přečtených knih a třetí karta "Rozbory" indikuje počet vypracovaných rozborů. Každá karta má vlastní ikonu a pozadím je barevně oddělena pro snadnou orientaci.

Hlavní obsah stránky je rozdělen do sekcí podle kategorií knih – uživatelé zde mohou najít nejpopulárnější knihy nebo knihy s nejmenším počtem stran. Nad každou sekcí je umístěn přehledný navigační systém, který umožňuje rychlé filtrování zobrazených knih podle literárního druhu. Na pravé straně každé sekce se nachází tlačítko "Zobrazit vše", které přesměruje uživatele na kompletní seznam knih v knihovně.

Knihy v každé sekci jsou zobrazeny v horizontálním seznamu. Každé políčko v dlaždicovém zobrazení obsahuje obálku a název knihy, přičemž při najetí myší na knihu se zobrazí stavové ikony pro rychlé přidání do seznamu. Tento způsob zobrazení byl zvolen pro snadnou interakci pro uživatele.

Důležitým prvkem designu je propojení jednotlivých sekcí pomocí odkazů v informačních kartách. Například tlačítko "Knihovna" v horní části vede na stránku s kompletním seznamem knih, zatímco "Můj seznam" směřuje na přehled vybraných knih.

Stránka "Knihovna" představuje hlavní bod aplikace, kde uživatelé naleznou kompletní seznam dostupných literárních děl. Design této stránky je navržen s důrazem na přehlednost a efektivní filtrování knih.

Hlavní obsah stránky je rozdělen do dvou částí – seznam knih a filtrační panel. Seznam knih je uspořádán do mřížky, kde každá kniha je opět reprezentována svou obálkou a názvem. Tento způsob zobrazení byl zvolen pro optimální vizuální přehlednost a rychlou orientaci. Obálky knih slouží jako vizuální kotvy, které umožňují uživatelům rychle identifikovat hledanou knihu.

Mřížkové uspořádání se automaticky přizpůsobuje velikosti obrazovky, čímž je zajištěna plná responzivita stránky.

Pravý panel obsahuje filtrovací systém, který je důležitý pro efektivní práci s knihovnou. V horní části panelu se nachází vyhledávací pole, kam mohou uživatelé zadat název knihy nebo jméno autora. Pod ním jsou umístěny různé kategorie filtrů:

- Literární druh – implementováno pomocí zaškrtávacích políček
- Literární žánr – také implementováno pomocí zaškrtávacích políček
- Počet stran – implementováno pomocí posuvníku

Pro mobilní zařízení je filtrační panel skryt a lze jej zobrazit pomocí tlačítka umístěného v pravém dolním rohu obrazovky. Toto umístění bylo zvoleno s ohledem na používání mobilních zařízení, kde většina uživatelů ovládá telefon pravým palcem. Po aktivaci se filtr zobrazí jako okno přes seznam knih, což umožňuje uživateli plně se soustředit na nastavení filtru.

Ve spodní části filtračního panelu je umístěno tlačítko "Vymazat filtry", které jedním kliknutím resetuje všechna nastavení filtru do výchozího stavu. Toto tlačítko je vizuálně odlišeno tmavým pozadím, aby bylo snadno identifikovatelné v případě potřeby.

Interakce s jednotlivými knihami je řešena pomocí stavových ikon, které se objeví při najetí myší na obálku knihy. Tyto ikony umožňují rychlé přidání knihy do seznamu bez nutnosti otevírat detail knihy, což významně zefektivňuje práci s aplikací. Tento přístup k designu zajišťuje, že uživatelé mohou efektivně pracovat s rozsáhlým seznamem knih a snadno najít přesně to, co hledají.

Na stránce "Můj seznam" uživatel najde přehled všech knih, které si přidal do svého seznamu. V horní části stránky se nachází navigační lišta, která obsahuje název stránky a na pravé straně můžeme vidět počítadlo splněných kritérií. Toto počítadlo je důležitým prvkem, protože uživatel okamžitě vidí, kolik knih má přečteno z celkového počtu potřebného pro maturitní zkoušku.

Pod navigační lištou se nachází tři hlavní karty, které jsou vizuálně odděleny a každá obsahuje stejné informace, které můžeme najít na přehledové stránce.

Na pravé straně obrazovky je umístěn panel s kritérii, který je rozdělen do tří hlavních kategorií - Próza, Poezie a Drama. Každá kategorie má svůj vlastní číselný indikátor a barevné označení pro lepší orientaci. Pod těmito kategoriemi jsou další podkategorie rozdělené podle literárních

období. Toto rozdělení pomáhá uživatelům sledovat kritéria v jednotlivých žánrech a obdobích literatury.

Hlavní část stránky je seznam knih, které jsou systematicky rozděleny podle literárních období.

Ve spodní části stránky se nachází tlačítko "Stáhnout seznam", které umožňuje uživatelům exportovat jejich seznam knih do svého zařízení. Toto tlačítko je umístěno na strategické pozici, kde je snadno dostupné.

3.3.2 Výběr technologií

Pro implementaci našeho projektu jsme pečlivě zvážili výběr technologií a rozhodli jsme se pro kombinaci moderních nástrojů, které nejlépe odpovídaly našim potřebám. Zvolili jsme React jako hlavní knihovnu pro frontend, TailwindCSS pro styling a Shadcn/ui jako knihovnu komponent.

React jsme si vybrali hlavně kvůli našim předchozím zkušenostem s jeho použitím. Volba byla poměrně jasná a naše znalosti nám značně pomohli v průběhu vývoje. Jako alternativu jsme mohli zvolit například *Vue.js,* ale v tomto případě jde už o kompletní framework značně ovlivňující způsob vývoje aplikace v JS definováním vlastních pravidel a způsobů práce. To se nám nelíbilo.

React samotný je pouhá knihovna rozšiřující funkcionalitu samotného Javascriptu. Pokud chce vývojář nad Reactem postavit komplexní aplikaci, musel by se postarat o mnoho dalších věcí – jako je třeba integrace externích knihoven nebo možnosti sestavení aplikace do produkční podoby. Pro tyhle účely jsou zde ale nástroje nebo frameworky, které tyto úlohy plní a integrují. I nad Reactem samotným tedy existuje velká řada frameworků, nebo pomocných nástrojů pro tyto účely. My jsme si díky pozitivním předchozím zkušenostem zvolili nástroj *Vite.js*, který zajišťuje všechny potřebné funkce pro základní vývoj, ale zároveň nezahlcuje projekt a vývojový proces nadbytečnou vlastní vrstvou funkcí a principem fungování – což často plnohodnotné frameworky dělají.

Pro stylování jsme zvolili TailwindCSS, který nám umožnil mít všechny styly přímo v HTML kódu. Toto rozhodnutí bylo založeno na našich předchozích pozitivních zkušenostech s tímto frameworkem. Na rozdíl od Bootstrapu, který často vede k podobně vypadajícím aplikacím, nám Tailwind poskytl flexibilitu a naprostou kontrolu nad vzhledem. Rozhodnutí nepoužít čisté CSS bylo domluveno z toho důvodu, že nám neposkytuje dostatečně rychlý vývoj a je zde těžší implementovat responzivní design. Utility třídy Tailwindu významně urychlily proces vytváření uživatelského rozhraní a umožnily nám vytvořit unikátní design.

Jako další klíčovou technologii jsme zvolili knihovnu Shadcn/ui, která je plně kompatibilní s TailwindCSS. Tato knihovna nám poskytla předpřipravené komponenty s moderním designem a plynulými animacemi. I když jsme zvažovali populární alternativy jako Material-UI, Shadcn jsme upřednostnili kvůli jeho jednoduchým a moderním vizuálním prvkům. Přestože integrace

funkcionalit může být složitější než u Material-UI, výsledný vzhled a animace komponent byly pro náš projekt klíčové a pomohly nám vytvořit profesionálně vypadající aplikaci.

3.3.3 Integrace designu

Pro implementaci našeho designového návrhu do skutečné aplikace jsme využili připravený návrh ve Figmě jako referenční bod. Snažili jsme se co nejlépe napodobit původní design, i když během implementace došlo k několika úpravám pro lepší funkčnost.

Přestože je v současné době populárnější přístup "mobile-first", kdy se nejprve vytváří mobilní verze aplikace a až poté desktopová, my jsme zvolili opačný postup. Bylo to především proto, že náš původní návrh ve Figmě byl vytvořen pouze pro desktopovou verzi.

Mít připravený design před začátkem programování se ukázalo jako velmi efektivní přístup. Značně to zjednodušilo proces implementace, protože jsme nemuseli během vývoje vymýšlet vizuální stránku aplikace od základů. Práce s TailwindCSS a jeho třídami významně urychlila proces stylování, zatímco komponenty ze Shadcn/ui nám poskytly velmi dobrý základ pro interaktivní prvky

Při zakládání projektu jsme věnovali pozornost organizaci souborové struktury. Vytvořili jsme jasně definovanou hierarchii složek:

- Složka *components* obsahuje všechny znovupoužitelné komponenty
- Podsložka *components/ui* je vyhrazena pro komponenty ze Shadcn knihovny
- Ve složce *pages* jsou umístěny jednotlivé stránky aplikace

Každá stránka má svou vlastní složku, která obsahuje hlavní komponentu a související podkomponenty specifické pro danou stránku. Tato struktura nám umožnila efektivně organizovat kód a udržovat ho přehledný i při rostoucím počtu nových souborů.

I přesto, že jsme měli připravený návrh ve Figmě, během implementace došlo k několika významným změnám v designu. Zjistili jsme, že první verze návrhu není vždy tou nejlepší, a že při praktickém převodu do kódu často objevíme lepší řešení. Například úvodní stránku jsme kompletně přepracovali, i když už byla naprogramovaná podle původního návrhu, protože jsme viděli potenciál pro výrazné vylepšení.

V průběhu vývoje jsme stále upravovali různé aspekty designu – od drobných změn v nadpisech kategorií až po výraznější úpravy barevných schémat. Tento přístup k designu nám umožnil vylepšovat uživatelské rozhraní na základě praktických zkušeností s implementací. Ukázalo se,

že není možné se striktně držet prvního návrhu, protože při samotném programování často objevíme lepší způsoby, jak určité prvky realizovat.

3.3.4 Integrace funkcionality

React, resp. jeho zahrnutí ve Vite.js, je primárně určeno pro tzv. *SPA*[11] aplikace. Takové typy aplikací mění svůj obsah přepisováním aktuálního obsahu novým. Nedochází ke kompletnímu načítání nových stránek, což zvyšuje jejich reakční dobu a možnosti interaktivity pro cílového uživatele. V praxi to také znamená, že jednotlivé lokace (např. */library*, */book?id=64*) nejsou prezentovány žádnými fyzickými soubory. Veškeré požadavky na jednotlivé podstránky musí být směrovány na centrální bod aplikace, zpravidla jde o soubor *index.html*. Ten sám se uvnitř sebe postará o zobrazení konkrétního obsahu.

Aby bylo možné zobrazovat efektivně odlišný obsah na základě URL adresy, používá se uvnitř aplikace metoda *routingu*[12]. Jeho funkcionalita není ve Vite nativně k dispozici, je potřeba ji tedy zajistit instalací knihovny. V našem případě došlo k výběru oblíbené *react-router-dom*. Při její implementaci jsme navíc využili funkcionalitu *lazy loadingu* pro jednotlivé podstránky reprezentované komponentami. Normálně se totiž celá aplikace přeloží do jednoho Javascript souboru, jehož načítání je poté náročné na datový přenos. V našem případě se jednotlivé moduly aplikace kompilují[13] zvlášť, do oddělených souborů.

Každá aplikace React potřebuje *index.html* soubor, jež v těle obsahuje pouhé dva elementy. Jedním z nich je *div* element s id *'root'*. Druhým elementem je skript odkazující na soubor *main.tsx*. V něm samotném probíhá inicializace, jak ukazuje ukázka kódu č. 13. Do div elementu je zde pomocí funkce *render()* vykreslena samotná aplikace.

```
ReactDOM.createRoot(document.getElementById('root')!).render(
    <ContextWrapper>
      <RouterProvider router={router} />
    </ContextWrapper>
)
```

Kód 13: Integrace React uvnitř main.tsx souboru

Nyní je už celý obsah v gesci Reactu a jeho komponent. V inicializační metodě *render()* dochází k vykreslení prvku *RouterProvider*. Ten zajišťuje právě zmíněný routing pro jednotlivé podstránky aplikace. Samotná mapa stránek je definována objektem *router*, který je zde předáván skrze parametr. Zkrácenou verzi zobrazuje ukázka kódu č. 14. Zde je vidět například i metoda tvorby řetězení podstránek na sebe pomocí definice v *children* parametru. Parametr

[11] Single-page application
[12] Směrování na konkrétní komponenty aplikace na základě adresy
[13] Proces převodu zdrojového kódu na spustitelnou aplikaci

element už odkazuje přímo na konkrétní komponentu, která má být při vstupu na danou URL adresu vykreslena.

```
const router = createBrowserRouter([
  {
    path: 'resetPassword',
    element: <LazyLoader><ResetPassword/></LazyLoader>,
  },
  {
    path: '/dashboard',
    element: <LazyLoader><Dashboard/></LazyLoader>,
    children: [
      {
        path: 'analysis',
        element: <LazyLoader><Analysis/></LazyLoader>,
      },
      {
        path: 'analysis/:id',
        element: <LazyLoader><AnalysisBook /></LazyLoader>,
      },
      {
        path: 'analysis/create/:id',
        element: <LazyLoader><BookAnalysisCreate /></LazyLoader>,
      }
    ]
  }
])
```

Kód 14: Definice routingu v aplikaci

Některé stavy je třeba mít dostupné z několika částí aplikace. Pro takový účel existují v reactu *kontexty*, které zajišťují sdílení hodnot v rámci většího aplikačního celku [8]. Na základě umístění *wrapperu*[14] kontextu dochází k jeho zpřístupnění všem jeho podkomponentám. My

[14] Element nesoucí v sobě jiné elementy, často je v rámci tohoto celku doplněna určitá funkcionalita

jsme si wrapperem obalili přímo element pro routování, abychom měli zajištěno, že kontext bude k dispozici v celé aplikaci, nehledě na právě zobrazenou komponentu. V našem kontextu uchováváme stav uživatelské relace, aktuální barevný režim, nebo například styl zobrazení postranního menu.

3.3.5 Typování

V dřívějších dobách nebylo zvykem při vývoji frontendu používat typování. Typováním je možné každému datovému modelu definovat, s jakými přesnými typy pracuje – tedy zda jde o číslo, textový řetězec nebo třeba funkci. Javascript je dynamicky typovaný jazyk. To znamená, že datové typy vyhodnocuje za běhu programu. Pokud někde dojde k nesouladu, je taková chyba zjištěna až za běhu, kdy konkrétní část programu přestane fungovat tak, jak má, a v případě nedefinovaných hodnot vyvolá výjimku. Předejít tomuto chování lze *typovou anotací*, tedy striktním stanovením datového typu jednotlivým objektům. To ale Javascript nativně nepodporuje, což značně komplikovalo vývojový proces aplikací. Vývojáři sami museli ručit a být odpovědní za předávání správných typů v rámci aplikace.

S rostoucí popularitou Javascriptu ale bylo volání po podpoře typování ze strany vývojářů stále vyšší a vyšší. Microsoft tedy v roce 2012 představil jazyk *TypeScript*. Ten nenahrazoval samotný JavaScript, jen nad něj přidal podporu typových anotací a kontroly typů během vývoje [9]. Typování je zde statické, kontrola správnosti probíhá jen před *transpilací*. Transpilace je proces zpětného převodu kódu s TypeScriptem do jazyka Javascript, což je nutné pro jeho spuštění. Webové prohlížeče totiž TypeScript nepodporují, už od začátku byl tedy jazyk navržen tak, aby kontroloval vývojáře během vývojového procesu.

Integrací typových anotací se vývojový proces značně zjednoduší. Kód je více předvídatelný a během psaní má vývojář přehled nad tím, s jakými hodnotami pracuje. To přináší nedocenitelnou výhodu. Předchází se tak mnoha chybám, které by se jinak ve vývojovém procesu neodhalili a jejich zpětné dohledání by bylo náročné. TypeScript jsme tedy do našeho projektu kompletně zahrnuli.

V ukázce kódu č. 15 je vidět praktické využití takového typování. Backend nám bude vracet informace o jedné knize ve formě objektu s jednotlivými parametry. Podobu objektu si tedy musíme nejdříve nadefinovat tak, aby odpovídala struktuře vrácené z backendu. S tím nám pomůže nástroj Swagger, popsaný v kapitole 3.2.8, který má o datové struktuře vracených dat přehled. Jakmile máme typ Book hotový, můžeme ho následně už využívat. Ve funkci *useState()* deklarujeme, že bude držet hodnotu typu *Book*, nebo *null* – to v případě, kdy teprve budeme čekat na informace z backendu. Při volání *id()* reprezentuje *Book* formu navrácených dat. Pokud bychom zde definovali, že funkce bude vracet třeba textový řetězec, už během

vývoje by se nám toto indikovalo jako chybný zápis. Problém by byl ve volání *setBook()*, které vyžaduje vstupní parameter *Book* nebo *null*, textový řetězec by tedy nebyl přijat jako validní.

```
type Book = {
    id: number,
    author: string,
    class: string,
    created: string,
    description?: string,
    difficulty: string,
    films: BookFilm[],
    …
}
const [book, setBook] = useState<Book|null>(null);
booksEndpoints.id<Book>(id).then(setBook).catch(console.error);
```

Kód 15: Použití TypeScriptu u entity knihy

3.3.6 Reaktivita

Každá webová stránka a její obsah je prezentován skrze DOM. Jde o strukturu, kde je každý element stránky popisován jako *node*. K těmto elementům se při použití obyčejného Javascriptu přistupuje na základě některých jeho vlastností, třeba přes ID objektu, jeho třídu aj. Veškerá práce s nimi probíhá skrze imperativní paradigma[15] programování. Manipulace je prováděna postupným prováděním příkazů.

React přináší zásadní změnu v pohledu na tvorbu webových stránek, resp. zajištění jejich funkcionality. Používá paradigma deklarativního programování. To lze popsat tak, že je celá tvorba více specifikována na konkrétní úkony, ne na postup jejich vykonání. React samostatně zajišťuje manipulaci s elementy a jejich odpovídající reakce, vývojář jen definuje, co přesně takové změny vyvolá.

Často je třeba změnit obsah stránky na základě nového stavu dat. V našem případě toto nastává zejména ve dvou situacích. Jednou z nich je čekání na data od backendu. Jak bylo zmíněno dříve, nejdříve je na zařízení zobrazena základní struktura webu. Až po přijetí dat dochází k jejich zobrazení. Je potřeba tedy stránku znovu překreslit v okamžiku úspěšného získání. Pro takový efekt stačí definovat určitý objekt, který změnu zajistí. Při použití klasické proměnné

[15] V programování popisuje přístup k řešení problémů a tím odpovídající schéma zápisu kódu

nemá React možnost zjistit její změnu, proto se pro dynamická data využívají tzv. *staty*. Jejich způsob zápisu se mírně liší od klasických proměnných a připomíná více možnosti OOP jazyků. Jak ukazuje ukázka kódu č. 16, takový state je definován polem o dvou objektech. Jedním z nich je getter, který slouží pro čtení hodnoty. K přiřazení nové hodnoty slouží naopak druhý objekt, setter.

```
const Book = () => {

    const [book, setBook] = useState<Book>(); //definice state


    useEffect(() => {

        booksEndpoints.id<Book>(id).then(setBook); //správný přístup

        booksEndpoints.id<Book>(id).then(res => book = res); //nezajistí re-render

    }, []);


    if (!book) return <LoadingPage/>


    return(

        <div>

            <p>{book.title}</p>

        </div>

    )

}
```

Kód 16: Ukázka práce se stavem v Reactu

V ukázce je možné si povšimnout další zajímavé součásti Reactu, a sice *useEffect()*. Tento *hook*[16] zajišťuje vykonání určitých instrukcí poté, co dojde ke změně hodnoty definovaného stavu. React totiž sám zajistí na základě změny hodnoty re-render stránky a její naplnění daty novými, někdy je ale žádoucí po změně vykonat i další činnosti. K tomu právě slouží

[16] Speciální funkce Reactu umožňující přístup k jeho vlastním strukturám (state, kontext aj.)

useEffect(). V našem konkrétním případě by mohlo třeba po získání informací o knize dojít k zavolání další funkce k získání rozborů pro tuto konkrétní knihu.

3.3.7 Komunikace s API

Získávání dat z backendu funguje na základě volání požadavků na URL adresu endpointu. Pokud je to u konkrétního endpointu vyžadováno, mohou být v těle požadavku, nebo v parametrech URL, zaslány dodatečné parametry a data pro specifikaci určitých hodnot. V našem případě se tak děje například při získávání informací ke konkrétní knize, kdy musíme backendu specifikovat, ohledně jaké knihy informace potřebujeme. Každý požadavek poté vrací odpověď. Její první důležitou částí je *http* status kód. Ten nám definuje, zda byl požadavek serverem správně vyřízen, nebo došlo k určité chybě. Pokud k chybě nedošlo, je zpravidla vrácen i nějaký obsah.

V Javascriptu je nativně několik možností, jak takové požadavky volat a zpracovávat. V základu lze každý takový požadavek definovat skrze objekt *XMLHttpRequest*, jež na sebe poté váže metody pro odeslání a zpracování. Určitou míru abstrakce nad touto funkcionalitou nabízí další nativní způsob, a sice funkce *fetch()* dostupnou z *FETCH API* [10]. Ani ta nám ale

nestačila, a tak jsme sáhli po komplexní knihovně *axios*. V souboru *restClient.ts* si definujeme vlastní funkce pro zasílání požadavků využívající funkce axiosu (ukázka kódu č. 17).

```
const getBase = async <T,>(url: string, config?: AxiosRequestConfig<any>, params?:
any): Promise<T> => await (await axios.get(url, { ...config, withCredentials: true,
params: params })).data;


const getProcedure = <T,>(name: string, params?: any) =>
getBase<T>(`${apiUrl}${name}`, params).then((data) => data, (err) =>  throw err});
```

Kód 17: Vlastní funkce pro práci s API

Všechny endpointy definujeme v oddělených objektech v souboru *proxy.ts*. Každý endpoint zde má přiřazeno volání konkrétní funkce ze souboru *restClient.ts* s danými parametry a adresou (ukázka kódu č. 18)

```
export const booksEndpoints = {

    all: <T>() => getProcedure<T>("books/all"),

    id: <T>(id: number) => getProcedure<T>(`books/id?id=`+id),

    idSimple: <T>(id: number) => postProcedure<T>(`books/id/simple?id=${id}`),

    isInList: <T>(id: number) => getProcedure<T>(`books/isInList?id=${id}`),

    updateInList: <T>(id: number) =>
postProcedure<T>(`books/updateInList?id=${id}`),

}
```

Kód 18: Definice API endpointů

Návratový typ většinou není explicitně definován. Namísto toho je využita generika. To znamená, že funkce může pracovat s vícero datovými typy a konkrétní datový typ je indikován až při jejím volání. V našem případě je návratový typ vždy jeden, využití generiky má tedy jiný důvod. Některé typy v našem projektu neexportujeme, a jsou tak dostupné jen na konkrétních místech aplikace. Abychom nemuseli všechny typy exportovat globálně, a to pouze z důvodu definice návratové hodnoty endpointů, definujeme návratový typ v místě volání funkce konkrétního endpointu (ukázka kódu č. 19). Zde si také můžeme všimnout způsobu předávání

vstupních parametrů do požadavků a následnou možnost práce s výsledkem (*then*), případně s chybou (*catch*). To je možné díky architektuře funkce *getBase*, která vrací *Promise*[17].

```
booksEndpoints.id<Book>(id).then(setBook).catch(console.error);
```

Kód 19: Volání funkce pro získání knihy z API

[17] JS objekt reprezentující příslib budoucího výsledku funkce

3.4 Vývoj aplikace, nasazení

3.4.1 Vývojové prostředí

K vývoji samotné aplikace je potřeba správné vývojové prostředí (IDE). Ideální je volit takové IDE, které je pro práci s konkrétní technologií, postupy nebo jazykem přizpůsobeno, a může tak vývojáři tvořit značného pomocníka při rutinních úkolech.

Pro psaní backend funkcionality jsme využívali každý z nás odlišný program, Rider a Visual Studio. Obě prostředí jsou zaměřeny na práci s jazykem C# a mají podpůrné funkce pro jejich konkrétní typy implementací, včetně tvorby *ASP.NET Core* aplikace s Web API. Zatímco Visual Studio je vyvíjeno společností Microsoft Corporation a jde o velmi komplexní vývojové prostředí s podporou mnoha dalších technologií, Rider od společnosti JetBrains je zaměřen specificky na jazyk C#.

Pro psaní frontend kódu jsme využili open-source programu Visual Studio Code, který v základu funguje jako pokročilý textový editor. S pomocí rozšíření ale nabízí podporu pro konkrétní jazyky. Pro detailní debugging frontend kódu jsme využívali doplněk *React DEV tools* dostupný v prohlížeči Chrome.

3.4.2 Správa kódu, spolupráce, build

Jako jednotné úložiště kódu jsme využívali GitHub, které nám také umožnilo snadno na projektu spolupracovat. Celý projekt byl tedy spravován skrze nástroj *git*. Naši práci jsme měli relativně dobře rozdělenou, nemuseli jsme se tedy zabývat pull requesty[18] a publikováním kódu do samostatných větví.

Aby mohla být aplikace správně spuštěna, je potřeba vytvořit po jejím vydání *build*. Buildem se rozumí převedení jejího zdrojového kódu do samostatné funkční aplikace, která je poté připravena fungovat sama o sobě. V našem případě jsme ovšem měli dvě části aplikace – frontend a backend. Při buildu ale bylo potřeba tyto dvě části spojit dohromady, i proto jsou v našem případě součástí společného řešení[19], jak znázorňuje struktura níže.

```
Scoolib
├── scoolib.client                                    frontend
└── Scoolib.Server                                     backend
```

Kód 20: Struktura řešení Scoolib

Kouzlem celého buildu je fakt, že ve skutečnosti probíhá build jen projektu *Scoolib.Server*. Je to z toho důvodu, že ačkoliv nepoužíváme *server-side rendering* a frontend je skutečně plně oddělen od backendu, incializaci frontendu a směrování na něj má přesto v gesci serverová část projektu, v určitém kontextu chápaní tedy backend skutečně frontend spravuje. V souboru

[18] Proces sloučení změn v kódu z několika větví
[19] Virtuální struktura zapouzdřující projekty spolu související

Scoolib.Server.csproj, který je konfiguračním souborem celého backend projektu, odkazujeme na projekt *Scoolib.client*. Během buildu aplikace tedy dojde i k provedení operací definovaných v konfiguračním souboru projektu *scoolib.client*, jejichž součástí je mj. i build této části a její přesunutí do výstupní složky buildu *Scoolib.Server*. Tímto způsobem dojde tedy k zahrnutí souborů z frontendu. V projektu *Scoolib.Server*, konkrétně v jeho vstupním souboru *Program.cs*, je definováno, aby veškeré požadavky, pro které neexistuje definovaný endpoint, byly směrovány na specifické umístění, v našem případě jde o soubor *index.html*, což je vstupní bod celého frontendu (kód 21).

```
app.MapFallbackToFile("/index.html");
```

Kód 21: Přesměrování na vstupní soubor frontend části

3.4.3 Nasazení na server

Aby mohla být aplikace veřejně k dispozici na internetu, bylo potřeba obstarat si server, na kterém by projekt běžel. Používáme Contabo VPS. Na sdíleném serveru máme vyhrazený úložný prostor, včetně výpočetního výkonu a máme kompletní administrátorský přístup do systému serveru. Díky tomu jsme získali vhodné prostředí, kde aplikaci provozovat.

Pro zajištění stability aplikace i serveru samotného jsme se rozhodli pro její *kontejnerizaci*. V dnešní době existují technologie, které dokážou v systému vytvořit izolované prostředí určené pro specifický účel. Do takového prostředí se poté umístí aplikace, což umožňuje chránit ji a ostatní programy od vzájemných vlivů a zásahů. Také to značně zjednodušuje jejich správu a dokáže usměrnit využití systémových zdrojů. Jako nevýhodu lze zmínit vyšší požadavky na úložný prostor, protože každé izolované prostředí obsahuje kromě aplikace samotné i ekosystém okolních služeb a programů, které by jinak mohly být sdílené v rámci celého systému. Přesto jsme si chtěli práci s tímto způsobem nasazování vyzkoušet, a tak jsme pro naši aplikaci úspěšně zprovoznili – pro tyto účely dobře známý – software *Docker*.

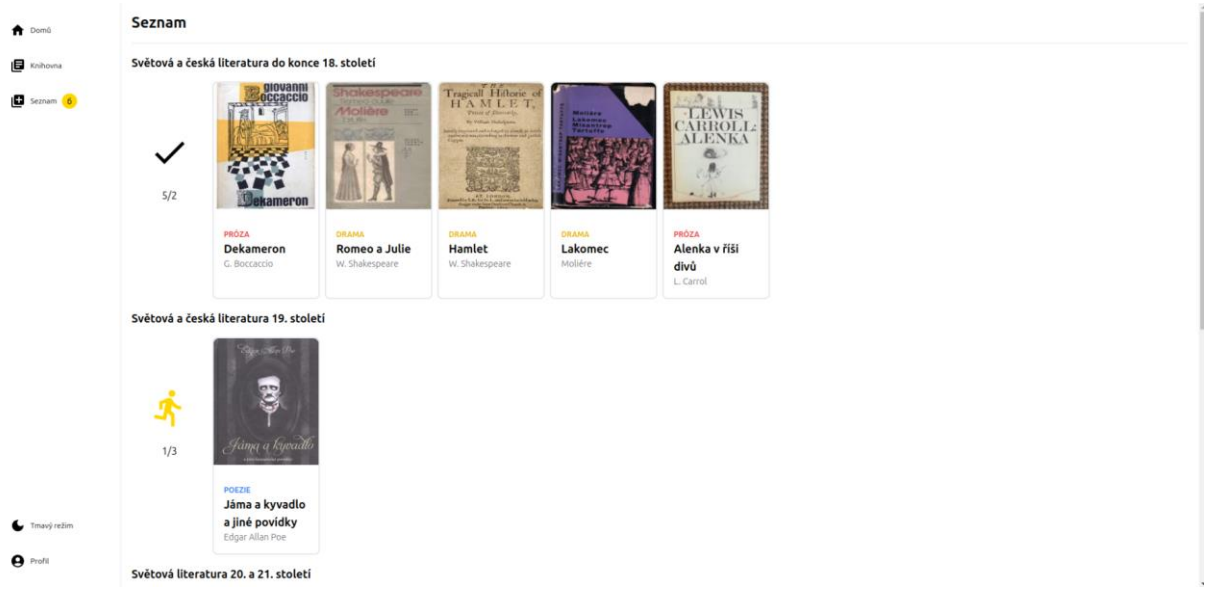
Abychom si usnadnili proces publikování projektu do jeho veřejné verze přístupné z internetu, využili jsme funkcionalitu GitHubu, konkrétně. *GitHub Actions*. To je prostředí, které dokáže samostatně na základě změn v repozitáři reagovat určitými akcemi. Toto prostředí buď běží na serverech GitHubu, nebo je možné si ho nasadit na vlastní server. Provedli jsme instalaci a zprovoznění na našem serveru. Poté již stačilo jen nadefinovat konkrétní úkony a jaká událost je bude spouštět. Pro začátek jsme nastavili, že při jakémkoliv novém příspěvku kódu (commitu) do repozitáře bude zahájen převod změn na server, kde bude proveden nový build[20]

[20] Převedení zdrojového kódu na spustitelnou produkční aplikaci

aplikace a ta následně nahradí instanci staré verze. Vše se tak děje průměrně do dvou minut. Tímto nám odpadla velká část starostí s manuálním nasazováním nových verzí.

3.5 Historie projektu

Myšlenka funkcionality digitalizace seznamu maturitní četby vznikla během únorového hackathonu v roce 2024, kterého jsme se i my zúčastnili. Šlo o jedno z nabízených témat k řešení a náš tým se ho tenkrát ujal. Během 24 hodin jsme vytvořili jednoduchou aplikaci postavenou nad knihovnou React. Backend byl psaný v PHP, obstarával jen nejzákladnější výměnu dat s databází a měl celkově okolo 100 řádků kódu. Vzhled aplikace v tehdejší podobě je vidět na obrázku č. 3.



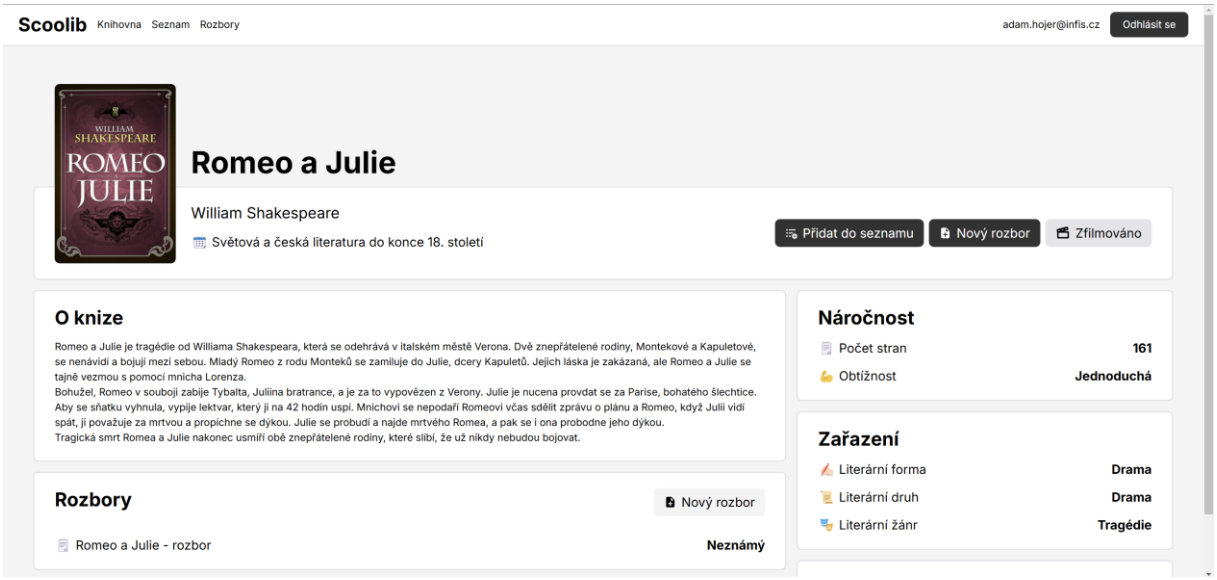
Obrázek 11: První verze aplikace

Ačkoliv naše aplikace zdaleka nedosahovala produkčních kvalit, nápad se nám líbil a chtěli jsme v jeho realizaci pokračovat. Během jara a léta jsme ve dvou připravovali zcela novou verzi aplikace, v tuto dobu již nazývané *Scoolib*. Backend i frontend jsme se rozhodli začlenit pod společný projekt psaný přes framework *Next.js*. Databázi tehdy zastupoval CMS[21] systém *Pocketbase*, jež implementoval základní API pro přístup k datům. Na aplikaci jsme pracovali poměrně dlouhou dobu a čas nad ní strávený byl skutečně znát. Ukázky této verze jsou k vidění na obrázku č. 4. Jak ale aplikace rostla, začali jsme postupně zjišťovat, že se nám nelíbí způsob jejího vnitřního fungování. Psaní backendu a frontendu dohromady bez předchozích zkušeností se nám lehce vymstilo a s každou nově napsanou funkcí bylo znát, že z vývojářského hlediska aplikace nefunguje tak, jak by měla. Jednou z možností bylo provést refactoring[22] kódu, avšak při hlubším zkoumání jsme došli k názoru, že se nám přístup vývoje stanovený *Next.js*

[21] Content management systém – software pro správu dat
[22] Oprava, údržba kódu

frameworkem nelíbí. Určité nedostatky byly jistě zajištěny i tím, že jsme pro ukládání dat využívali *Pocketbase*, který nás hodně omezoval svým definovaným API.

Postupně jsme si tedy stanovili, že aplikaci postavíme znovu pomocí zcela jiných technologií, a tak začal těsně před koncem letních prázdnin roku 2024 vznikat *Scoolib* v podobě, jak ho známe dnes z této práce.



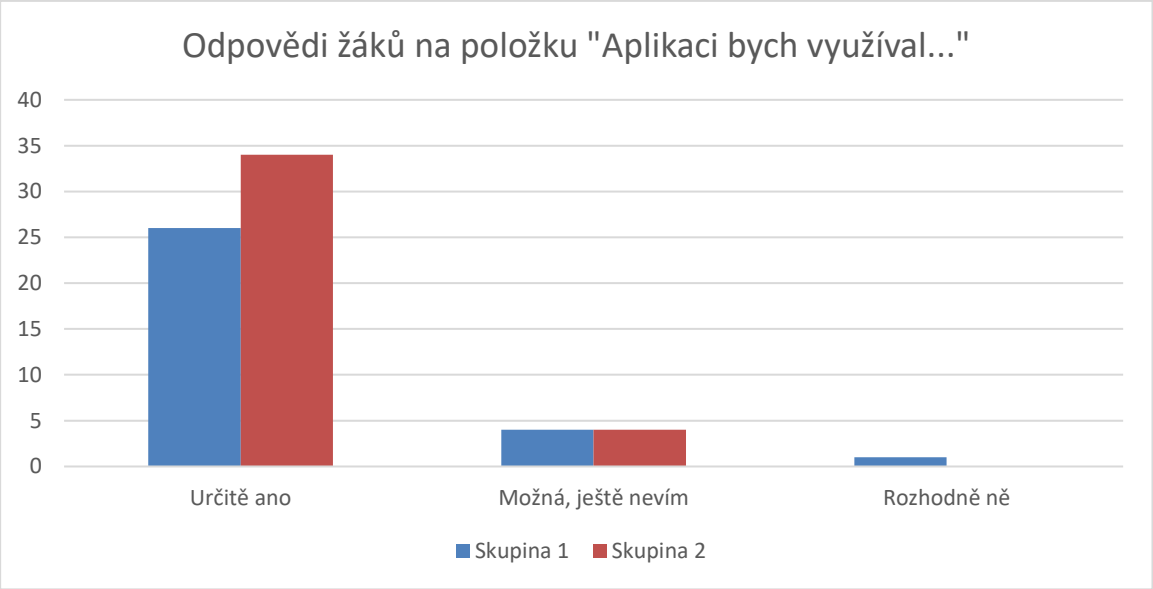
Obrázek 12: Druhá verze aplikace v Next.js

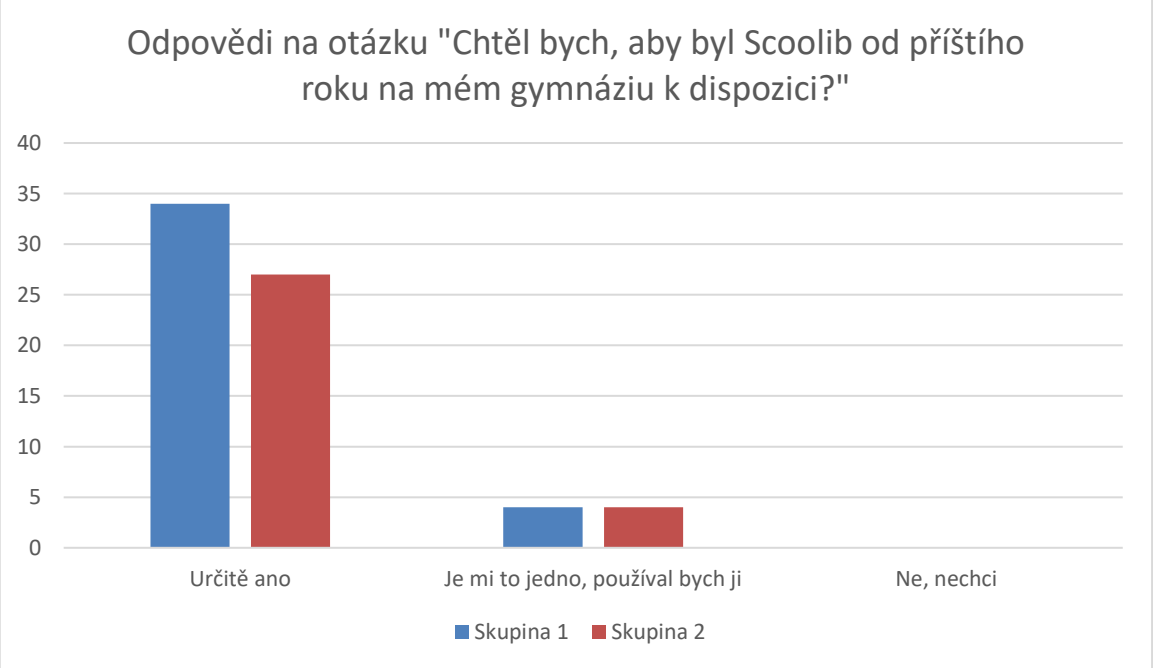
Aplikace *Scoolib* si už prošla třemi fázemi vývoje, přičemž doufáme, že se nacházíme již v té poslední. Tvorba druhé verze, která se po měsících práce zahodila, nás zajisté hodně zdržela. Na druhou stranu nám ale přinesla potřebné zkušenosti, ze kterých jsme nyní mohli čerpat. Při zpětném pohledu jsme za tuto „nevyužitou odbočku“ rádi, protože nás mnohé naučila a díky ní jsme nyní mohli vytvořit skutečně kvalitní produkt, kterému věříme a jsme na něj hrdí.

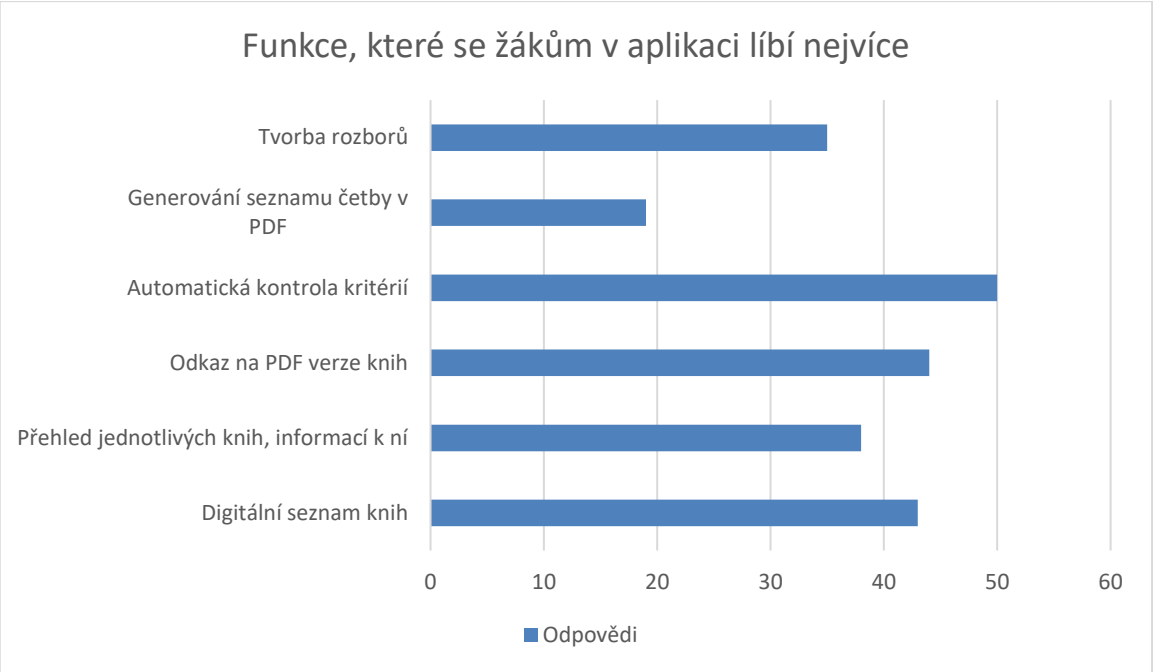
3.6 Využití v praxi

Po dokončení vývoje všech základních funkcionalit jsme začali uvažovat o nasazení aplikace na prvních školách. Kontaktovali jsme pana ředitele Gymnázia Luďka Pika v Plzni a domluvili si schůzku s tamní paní učitelkou literatury. S tou jsme se sešli a probrali možnosti nasazení a využití mezi žáky školy. Nedlouho na to jsme se na gymnázium vydali znovu a pro vybrané žáky třetích ročníků jsme aplikaci na místě odprezentovali. Po ukázce a vyzkoušení jsme žáky požádali o vyplnění dotazníku. Chtěli jsme zjistit, zda mají žáci o nasazení aplikace zájem a co bychom mohli ještě zlepšit pro její největší úspěch.

Níže prezentujeme vybrané výsledky ankety mezi žáky. Proběhly dvě prezentace pro odlišné třídy žáků, z toho důvodu dělíme výsledky na dvě skupiny.







Nadále jsme do dotazníku zahrnuli třeba otevřenou otázku ohledně funkcí a vlastností, které by žáci v aplikaci ještě uvítali nad rámec aktuálních možností.

Zpětná vazba byla pro nás podnětná ve dvou ohledech. Jednak jsme si ověřili, že o aplikaci je mezi žáky zájem a že se jim náš nápad a záměr líbí. Zároveň jsme ale získali cenné informace o nedokonalostech nebo chybějících funkcích, které v aplikaci vidí. Díky tomu jsme dokázali do našich dalších plánů zařadit mnoho nových úkolů, na nichž budeme v brzké době pracovat, a implementujeme tak tunu nových funkcionalit. Půjde třeba o možnost hodnotit knihy i samotnými studenty či vyučujícími. Do přehledu knih také přidáme oddíl informací o autorovi. Při tvorbě rozborů lehce upravíme náš dosavadní interaktivní editor pro to určený a umožníme nahrávat rozbory ve formátech .pdf/.docx.

Tyto všechny skvělé funkce nás čekají a jsme moc rádi, že se jich mj. dočkají právě studenti gymnázia Luďka Pika, kde aplikaci budeme do konce aktuálního školního roku spouštět v testovací fázi.

3.7 Plány do budoucna

Naše aplikace má rozhodně do budoucna potenciál stát se užitečným pomocníkem pro stovky studentů během jejich přípravy na maturitní zkoušku. K tomu jí může dopomoci jak údržba technické části, tak implementace nových funkcí. Pojďme si nyní krátce představit, jaké konkrétní akce bychom rádi učinili pro zkvalitnění aplikace.

3.7.1 Technická část

Z technické části bychom rádi nadále pokračovali ve využívání moderních přístupů k řešení problémů. Naše aplikace má v aktuálním stavu mnoho míst pro zlepšení v této oblasti. Jako příklad lze uvést využívání enumů, které bychom určitě mohli rozšířit na vícero vlastností. S tím souvisí i zaměření se na samotné datové modely, kde bychom měli lépe definovat jejich různé DTO[23] varianty určené pro specifické endpointy. Často využíváme jeden model pro všechny endpointy dané entity, a tak mnohdy pracujeme s nepotřebnými daty, což snižuje celkovou přehlednost a efektivitu.

Na straně frontendu bychom mohli lépe ošetřit správnou synchronizaci stavu v jednotlivých komponentách se serverem. Pokud například přidáme knihu do seznamu četby, po vrácení úspěšné odpovědi ze serveru okamžitě indikujeme nový stav. Měli bychom ale prozkoumat možnosti, jak zajistit, že tento stav skutečně odpovídá stavu databáze, a následně analyzovat, zda je pro účely naší aplikace implementace potřeba.

3.7.2 Funkcionální část

Mladá generace a lidé obecně hojně využívají generativní technologie pro usnadnění práce. Implementací AI do naší aplikace bychom mohli uživatelům pohodlně zpřístupnit generativní funkce v kontextu našich modulů. Žák by se tak mohl například při tvorbě rozboru AI doptávat na podrobnosti, nebo si nechat napsané informace zkontrolovat.

Zajímavou myšlenkou pro nás do budoucna představuje nápad, že bychom naši aplikaci posunuli ze statusu *informující aplikace* na aplikaci *edukativní*. Rádi bychom studentům informace o knihách nejen poskytovali, ale zároveň skrze aplikaci pomáhali s jejich naučením – například formou kvízů. Výrazně by této myšlence pomohlo využití AI. Je však otázkou, nakolik bychom dokázali takové edukační funkce vhodně zahrnout, a zda to není vzhledem k existujícím alternativám zbytečné. Budeme si muset stanovit naše další cíle v širším časovém horizontu a správně zhodnotit, zda je toto tím správným krokem.

[23] A Data Transfer Object – objekt definující datovou strukturu pro konkrétní komunikační účel

4 ZÁVĚR

Výstupem této práce je webová aplikace, jejímž hlavním cílem je pomoci maturujícím žákům během přípravy na maturitní zkoušku z českého jazyka a literatury. Výstupní aplikace správně implementuje definované cíle v zadání, a zároveň je doplňuje o nové funkce plynoucí ze zjištění v diskuzi se žáky. Implementací všech těchto funkcí se podařilo vytvořit skvělý informační zdroj pro žáky a omezit pro ně potřebu dohledávání specifických informací ke každé knize zvlášť, mezi různými informačními zdroji. Aktuální verze aplikace sjednocuje všechny potřebné funkcionality na jedno místo, a představuje tak pro žáky během celého procesu přípravy vždy skvělý výchozí bod.

Vývojový proces představoval pro autory mnoho nových výzev. Vypracování práce jim umožnilo získat detailní pohled do komplexního vývojového procesu. Během něj se detailně zaměřili na každou jeho část – od definice požadovaných funkcí, přes návrh designových prvků, až po psaní kódu a nasazení aplikace. V některých aspektech vývoje autorům pomohly jejich znalosti získané studiem a praxí, přesto bylo stále potřeba učit se fungování využitých technologií a hledat pro takové účely vhodné studijní materiály. Během procesu se mnohdy ukázalo, že použitý postup není pro udržitelnost aplikace ideální, a muselo tedy dojít ke změně přístupu k problému. Vývoj se neobešel ani bez slepých odboček, každá taková zkušenost ale dokázala vývojáře posunout v jejich znalostech a motivovat je k dalšímu postupu. Aktuální verze aplikace reflektuje úsilí autorů o stabilní software, jež využívá moderní přístupy k vývoji. Použitá struktura projektu a kódu umožňuje jednoduchou implementaci dalších funkcionalit, bez potřeby dodatečně měnit stávající funkčnost.

Tvorba aplikace a možnosti jejího praktické využití znamenají výrazný posun ve směru digitalizace seznamů děl k maturitní četbě. Samotná myšlenka digitalizace byla rozvinuta a doplněna o několik klíčových funkcí, i přesto je zde stále potenciál funkce aplikace rozšířit.

5 POUŽITÉ INFORMAČNÍ ZDROJE

1. ***Zákon č. 121/2000 Sb.*** **Česká republika, Česká republika : autor neznámý, 2000.**

2. Howell, Beryl A., [poradci]. ***UNITED STATES DISTRICT COURT.*** **22-1564 (BAH), místo neznámé : United States District Court for the District of Columbia, 18. Srpen 2023.**

3. tdykstra, Rick-Anderson a joostas. Choose between controller-based APIs and minimal APIs. ***Microsoft.*** **[Online] 4. 11 2023. [Citace: 2. 1 2025.] https://learn.microsoft.com/en-us/aspnet/core/fundamentals/apis?view=aspnetcore-9.0.**

4. Belouche, Younes. Code First Approach vs. Database First Approach. ***Medium.*** **[Online] 20. 5 2023. [Citace: 2. 1 2025.] https://medium.com/codex/code-first-approach-vs-database-first-approach-a3830c0cc9b6.**

5. IdentityErrorDescriber Class. ***Microsoft.*** **[Online] [Citace: 3. 1 2025.] https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.identity.identityerrordescriber?view=aspnetcore-9.0.**

6. Introduction to PDFsharp & MigraDoc. ***PdfSharp.*** **[Online] 6.1.0. [Citace: 3. 1 2025.] https://docs.pdfsharp.net/General/Overview/Introduction.html.**

7. BillWagner, pkulikov a Thraka. Enumeration types (C# reference). ***Microsoft.*** **[Online] 8. 4 2023. [Citace: 3. 1 2025.] https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/enum.**

8. Passing Data Deeply with Context. ***React.dev.*** **[Online] [Citace: 3. 1 2025.] https://react.dev/learn/passing-data-deeply-with-context.**

9. Reed, Chris. TypeScript does not add type safety to JavaScript, and other semantic complaints screamed into the void. ***Medium.*** **[Online] 16. 8 2023. [Citace: 3. 1 2025.] https://medium.com/@chrisreed_26015/typescript-does-not-add-type-safety-to-javascript-and-other-semantic-complaints-screamed-into-the-489d603eaeab.**

10. Fetch API. ***mdn web docs.*** **[Online] [Citace: 3. 1 2025.] https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API.**

11. tdykstra, alexbuckgit, guardrex aj. Choose between ASP.NET 4.x and ASP.NET Core. [Online] 2024. [Citace: 02. 01 2025.] https://learn.microsoft.com/en-us/aspnet/core/fundamentals/choose-aspnet-framework?view=aspnetcore-9.0.

12. OpenAI. OpenAI - Terms of Use. [Online] 2024. [Citace: 1. 1 2025.] https://openai.com/policies/row-terms-of-use/.

13. Laurent, Lya. 6 Rules of REST APIs. ***AppMaster.io.*** **[Online] 31. 8 2023. [Citace: 2. 1 2025.] https://appmaster.io/blog/the-six-rules-of-rest-apis.**