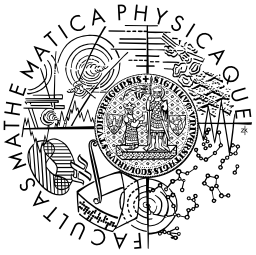


MATEMATICKO-FYZIKÁLNÍ FAKULTA
Univerzita Karlova

BAKALÁŘSKÁ PRÁCE

Vojtěch Lengál

Aplikace pro správu výukových kurzů

Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: doc. RNDr. Jan Kofroň, Ph.D.
Studijní program: Informatika
Studijní obor: Softwarové a datové inženýrství

Praha 2021

Prohlašuji, že jsem tuto bakalářskou práci vypracoval(a) samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů. Tato práce nebyla využita k získání jiného nebo stejného titulu.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V ..... dne ....

Podpis autora

Tímto bych chtěl poděkovat vedoucímu práce doc. RNDr. Janu Kofroňovi, Ph.D. za jeho rady a připomínky.

Název práce: Aplikace pro správu výukových kurzů

Autor: Vojtěch Lengál

Katedra: Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: doc. RNDr. Jan Kofroň, Ph.D., Katedra distribuovaných a spolehlivých systémů

Abstrakt: Cílem práce je vytvořit webovou aplikaci pro správu různých typů výukových kurzů, jako například vysokoškolské přednášky a cvičení, předměty na základní či střední škole nebo také libovolný zájmový kurz. Program bude poskytovat rozhraní jak pro správce – učitele, tak pro uživatele – studenty. Bude umožňovat vytváření a správu kurzů. Správci kurzu budou moci měnit jeho obsah (přidávat, odebírat materiály, apod.) a zakládat nové úkoly a testy, které budou uživatelé následně vyplňovat. Otázky s nabídkou odpovědí bude systém vyhodnocovat automaticky, zbytek bude správce kurzu hodnotit ručně. Uživatel si pak bude moci v daném kurzu prohlédnout všechny známky, a také opravené testy a úkoly.

Klíčová slova: výukový kurz .NET Core Angular

Title: Application for Educational Courses Management

Author: Vojtěch Lengál

Department: Department of Distributed and Dependable Systems

Supervisor: doc. RNDr. Jan Kofroň, Ph.D., Department of Distributed and Dependable Systems

Abstract: Abstract.

Keywords: educational course .NET Core Angular

Obsah

1. Úvod

IT technologie pomáhají v každodenním životě s jeho organizací a dají se využít ke zvýšení komfortu. Přehled informací o studiu, ať už se jedná o základní, střední, či vysokou školu, může výrazně ulehčit orientaci ve větším množství úkolů a studovaných předmětů.

Cílem práce je vytvořit webovou aplikaci pro správu různých typů výukových kurzů, jako například vysokoškolské přednášky a cvičení, předměty na základní či střední škole nebo také libovolný zájmový kurz. Program bude poskytovat rozhraní jak pro správce – učitele, tak pro uživatele – studenty. Bude umožňovat vytváření a správu kurzů. Správci kurzu budou moci měnit jeho obsah (přidávat, odebírat materiály, apod.) a zakládat nové úkoly a testy, které budou uživatelé následně vyplňovat. Otázky s nabídkou odpovědí bude systém vyhodnocovat automaticky, zbytek bude správce kurzu hodnotit ručně. Uživatel si pak bude moci v daném kurzu prohlédnout všechny známky, a také opravené testy a úkoly.

Alternativy jsou např. aplikace Bakaláři a Moodle. Na rozdíl od těchto programů nebude výsledná aplikace tak úzce zaměřená pro školy (bude ji možné snadno použít i např. na jazykové, zájmové kurzy, apod.). Program je vyvíjen jako open-source, zdrojový kód se nachází ve veřejném repozitáři na Githubu.

Program bude fungovat jako single-page aplikace, s rozdělením na serverovou a klientskou část.

2. Použité technologie

V této kapitole se nachází popis technologií, použitých při vývoji aplikace.

2.1 Serverová část

2.1.1 Jazyk C#

C# je objektově orientovaný, staticky typovaný programovací jazyk, vyvinutý firmou Microsoft. C# podporuje koncepty zapouzdření, dědičnosti a polymorfismu. Programy v jazyce C# běží na platformě .NET. Při kompilaci C# programu je zdrojový kód nejprve zkompilován do mezikódu zvaného CIL. Po spuštění programu pak modul CLR, který je součástí platformy .NET, provede JIT (just-in-time) kompilaci IL kódu do strojových instrukcí počítače. Jazyk C# lze využít k tvorbě konzolových aplikací, webových aplikací a stránek, formulářových aplikací ve Windows, softwaru pro mobilní zařízení, apod. [1]

Jazyk C# využíváme v serverové části aplikace. Používáme platformu .NET Core verze 3.1 a jazyk C# verze 8.0.

2.1.2 ASP .NET Core

ASP.NET Core je open source framework, který slouží k vývoji webových aplikací na platformě .NET Core. Aplikace je možné psát v libovolném jazyce, který běží na platformě .NET Core (například v jazyce C#). Jedná se o novější alternativu k frameworku ASP .NET. Součástí frameworku je mimo jiné webový server Kestrel a vestavěný IoC kontejner. [2]

V aplikaci používáme framework ASP .NET Core verze 3.1 v projektu API. Tento projekt funguje jako REST API a slouží k obsluze HTTP požadavků.

2.1.3 Entity Framework Core

Entity Framework Core je ORM framework. Objektově relační mapování je technika, která nám umožňuje objektově pracovat s daty v relační databázi. Framework reprezentuje databázové tabulky pomocí kolekcí, jednotlivé objekty v kolekci pak představují řádky v dané tabulce. Při práci s databází pak vůbec nepoužíváme jazyk SQL, pouze pracujeme s objekty a kolekcemi. [3]

Tento framework využíváme v projektu Data, ve kterém se nachází objekty reprezentující databázové entity, a v projektu *Services*, který obsahuje služby pro komunikaci s databází.

2.1.4 xUnit.net

xUnit.net je framework, který slouží k testování aplikací na platformě .NET. Nejčastěji se používá k unit a integračním testům. [4]

V aplikaci tento používáme xUnit.net v projektu TestEvaluation.Tests, který obsahuje unit testy tříd z projektu TestEvaluation.

2.2 Klientská část

2.2.1 Jazyk TypeScript

TypeScript je programovací jazyk vyvinutý firmou Microsoft. Jedná se o nadstavbu jazyka JavaScript, oba jazyky tedy používají stejnou syntaxi. Kód v jazyce TypeScript se kompiluje do JavaScriptu. Oproti jazyku JavaScript používá statické typování a umožňuje používat třídy, rozhraní a další konstrukce z OOP. [5]

Téměř celá klientská část aplikace (kromě HTML šablon a CSS stylů) je napsána v jazyce TypeScript.

2.2.2 Angular

Angular je frontend framework pro tvorbu single-page webových aplikací. Framework používá jazyk TypeScript a jeho architektura je založená na komponentách. Komponenty jsou jednotky kódu, ze kterých se skládá UI aplikace. Každá komponenta obsahuje TypeScript třídu označenou dekorátorem *@Component()*, HTML šablonu, a případně i soubor s CSS styly. Součástí frameworku je i vestavěný HTTP klient pro snadnější komunikaci pomocí HTTP, router sloužící k navigaci mezi stránkami a velké množství dalších knihoven. Angular nám umožňuje používat obousměrný data binding. [6]

Tento framework využíváme v klientské části aplikace, celá klientská část programu je Angular projekt.

2.2.3 Bootstrap

Bootstrap je CSS framework, který se používá k designu webových stránek. Umožňuje jednoduše vytvářet responzivní layout a obsahuje velké množství definovaných CSS stylů, které lze použít ke stylování HTML elementů. [7]

Bootstrap hojně používáme ke stylování HTML elementů v šablonách Angular komponent.

2.3 Microsoft SQL Server

Microsoft SQL Server je multiplatformní relační databázový systém. Ke komunikaci s databází využívá jazyk T-SQL, což je rozšíření klasického SQL. K administraci Microsoft SQL Serveru a práci s databázemi můžeme použít například aplikaci SQL Server Management Studio. [8]

Microsoft SQL Server používáme jako databázi pro data v naší aplikaci. Jazyk T-SQL vůbec nepoužíváme, jelikož s daty pracujeme pomocí ORM frameworku Entity Framework Core.

2.4 Git

Git je distribuovaný systém správy verzí, který se velmi často používá při vývoji softwaru. Podporuje tzv. větvení, což znamená, že se můžeme odloučit od hlavní linie vývoje a pokračovat ve vývoji v nové větvi, aniž bychom do původní

větve zasahovali. Pomocí příkazu *merge* pak můžeme změny promítnout zpět do hlavní větve. [9]

Při vývoji aplikace používáme několik větví: *master*, *develop* a *feature* větve.

- *Feature* větve používáme při programování nové funkcionality. Typicky máme pro každou funkcionalitu samostatnou větev, jež se ve většině případů jmenuje *feature/popis-funkcionality*.
- *Develop* je větev, do které se provádí *merge* z *feature* větví. Při vývoji nové funkcionality tedy nejprve vytvoříme novou větev a po dokončení vývoje provedeme *merge* do větve *develop*.
- *Master* větev obsahuje verzi aplikace, která je plně funkční a lze spustit. V případě, že dokončíme nějakou ucelenou část aplikace a ověříme, že funguje správně, můžeme provést *merge* z větve *develop* do *master*.

Díky tomuto rozdělení větví můžeme pohodlně pracovat na více částech aplikace zároveň, a máme stále aspoň jednu verzi aplikace, která je plně funkční (ve větvi *master*).

Aplikace je uložená ve veřejném GitHub repozitáři na této URL: `https://github.com/VL-CZ/CourseManagementSystem`.

2.5 GitHub Actions

Technologie GitHub Actions slouží k automatizaci úloh během životního cyklu vývoje softwaru. Používají tzv. CI Workflows, což je automatizovaná procedura, která je typicky spuštěna nějakou akcí (např. přidáním nového commitu). Ke konfiguraci Workflows se používají soubory ve formátu YAML, které se nachází ve složce *.github/workflows*. [10]

V této aplikaci používáme dva Workflows, které se spustí po každém commitu ve větvích *master* a *develop*.

- .NET Workflow - provede build serverové části aplikace a spustí všechny testy
- Angular Workflow - provede build klientské části aplikace

Použitím technologie GitHub Actions si můžeme lehce ověřit, že přidáním nové funkcionality jsme nerozbili nějakou již existující část aplikace. Po přidání nových změn si můžeme jednoduše ověřit, že Workflows proběhly úspěšně.

3. Architektura a implementace aplikace

Aplikace se skládá ze 2 hlavních částí: serverové a klientské.

Na obrázku 3.1 vidíme architekturu aplikace, šipky znázorňují přenos dat. Jednotlivým částem se budeme podrobněji věnovat v této kapitole.

3.1 Serverová část

Serverová část aplikace je naprogramovaná v jazyce C# s použitím frameworku ASP .NET Core. Je rozdělena do těchto projektů:

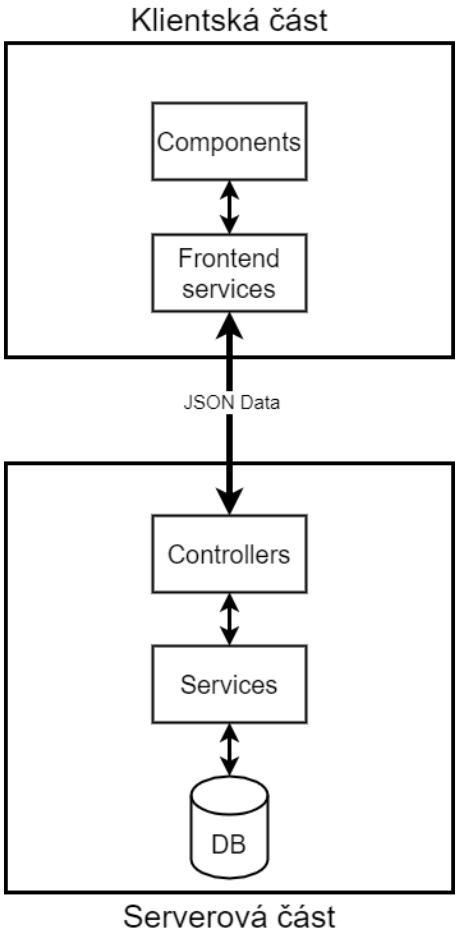
- Data
- Services
- API
- TestEvaluation
- TestEvaluation.Tests

3.1.1 Data

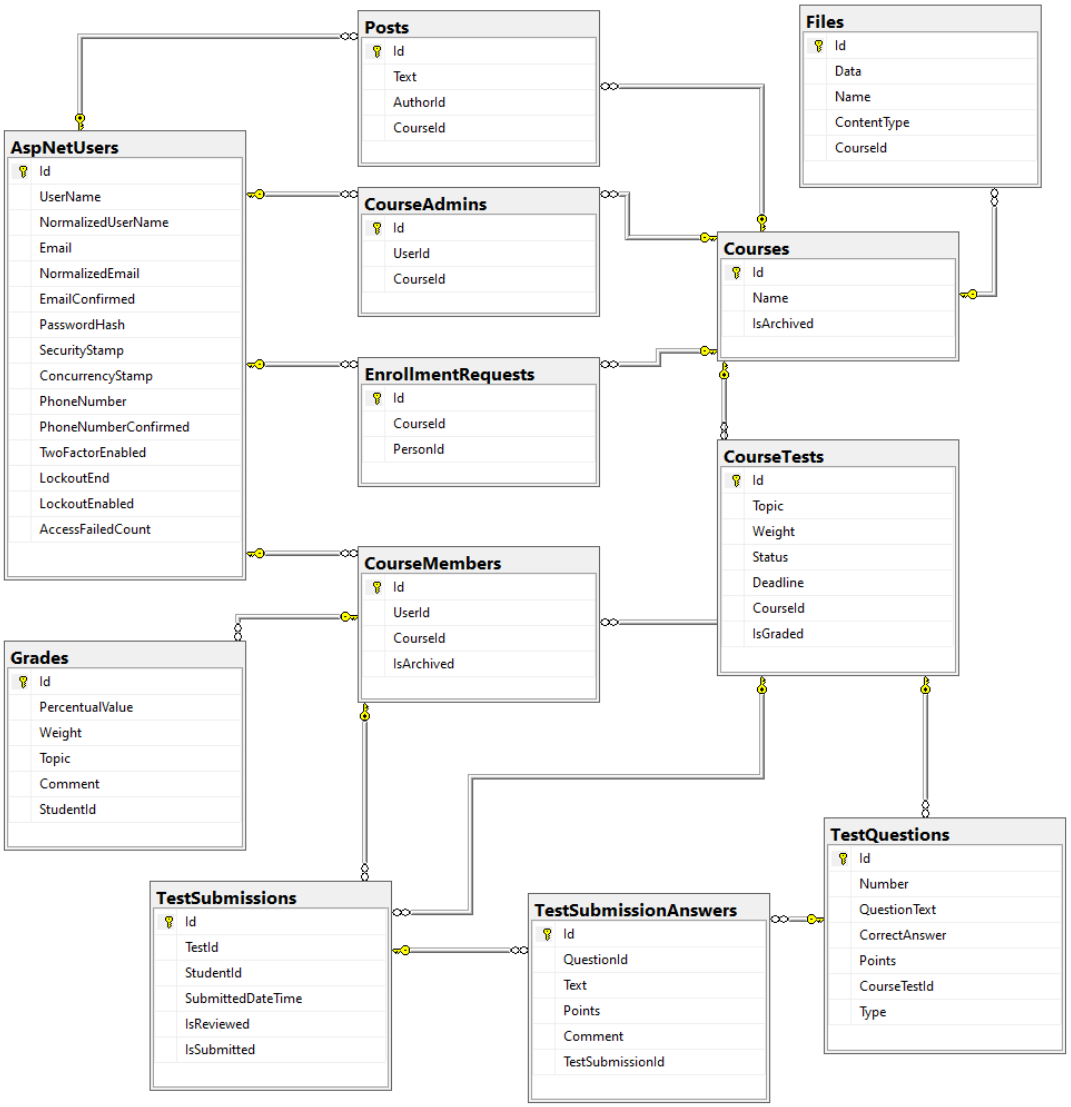
Tento projekt se stará primárně o komunikaci s databází, pro tento účel jsme použili ORM framework Entity Framework Core.

Ve složce *Models* jsou třídy reprezentující databázové entity. Každá z entit pak v databázi představuje jednu tabulku. V aplikaci jsme použili tyto entity:

Název entity	reprezentovaný objekt
Course	kurz
CourseAdmin	vztah administrátor mezi entitami Course a Person
CourseFile	nějaký soubor sdílený v kurzu
CourseMember	členství uživatele v daném kurzu. K této entitě se pak vážou všechny známky a odeslané testy.
CourseTest	test v kurzu. Testy dělíme na hodnocené a nehodnocené (tzv. kvízy).
EnrollmentRequest	žádost o zapsání uživatele do daného kurzu
ForumPost	příspěvek ve fóru k danému kurzu
Grade	známku kterou student obdržel (kromě známek z testů)
Person	uživatele aplikace
TestQuestion	jednu otázku v testu
TestSubmission	test s odpověďmi odeslaný uživatelem
TestSubmissionAnswer	odpověď k dané otázce v testu



Obrázek 3.1: Architektura aplikace



Obrázek 3.2: Databázový model aplikace

Pro ilustraci uvedeme kód třídy *Course*:

Vidíme, že každá entita obsahuje veřejné vlastnosti (public properties) s gettery a settery. Tyto vlastnosti budou v databázové tabulce reprezentovány jako sloupce. Některé z nich (jako např. *Id*) obsahují ještě doplňující atributy, ty slouží k upřesnění informací o dané vlastnosti. Například atribut *[Key]* určuje, že tato vlastnost bude v databázi primární klíč, atribut *[Required]* určuje, že daný sloupec bude v tabulce u všech záznamů povinný (tedy hodnoty budou *NOT NULL*). Dále musí každá entita obsahovat konstruktor bez parametrů.

Vazby mezi entitami jsou reprezentované pomocí tzv. navigačních vlastností. V případě, že chceme vytvořit vazbu typu one-to-many mezi entitami A a B, stačí do vlastností třídy A přidat kolekci objektů typu B, a naopak do třídy B vlastnost typu A. Framework pak při provádění migrace vytvoří v databázové tabulce entity B vytvoří sloupec s cizím klíčem, který bude obsahovat identifikátor entity A, ke které patří.

V aplikaci je vazba one-to-many použita mimo jiné mezi entitami *Course* a

Kód 1 Ukázka třídy Course

```
public class Course : IGuidIdObject
{
    public Course()
    {
        Members = new List<CourseMember>();
        Files = new List<CourseFile>();
        Tests = new List<CourseTest>();
        ...
    }

    public Course(string name, Person admin) : this()
    {
        Name = name;
        Admin = admin;
    }

    /// <summary>
    /// identifier of the couse
    /// </summary>
    [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
    [Key]
    public Guid Id { get; set; }

    /// <summary>
    /// name of the course
    /// </summary>
    [Required]
    public string Name { get; set; }

    ...
```

CourseTest, a to tak, že každý test je obsažen v právě jednom kurzu, a v daném kurzu může být N testů. Kód pak tedy vypadá takto: Ve třídě *Course* je kolekce objektů typu *CourseTest*

```
/// <summary>
/// tests in this course
/// </summary>
public ICollection<CourseTest> Tests { get; set; }
```

Ve třídě *CourseTest* je pak vlastnost typu *Course*

```
/// <summary>
/// course that contains this test
/// </summary>
[Required]
public Course Course { get; set; }
```

Po provedení databázové migrace (viz. dále) se v tabulce *CourseTest* vytvoří sloupec *CourseId* s cizím klíčem, který odkazuje na identifikátor kurzu (tzn. vlastnost *Course.Id*).

Také si můžeme všimnout, že všechny entity (kromě entity *Person*) implementují rozhraní IGuidObject. To je jednoduché rozhraní, které obsahuje pouze jednu vlastnost – *Id* typu *Guid*. Tímto máme zajištěnou jednotu identifikátorů, tedy že všechny entity, které toto rozhraní implementují, budou mít identifikátor typu *Guid*.

```
/// <summary>
/// interface for object with <see cref="Guid"/> identifier
/// </summary>
public interface IGuidIdObject
{
    /// <summary>
    /// identifier of the object
    /// </summary>
    Guid Id { get; set; }
}
```

Typ *Guid* jsme zvolili hlavně z toho důvodu, že vestavěné tabulky frameworku (např. *Identity*) mají také řetězcové identifikátory. Navíc se pak zjednoduší práce ve frontend části (není potřeba parsovat *string* na *int* např. v klientské části při práci s URL). Tyto identifikátory generuje databáze, takže je zajištěno, že jsou unikátní. Další možnost by byla použít jako identifikátor číslo (např. typ *int*), ale vzhledem k výše uvedeným argumentům je typ *Guid* v tomto případě lepší možnost.

Dále se v projektu nacházejí také rozhraní *ICourseReferenceObject* a *ICourseMemberReferenceObject*, která slouží k tomu, abychom mohli dále v aplikaci jednotně pracovat s objekty, které mají referenci na entitu *Course*, resp. *CourseMember*. Tato rozhraní implementují pouze nějaké entity.

Třída *CMSDbContext* reprezentuje databázový kontext této aplikace. Každý objekt typu *DbSet* pak představuje jednu databázovou tabulku. Ve třídě *CMSDbContext* je tedy kolekce typu *DbSet* pro každou z entit.

```
public DbSet<Grade> Grades { get; set; }

public DbSet<Course> Courses { get; set; }

...
```

Jediná výjimka je entita *Person*, která dědí ze třídy *IdentityUser*, a jejíž *DbSet* je nakonfigurovaný ve frameworku.

V programu pak dále používáme ke komunikaci s databází pouze třídu *CMSDbContext* a objekty typu *DbSet*. Třída *DbSet<TEntity>* implementuje rozhraní *IQueryable<TEntity>*, takže na ni lze použít LINQ. Pokud bychom chtěli například vybrat všechny kurzy, jejichž jméno začíná na písmeno C, pak stačí použít následující LINQ dotaz

```
dbContext.Courses.Where(course =>
   course.Name.StartsWith("C"))
```

kde proměnná *dbContext* je instance třídy *CMSDbContext*.

Ve třídě *CMSDbContext* je také metoda *ConfigureForeignKeys*, která provede konfiguraci cizích klíčů v aplikaci. Všechny políčka s cizími klíči jsou v databázi povinné (tzn. *NOT NULL*), to je zajištěno pomocí atributu *[Required]* daných

vlastností. Nastavením *DeleteBehavior.Restrict* u cizích klíčů zajistíme, že databáze zůstane v konzistentním stavu. Pokud bychom tedy chtěli smazat entitu, pak na ni nesmí pomocí cizích klíčů odkazovat jiné entity. V opačném případě program při zavolání metody *SaveChanges()* na databázovém kontextu vyhodí výjimku.

V programu je dále složka *Migrations*. Při vývoji byl použit princip Code first, tedy že v kódu specifikujeme entity pomocí klasických tříd. Framework se pak postará o vytvoření databázových tabulek z tohoto kódu.

Pokud tedy nějak změníme některou z entit (to může být např. přidání vlastnosti, změna jména vlastnosti, apod.), pak pomocí ORM můžeme vygenerovat soubor popisující tzv. databázovou migraci, která slouží k aplikaci změn z kódu do databáze. Ke každé migraci se vygeneruje jeden soubor, který obsahuje popis změn, které se později provedou v databázi.

K vytváření migrací jsem použijeme nástroj CLI tools for Entity Framework Core. [11] Pro vygenerování migrace ze změn v kódu použijeme příkaz:

```
dotnet ef migrations add
```

Tímto se vytvoří soubor popisující změny v migraci, ale databáze zatím zůstala beze změny. Tento soubor obsahuje třídu, jenž dědí ze třídy *Migration* a obsahuje metody *Up* a *Down*. V metodě *Up* je popis změn, které se provedou při aplikaci této migrace, naopak v metodě *Down* je popis změn, které se provedou v případě odstranění migrace.

Jako příklad si můžeme představit migraci, která obsahuje přidání vlastnosti *ScoreWeight* k entitě *CourseTest* (tato vlastnost popisuje váhu testu). Vygenerovaný kód migrace vypadá takto:

```
public partial class TestWeight_added : Migration
{
    protected override void Up(MigrationBuilder
       migrationBuilder)
    {
        migrationBuilder.AddColumn<int>(
        name: "ScoreWeight",
        table: "CourseTests",
        nullable: false,
        defaultValue: 0);
    }

    protected override void Down(MigrationBuilder
       migrationBuilder)
    {
        migrationBuilder.DropColumn(
        name: "ScoreWeight",
        table: "CourseTests");
    }
}
```

Pro promítnutí změn do databáze následně použijeme příkaz:

```
dotnet ef database update
```

Tímto tedy dojde k změnám v databázi (v našem příkladu se vytvoří sloupec *ScoreWeight* v tabulce *CourseTests*).

V obou příkazech je potřeba specifikovat cílový a startup projekt. Cílový projekt je ten, který obsahuje databázový kontext a entity naší aplikace (v tomto případě projekt Data). Naopak startup projekt je projekt, který je spouštěný frameworkem, což je potřeba pro získání konfiguračních informací o projektu, jako je například connection string do naší databáze.

3.1.2 Services

V tomto projektu se nachází pomocné služby pro komunikaci s databází.

Jako základ pro všechny služby slouží abstraktní třída *DbService*, která obsahuje referenci na databázový kontext aplikace a jedinou metodu *CommitChanges()*. Ta slouží k uložení změn provedených v databázovém kontextu do databáze.

To je potřeba, protože k uložení změn do databáze dojde až tehdy, když na databázovém kontextu zavoláme metodu *SaveChanges()*. Pokud bychom tedy například do databázového kontextu něco uložili (např. takto:

```
dbContext.Grades.Add(new Grade())
```

) a nezavolali metodu *dbContext.SaveChanges()*, data by se neuložila.

```
/// <summary>
/// class representing base database service
/// </summary>
public abstract class DbService : IDbService
{
    /// <summary>
    /// context of the CMS database
    /// </summary>
    protected readonly CMSDbContext dbContext;

    /// <summary>
    /// construct a new database service
    /// </summary>
    /// <param name="dbContext">CMS database
       context</param>
    protected DbService(CMSDbContext dbContext)
    {
        this.dbContext = dbContext;
    }

    /// <inheritdoc/>
    public void CommitChanges()
    {
        dbContext.SaveChanges();
    }
}
```

Tato třída implementuje rozhraní *IDbService*, které obsahuje pouze metodu *CommitChanges()*.

Dále jsou ve složce *Interfaces* rozhraní pro další služby, ty jsou rozdělené podle entit (typicky máme pro jednu entitu jednu službu). Všechny tyto rozhraní také implementují rozhraní *IDbService*.

Například rozhraní *ICourseService* (slouží pro práci s kurzy) vypadá takto:

```
public interface ICourseService : IDbService
{
    /// <summary>
    /// get course by its id
    /// </summary>
    /// <param name="courseId">identifier of the
       course</param>
    /// <returns></returns>
    Course GetById(string courseId);

    /// <summary>
    /// archive course by its id
    /// </summary>
    /// <param name="courseId">id of the course to
       delete</param>
    void ArchiveById(string courseId);

    /// <summary>
    /// add the course into the database
    /// </summary>
    /// <param name="course">course to add</param>
    void AddCourse(Course course);
    ...
```

Ve složce *Implementations* jsou potom implementace těchto rozhraní. Můžeme vidět, že všechny implementace dědí ze třídy *DbService*, a zároveň také tranzitivně implementují *IDbService*.

Například třída *CourseService*, která implementuje rozhraní *ICourseService* vypadá takto:

```
public class CourseService : DbService, ICourseService
{
    public CourseService(CMSDbContext dbContext) :
       base(dbContext)
    { }

    /// <inheritdoc/>
    public void ArchiveById(string courseId)
    {
        Course c = GetById(courseId);
        c.IsArchived = true;
    }

    /// <inheritdoc/>
    public Course GetById(string courseId)
    {
        return dbContext.Courses.FindById(courseId);
    }

    /// <inheritdoc/>
    public void AddCourse(Course course)
    {
```

```
            dbContext.Courses.Add(course);
        }
        ...
```

Vidíme, že služby typicky pracují s databázovým kontextem a s daty (vyhledávání, mazání, apod.). Dále si můžeme všimnout, že v žádné metodě se neukládají změny do databáze (tzn. volání metody *CommitChanges()*). To je z toho důvodu, že ve vyšších vrstvách aplikace (např. API) v jedné metodě často voláme několik služeb, příp. několik metod z jedné služby. Pokud bychom v metodách služeb přímo ukládali změny do databáze (metoda *CommitChanges()*), pak bychom se mohli lehce dostat to nekonzistentního stavu. To například tak, že při volání několika služeb v rámci jedné metody by mohla některá ze služeb vyhodit výjimku, nicméně všechny služby zavolané předtím by už data uložily.

Je na zodpovědnosti volajícího provést uložení změn do databáze, tedy zavolat metodu *CommitChanges()*, což je typicky poslední příkaz v dané metodě. Tím jsme tedy zajistili konzistenci dat - buď se do databáze uloží všechny změny provedené v databázovém kontextu, nebo žádné.

Dále si můžeme všimnout, že ve službách pracujeme s identifikátory typu *string*, ale v databázi používáme typ *Guid*. To je z toho důvodu, že ve vyšších vrstvách aplikace se pohodlněji pracuje se stringy (např. často dostáváme ID jako URL parametr). Převod mezi typy *string* a *Guid* pak řešíme ve službách.

Ve složce *Extensions* jsou pak pomocné metody pro práci s některými třídami.

Ve třídě *DbSetExtensions* se nachází extension metody pro třídu *DbSet<T>*. Vybereme si například metodu *GetCourseIdOf*, ta slouží k získání ID kurzu, ke kterému daný objekt, jenž má referenci na entitu *Course*, patří.

```
/// <summary>
/// get id of <see cref="Data.Models.Course"/> that the
   object belongs to
/// </summary>
/// <typeparam name="T">type of items</typeparam>
/// <param name="dbSet">database set</param>
/// <param name="objectId">identifier of the object we
   look for in <paramref name="dbSet"/></param>
/// <returns></returns>
public static string GetCourseIdOf<T>(this DbSet<T> dbSet,
   string objectId) where T : class,
   ICourseReferenceObject, IGuidIdObject
{
    return dbSet.Include(item => item.Course)
        .Single(item => item.Id.ToString() == objectId)
        .Course.Id.ToString();
}
```

Vidíme, že toto je jeden z příkladů použití rozhraní *ICourseReferenceObject*, které se nachází v projektu Data.

V projektu též máme rozhraní *ICourseReferenceService*, to implementují všechny služby, jejichž entity logicky patří k nějakému kurzu. Rozhraní obsahuje jedinou metodu *GetCourseIdOf(string objectId)*, která získá ID kurzu, ke kterému daná entita patří. V implementaci této metodu pak typicky používáme extension metodu *GetCourseIdOf* pro třídu *DbSet<T>*. Například ve třídě *CourseTestService*

vypadá implementace takto:

```
public string GetCourseIdOf(string objectId)
{
    return dbContext.CourseTests.GetCourseIdOf(objectId);
}
```

Podobně je v projektu i rozhraní *ICourseMemberReferenceService*, které používají služby, jejichž entity mají referenci na třídu *CourseMember*.

Použitím služeb jsme odstranili duplikátní kód (např. hledání kurzu podle ID se používá na několika místech ve vyšších vrstvách), a extrahovali některé složitější dotazy do samostatných metod. To je výhodné hlavně z toho důvodu, že je pak můžeme nezávisle otestovat. Další výhoda je, že vyšší vrstvy jsou odstíněny od použití ORM (a databázového kontextu), pouze volají tyto služby.

Dále si můžeme všimnout, že ve službách používáme při komunikaci s databázovým kontextem tzv. Eager loading (pomocí metod *Include()*). To znamená, že data databázových entit které daná entita referencuje, se při výchozím chování nenačtou (pokud nepoužijeme metodu *Include*), tzn. budou mít hodnotu *NULL*.

Například pro získání testu s otázkami můžeme použít tento příkaz:

```
dbContext.CourseTests
    .Include(test => test.Questions)
```

Pokud bychom volání metody Include() vynechali, tak by se data otázek nenačetla.

3.1.3 API

V této části se nachází popis projektu API, které slouží pro komunikaci klientské části se serverovou.

Controllers

Controllery slouží primárně ke zpracování HTTP požadavků. Vidíme, že Controller je klasická C# třída, jež dědí ze třídy *ControllerBase*. Ve většině případů obsahuje reference na služby (jako privátní položky).

```
[Route("api/[controller]")]
[ApiController]
[Authorize]
public class CoursesController : ControllerBase
{
    private readonly IHttpContextAccessor
       httpContextAccessor;
    private readonly ICourseService courseService;
    private readonly IPeopleService peopleService;
    private CourseTestFilter courseTestFilter;
    ...

    public CoursesController(IHttpContextAccessor
       httpContextAccessor, ICourseService courseService,
       IPeopleService peopleService)
    {
```

```
        this.httpContextAccessor = httpContextAccessor;
        this.courseService = courseService;
        this.peopleService = peopleService;
        courseTestFilter = new CourseTestFilter();
        ...
    }
    ...
```

Jednotlivé Controllery pak obsahují veřejné metody, jež odpovídají HTTP endpointům.

Například následující metoda slouží k získaní všech členů daného kurzu.

```
/// <summary>
/// get all course members
/// </summary>
/// <param name="courseId">Id of the course</param>
[HttpGet("{courseId}/members")]
[AuthorizeCourseAdminOf(EntityType.Course, "courseId")]
public IEnumerable<CourseMemberOrAdminVM>
   GetAllMembers(string courseId)
{
    var people =
       courseService.GetMembersWithUsers(courseId);
    return people.Select(cm => new
       CourseMemberOrAdminVM(cm.Id.ToString(),
       cm.User.UserName, cm.User.Email));
}
```

Vidíme, že je tedy označená atributem *HttpGet*, který zároveň obsahuje URL, přes kterou lze tuto metodu zavolat. V tomto případě je URL suffix *{courseId}/members*. Položka *courseId* označuje parametr, jenž má stejnou hodnotu jako proměnná *courseId* (parametr metody). Tento suffix se připojí za URL daného Controlleru, a tím získáme celou URL k zavolání této metody. V našem případě dostaneme
http://host/api/courses/course-id/members, kde *course-id* je identifikátor příslušného kurzu a host označuje URL serveru, na kterém aplikace běží. Pokud máme aplikaci spuštěnou lokálně, pak by hodnota měla být *localhost:5001*. Při HTTP GET dotazu na tuto URL by tedy aplikace zavolala tuto metodu, a vrátila data ve formátu JSON.

Metoda dále obsahuje atribut určený k autorizaci, pomocí něhož ověříme, že klient má dostatečná práva. V tomto případě povolíme zavolat metodu pouze administrátorům daného kurzu.

V těle metody pak zavoláme konkrétní službu, která typicky pracuje s databází. Tato služba nám vrátí data (v tomto případě všechny členy daného kurzu), které potom namapujeme na objekty typu ViewModel, a ty vrátíme.

V dalším příkladu máme metodu, která slouží k vytvoření nového kurzu.

```
/// <summary>
/// create new course
/// </summary>
[HttpPost("create")]
```

```
public void Create(AddCourseVM courseVM)
{
    string currentUserId =
       httpContextAccessor.HttpContext.GetCurrentUserId();
    Person admin = peopleService.GetById(currentUserId);
    Course createdCourse = new Course(courseVM.Name,
       admin);

    courseService.AddCourse(createdCourse);

    courseService.CommitChanges();
}
```

Tato metoda je označena atributem *HttpPost*, který opět obsahuje URL suffix. Pokud bychom tedy chtěli zavolat tuto metodu, pak je potřeba provést HTTP POST požadavek na URL *http://host/api/courses/create*, který v těle obsahuje JSON objekt typu *AddCourseVM*. Framework pak provede deserializaci objektu, a objekt (instanci třídy *AddCourseVM*) předá jako parametr do této metody.

V těle metody následně vytvoříme nový objekt kurzu, a pomocí příslušné služby (v tomto případě *CourseService*) jej přidáme k existujícím kurzům. Můžeme si všimnout, že na konci metody voláme metodu *CommitChanges()*, která změny zapíše do databáze.

ViewModels

ViewModels reprezentují objekty pro komunikaci mezi Frontend a Backend částí (konkrétně mezi Frontend službami a API). Tedy např. při GET dotazu vrací příslušný Controller objekt (příp. objekty) typu ViewModel a framework automaticky provede serializaci do formátu JSON. Stejně tak při POST dotazu posílá klient v těle požadavku objekt typu ViewModel.

Příklad:

```
/// <summary>
/// viewmodel for submitting a test
/// </summary>
public class SubmitTestVM
{
    public SubmitTestVM()
    {
    }

    public SubmitTestVM(string testId, string testTopic,
       bool isSubmitted, IEnumerable<SubmissionAnswerVM>
       answers, bool isTestGraded, DateTime testDeadline)
    {
        TestSubmissionId = testId;
        TestTopic = testTopic;
        Answers = answers;
        IsSubmitted = isSubmitted;
        IsTestGraded = isTestGraded;
        TestDeadline = testDeadline;
```

```
    }

    /// <summary>
    /// id of the test submission
    /// </summary>
    [RequiredWithDefaultErrorMessage]
    public string TestSubmissionId { get; set; }

    /// <summary>
    /// topic of the test
    /// </summary>
    [RequiredWithDefaultErrorMessage]
    public string TestTopic { get; set; }

    /// <summary>
    /// check if the test has already been submitted
    /// </summary>
    public bool IsSubmitted { get; set; }
    ...
```

Tento ViewModel reprezentuje data odevzdávaného testu. Vidíme, že ViewModely jsou typicky veřejné třídy s veřejnými vlastnostmi. Dále vidíme, že ViewModel má veřejný bezparametrový konstuktor, ten je volán při deserializaci dat.

Také si můžeme všimnout, že u některé vlastnosti ještě obsahují doplňující atributy (např. atribut *[RequiredWithDefaultErrorMessage]* u vlastnosti *TestTopic*). Tyto atributy slouží primárně k validaci dat.

Ve ViewModelech poměrně často používáme dědičnost, podobně jako v tomto příkladu.

```
/// <summary>
/// base viewmodel for course tests
/// </summary>
public abstract class BaseCourseTestVM
{
    protected BaseCourseTestVM()
    {
    }

    protected BaseCourseTestVM(int weight, string topic,
       ...)
    {
        ...
    }

    /// <summary>
    /// weight of the score from the test (e.g. test of
       weight 2 has twice bigger impact on overall score
       than test of weight 1)
    /// </summary>
    [PositiveIntValue]
    public int Weight { get; set; }
```

```
    /// <summary>
    /// topic of the test
    /// </summary>
    [RequiredWithDefaultErrorMessage]
    public string Topic { get; set; }

    ...
}

/// <summary>
/// viewmodel for adding a course test
/// </summary>
public class AddCourseTestVM : BaseCourseTestVM
{
    public AddCourseTestVM() : base()
    { }
}

/// <summary>
/// viewmodel representing a test in a course
/// </summary>
public class CourseTestDetailsVM : BaseCourseTestVM
{
    public CourseTestDetailsVM() : base()
    { }

    public CourseTestDetailsVM(string id, string topic,
       int scoreWeight, TestStatus testStatus, ...)
    : base(scoreWeight, topic, ...)
    {
        Id = id;
        Status = testStatus;
    }

    /// <summary>
    /// id of the test
    /// </summary>
    [RequiredWithDefaultErrorMessage]
    public string Id { get; set; }

    /// <summary>
    /// status of the test
    /// </summary>
    public TestStatus Status { get; set; }
}
```

Chceme vytvořit alespoň dva ViewModely, jejichž struktura je velmi podobná a liší se jen přítomností několika málo doplňujících vlastností. Řešením je vytvořit Base třídu, která obsahuje všechny společné vlastnosti. Pro konkrétní ViewModely poté můžeme vytvořit třídy, které dědí z této rodičovské třídy, a obsahují již pouze doplňující vlastnosti.

Ve výše uvedeném příkladu máme tedy třídu *BaseCourseTestVM*, a od ní

dědí třídy *AddCourseTestVM* a *CourseTestDetailsVM*. Tyto ViewModely slouží k přidání nového testu, resp. získání informací o daném testu.

Vidíme, že vlastnosti *Id* a *Status* se nachází pouze ve třídě *CourseTestDetailsVM*. Toto dává smysl, jelikož při vytváření testu ještě neznáme jeho identifikátor (objekt ještě není uložený v databázi). Podobně je to i se stavem (vlastnost *Status*), jelikož ten bude při vytváření vždy nastaven na *New*.

Bylo by samozřejmě možné oba ViewModely sloučit do jedné třídy se všemi vlastnostmi. Toto řešení je ale podle mého názoru velmi matoucí, jelikož vlastnosti *Id* a *Status* jsou při vytváření testu zbytečné.

Atributy určené k validaci

V programu používáme několik vlastních atributů, určených k validaci dat, ty se nachází ve složce *Validation/Attributes*. Atribut je klasická C# třída, která dědí ze třídy *System.Attribute*. Její název obvykle končí suffixem *Attribute*.

Příklady:

```
/// <summary>
/// Validation attribute that marks this value as required
/// <br/>
/// if validation fails <see cref="defaultErrorMessage"/>
   is displayed
/// </summary>
public class RequiredWithDefaultErrorMessageAttribute :
   RequiredAttribute
{
    /// <summary>
    /// default error message displayed ({0} is replaced
       by the field name that this attribute belongs to)
    /// </summary>
    public const string defaultErrorMessage = "The field
       {0} is required";

    public RequiredWithDefaultErrorMessageAttribute()
    {
        ErrorMessage = defaultErrorMessage;
    }
}

/// <summary>
/// Validation attribute that validates if the double
   value is non-negative (e.g. >=0)
/// <br/>
/// if validation fails <see cref="defaultErrorMessage"/>
   is displayed
/// </summary>
public class NonNegativeDoubleValueAttribute :
   RangeAttribute
{
    /// <summary>
    /// default error message displayed ({0} is replaced
```

```
        by the field name that this attribute belongs to)
    /// </summary>
    public const string defaultErrorMessage = "The field
        {0} must be non-negative";

    public NonNegativeDoubleValueAttribute() : base(0,
        double.MaxValue)
    {
        ErrorMessage = defaultErrorMessage;
    }
}
```

- *NonNegativeDoubleValueAttribute* – Tento atribut validuje, že vlastnost typu *double*, ke které patří, je nezáporné číslo. V případě, že tomu tak není, nastavíme výchozí chybovou hlášku *The field {0} must be non-negative*, kde {0} je název vlastnosti, ke které atribut patří.
- *RequiredWithDefaultErrorMessageAttribute* – Tento atribut dědí z atributu *RequiredAttribute*. Pokud má příslušná vlastnost hodnotu *null*, nebo se jedná o řetězec, který je prázdný, nebo složený pouze z bílých znaků, tak validace selže. Atribut *RequiredWithDefaultErrorMessage* pak slouží jen k přidání výchozí chybové hlášky *The field {0} is required*, v případě, že validace selže.

Dále jsou v aplikaci ještě použité atributy *NonNegativeIntValueAttribute* a *PositiveIntValueAttribute*.

Obsluha chyb

V případě, že nastane chyba při deserializaci dat (např. klient pošle v těle POST dotazu nevalidní data), tak framework vyhodí výjimku a vrátí objekt, který popisuje chybu. [2]

Pokud nastane chyba při obsluze dotazu (např. klient se pomocí GET dotazu pokusí získat informace o nějaké již smazané entitě), pak v programu vyhodíme výjimku. Všechny výjimky jsou pak zachyceny třídou *ErrorHandlerController*, která funguje jako obecný Error handler. Toto je nakonfigurováno v metodě *Configure()*, jenž se nachází ve třídě *Startup*.

```
/// <summary>
/// controller for error handling
/// </summary>
[ApiExplorerSettings(IgnoreApi = true)]
public class ErrorHandlerController : ControllerBase
{
    private const string generalErrorText = "An error
        occurred while processing the request.";

    /// <summary>
    /// handle runtime error
    /// </summary>
    /// <returns></returns>
```

```
    [Route("errorHandler")]
    public IActionResult HandleError()
    {
        var context =
            HttpContext.Features.Get<IExceptionHandlerFeature>();
        var exception = context.Error; // thrown exception

        var errorDescription = new Dictionary<string,
            string[]>
        {
            { "Request failed", new string[] {
                generalErrorText } }
        };

        var errorsVM = new
            ErrorsDictionaryVM(errorDescription);
        return BadRequest(errorsVM);
    }
}
```

Metoda *HandleError* tedy nastaví kód odpovědi na *400 Bad Request*, a vrátí objekt typu *ErrorsDictionaryVM*. Ten v tomto případě obsahuje obecnou hlášku *An error occurred while processing the request.*

Autorizace

V programu máme také server-side autorizaci. Ta slouží k ověření toho, jestli je uživatel oprávněný provést danou akci. Základní entitou pro autorizaci je kurz (entita *Course*). V aplikaci tedy vždy ověřujeme, jestli je aktuální uživatel administrátor (příp. člen) kurzu, ke kterému se vztahuje daná entita (např. *CourseTest*).

K tomuto účelu používáme autorizační atributy, které se nachází ve složce *Auth/Attributes*. Abstraktní třída *CourseBasedAuthorizeFilter* slouží jako rodičovská třída pro všechny filtry, založené na autorizaci pomocí kurzů.

```
/// <summary>
/// filter for authorization based on <see
    cref="Data.Models.Course"/> related entity
/// </summary>
public abstract class CourseBasedAuthorizeFilter :
    IAuthorizationFilter
{
    /// <summary>
    /// type of the course related entity
    /// </summary>
    private readonly EntityType entityType;

    /// <summary>
    /// name of the field in HTTP route that contains the
        id of the entity
    /// </summary>
    private readonly string entityIdFieldName;
```

```
    /// <summary>
    /// factory for <see
       cref="Services.Interfaces.ICourseReferenceService"/>
       services
    /// </summary>
    private readonly ICourseReferenceServiceFactory
       courseReferenceServiceFactory;

    /// <inheritdoc/>
    public void OnAuthorization(AuthorizationFilterContext
       context)
    {
        if
           (context.HttpContext.User.Identity.IsAuthenticated)
        {
            string currentUserId =
               context.HttpContext.GetCurrentUserId();
            string objectId = context.HttpContext.Request
                .RouteValues[entityIdFieldName].ToString();

            var service = courseReferenceServiceFactory
                .GetByEntityType(entityType);
            string courseId =
               service.GetCourseIdOf(objectId);

            if (IsAuthorized(currentUserId, courseId,
               entityType, objectId))
            {
                // authorization passed -> proceed to
                   controller
                return;
            }
        }
        context.Result = new UnauthorizedResult();
    }

    /// <summary>
    /// check if the user is authorized to access the
       course related entity
    /// </summary>
    /// ...
    protected abstract bool IsAuthorized(string
       currentUserId, string courseId, EntityType
       entityType, string entityId);

    ...
}
```

Vidíme, že tato třída obsahuje typ entity (proměnná *entityType*), pomocí které provádíme autorizaci. Dále také název proměnné v URL, která obsahuje identifikátor této entity.

Metoda *OnAuthorization* se automaticky zavolá při autorizaci. V této metodě nejprve zkontrolujeme, že uživatel je přihlášený, poté získáme jeho ID (přes *HttpContext*) a ID dané entity. Poté získáme identifikátor kurzu, ke kterému daná entita logicky patří. Zde využíváme toho, že všechny služby implementují rozhraní *ICourseReferenceService*. Dále zjistíme, jestli má uživatel dostatečná práva. K tomuto účelu slouží abstraktní metoda *IsAuthorized*.

Jedná se o využití návrhového vzoru Template method, kdy implementaci této metody necháme na třídách, které dědí z *CourseBasedAuthorizeFilter*. Společné kroky autorizace nicméně implementujeme v této (rodičovské) třídě.

Pokud zjistíme, že uživatel nemá dostatečná práva, pak pouze vrátíme *UnauthorizedResult*, a požadovaná akce se neprovede.

Dále ještě potřebujeme v aplikaci mechanismus, který nám podle enumu *EntityType* vrátí příslušnou službu. K tomuto účelu používáme třídu *CourseReferenceServiceFactory*.

```
/// <summary>
/// factory for <see cref="ICourseReferenceService"/>
/// </summary>
public class CourseReferenceServiceFactory :
    ICourseReferenceServiceFactory
{
    ...

    private readonly IReadOnlyDictionary<EntityType,
        ICourseReferenceService> dataServices;

    public
        CourseReferenceServiceFactory(ICourseAdminService
        courseAdminService, ICourseMemberService
        courseMemberService, ...)
    {
        dataServices = new Dictionary<EntityType,
            ICourseReferenceService>
        {
            [EntityType.CourseMember] =
                courseMemberService,
            [EntityType.CourseAdmin] = courseAdminService,
            [EntityType.CourseTest] = courseTestService,
            ...
        };
    }

    /// <inheritdoc/>
    public ICourseReferenceService
        GetByEntityType(EntityType entityType)
    {
        return dataServices[entityType];
    }
}
```

Tato třída obsahuje slovník *dataServices*, ve kterém je pod daným klíčem typu *EntityType* uložená příslušná služba. V metodě *GetByEntityType* pak jen vrátíme

položku ze slovníku. Opět využíváme toho, že všechny služby implementují rozhraní *ICourseReferenceService*.

V aplikaci používáme tyto autorizační filtry (všechny dědí ze třídy *CourseBasedAuthorizeFilter*).

- *CourseAdminAuthorizeFilter* – ověřuje, jestli je aktuální uživatel administrátor kurzu, ke kterému patří daná entita. Používáme v akcích, které může provádět pouze administrátor daného kurzu (jako např. vytváření nových testů).
- *CourseAdminOrMemberAuthorizeFilter* – ověřuje, zda je aktuální uživatel administrátor nebo člen kurzu, ke kterému patří daná entita. Tento filtr používáme např. v metodě pro získání všech souborů v daném kurzu.
- *CourseAdminOrOwnerAuthorizeFilter* – podobně jako první filtr slouží k ověření toho, jestli je aktuální uživatel administrátor kurzu, k němuž patří daná entita. Zároveň ale akci povolíme provést, pokud je uživatel vlastníkem dané entity. To znamená, že příslušná entita (např. odevzdaný test - entita *TestSubmission*) se váže k danému uživateli. Toto ověření používáme například v metodě pro získání opraveného testu.

Například třída *CourseAdminOrMemberAuthorizeFilter* vypadá takto:

```
public class CourseAdminOrMemberAuthorizeFilter :
    CourseBasedAuthorizeFilter
{
    private readonly IPeopleService peopleService;

    /// <inheritdoc/>
    protected override bool IsAuthorized(string
        currentUserId, string courseId, EntityType
        entityType, string objectId)
    {
        return
            peopleService.IsAdminOfCourse(currentUserId,
            courseId) ||
            peopleService.IsMemberOfCourse(currentUserId,
            courseId);
    }
    ...
}
```

Ke každému autorizačnímu filtru se pak vztahuje atribut.

```
public class AuthorizeCourseAdminOrMemberOfAttribute :
    TypeFilterAttribute
{
    public
        AuthorizeCourseAdminOrMemberOfAttribute(EntityType
        entityType, string entityIdFieldName) :
        base(typeof(CourseAdminOrMemberAuthorizeFilter))
    {
        Arguments = new object[] { entityType,
            entityIdFieldName };
```

```
    }
}
```

Vidíme, že tento atribut má konstruktor se dvěma parametry, které následně předá autorizačnímu filtru.

Atributy se poté používají následujícím způsobem (v příkladu je atribut použitý na metodě Controlleru, která slouží ke stažení souboru podle daného *fileId*).

```
[HttpGet("{fileId}")]
[AuthorizeCourseAdminOrMemberOf(EntityType.CourseFile,
   "fileId")]
public IActionResult Download(string fileId)
{
    ...
}
```

Tímto způsobem tedy zaručíme, že k této metodě mají přístup pouze administrátoři a členové daného kurzu, ve kterém je tento soubor sdílen. Pomocí parametrů atributu popíšeme, že entita je soubor, a její identifikátor se nachází v proměnné s názvem *fileId* (což je parametr metody).

Dependency Injection

K předávání závislostí v aplikaci používáme mechanismus Dependency injection.

Používáme vestavěný IoC kontejner ve frameworku ASP .NET Core, konfigurace se nachází ve třídě *Startup* v metodě *ConfigureServices*. U každé závislosti uvedeme její rozhraní a implementaci. Závislosti vkládáme do kontejneru ve většině případů pomocí metody *AddTransient*. To znamená, že při každém dotazu na danou službu v kontejneru se vytvoří nová instance.

```
// add dependencies to IoC container
services.AddTransient<ICourseService, CourseService>();
services.AddTransient<IPeopleService, PeopleService>();
services.AddTransient<IGradeService, GradeService>();
...
```

Pokud by naše konfigurace vypadala takto, tak bychom do kontejneru vložili tři závislosti. První z nich má rozhraní *ICourseService* a implementaci *CourseService*, u ostatních závislostí je to obdobně.

Pokud tedy chceme, aby do nějaké třídy, jejíž instance jsou vytvářeny frameworkem, byly dosazeny příslušné závislosti (dependencies), použijeme Constructor injection. V konstruktoru dané třídy vytvoříme pro každou požadovanou závislost jeden parametr, který bude mít stejný typ jako rozhraní dané závislosti. Framework se pak postará o dosazení parametrů podle konfigurace IoC kontejneru.

```
public CoursesController(IHttpContextAccessor
   httpContextAccessor, ICourseService courseService,
   IPeopleService peopleService)
{
    this.httpContextAccessor = httpContextAccessor;
    this.courseService = courseService;
    this.peopleService = peopleService;
```

```
    ...
}
```

V tomto případě tedy framework ví, že třída má tři závislosti, které je potřeba vyhledat v kontejneru. Podle konfigurace poté například do parametru *courseService* dosadí novou instanci třídy *CourseService*.

Dependency injection používáme primárně v Controllerech. Nevytváříme tedy přímo instance služeb (pomocí operátoru new), naopak si služby vyžádáme přes parametry konstruktoru, a framework se postará o jejich správné dosazení.

Řešení pomocí Dependency injection je flexibilnější, jelikož Controller nezávisí na implementaci dané služby, ale pouze na jejím rozhraní.

Ostatní

V souboru *appsettings.json* se nachází konfigurace aplikace ve formátu JSON. Její součástí je i Connection string k databázi.

```
"ConnectionStrings": {
    "DefaultConnection": "Server=.;Database=CMS;
        Trusted_Connection=True;
        MultipleActiveResultSets=true"
}
...
```

Ve výchozím nastavení tedy používáme databázi se jménem *CMS* na lokálním SQL Serveru (znak tečky symbolizuje lokální server).

Ve složkách *Pages* a *Areas/Identity* se nachází stránky, které slouží k přihlášení a registraci uživatele. Tato část kódu je vygenerovaná frameworkem.

3.1.4 Ostatní

V projektu TestEvaluation se nachází třídy, které slouží ke spočítání bodů odevzdaných řešení. Projekt TestEvaluation.Tests pak obsahuje testy těchto tříd.

3.2 Klientská část

Klientská část aplikace se nachází ve složce *API/ClientApp*, a je napsaná v jazyce Typescript s použitím frameworku Angular.

Ve složce *src/api-authorization* se nachází modul pro přihlášení a registraci uživatele. Tato část aplikace je vygenerovaná frameworkem. Hlavní část aplikace je v adresáři *src/app*.

3.2.1 Services

Služby slouží ke komunikaci frontend části s API. Jedna služba typicky odpovídá jednomu Controlleru a jednotlivé metody služby pak slouží k volání metod API.

Všechny služby dědí z abstraktní třídy *ApiService*.

```
/**
* base class for all API services
```

```
*/
export abstract class ApiService {

    /**
    * HTTP client
    * @protected
    */
    protected readonly http: HttpClient;

    /**
    * url of the controller to fetch data (ends with /)
    * @protected
    */
    protected readonly controllerUrl: string;

    /**
    * describes how many times to repeat unsuccessful API
       request
    * @private
    */
    private readonly retryCount: number = 1;

    /**
    * create new ApiService
    * @param http http client
    * @param baseUrl base url of the app
    * @param controllerName name of controller that the
       service fetches data from
    * @protected
    */
    protected constructor(http: HttpClient, baseUrl:
       string, controllerName: string) {
        this.controllerUrl = baseUrl +
           `api/${controllerName}/`;
        this.http = http;
    }

    /**
    * handle failed HTTP response
    * @param response HTTP error response object
    * @private
    */
    private handleError(response: HttpErrorResponse) {
        const errorObject: ApiErrorResponseVM = response;
        return throwError(errorObject.error);
    }

    /**
    * process HTTP response
    *
    * if it succeeds, return it, otherwise retry {@link
```

```
        retryCount}-times, then throw an error
    * @param response observable of the response
    * @private
    */
    private processResponse<T>(response: Observable<T>):
        Observable<T> {
        return response.pipe(
            retry(this.retryCount),
            catchError(this.handleError)
        );
    }

    /**
    * execute HTTP GET request to the given URL
    * @param actionUrl URL of the action (will be added to
        {@link controllerUrl})
    * @protected
    * @returns An Observable of the HTTPResponse, with a
        response body in the requested type
    */
    protected httpGet<T>(actionUrl: string): Observable<T>
        {
        return this.processResponse(
            this.http.get<T>(this.controllerUrl +
                actionUrl));
    }
    ...
}
```

Vidíme, že třída obsahuje URL daného Controlleru, a také pomocné metody pro komunikaci s API. Metoda *processResponse* slouží ke zpracování odpovědi, v případě neúspěchu se pokusí dotaz opakovat. Pokud dotaz opět skončí neúspěchem, pak zavolá metodu pro obsluhu chyb (*handleError*).

Metodu *processResponse* pak využíváme například v metodě *httpGet*, která poté slouží k volání metod API pomocí HTTP GET dotazů. Obdobně třída obsahuje i metody pro jiné typy HTTP požadavků (POST, PUT, DELETE).

Konkrétní služby dědí ze třídy *ApiService*, a obsahují metody, které typicky slouží k jednomu dotazu na API.

```
export class CourseTestService extends ApiService {
    private static controllerName = 'courseTests';

    constructor(http: HttpClient, @Inject('BASE_URL')
        baseUrl: string) {
        super(http, baseUrl,
            CourseTestService.controllerName);
    }

    /**
    * get test by Id
    * @param testId
    */
```

```
    public getById(testId: string):
       Observable<CourseTestDetailsVM> {
        return this.httpGet<CourseTestDetailsVM>(testId);
    }

    /**
    * add new test to the given course
    * @param testToAdd test to add
    * @param courseId Id of the course
    */
    public addToCourse(testToAdd: AddCourseTestVM,
       courseId: string): Observable<{}> {
        return this.httpPost(courseId, testToAdd);
    }

    ...
}
```

Na příkladu vidíme, že služba obsahuje pevně dané URL Controlleru, který volá; a dále několik metod, které volají metody tohoto API Controlleru. Například metoda *getById* slouží k získání testu podle jeho identifikátoru. V těle pouze provedeme HTTP GET dotaz, pomocí generického parametru specifikujeme, že vrácený objekt má být typu *CourseTestDetailsVM*. Obdobně je to u metodu *addToCourse*, která se používá k přidání nového testu do kurzu. Oproti předchozí metodě obsahuje ještě jeden parametr – *testToAdd*, který reprezentuje test, jenž bude vytvořen. Tento objekt pak vloží do těla HTTP požadavku, a odešle. Můžeme si všimnout, že v tomto případě voláme metodu *httpPost* bez generického parametru. To je z toho důvodu, že odpověď je prázdná (tzn. neobsahuje žádný ViewModel).

Vidíme, že tělo metod je typicky velmi krátké, ve většině případů se jedná pouze o HTTP dotaz na danou adresu.

Služby fungují asynchronně, tedy nevrací přímo dané objekty, ale generický typ *Observable<T>*. Zavoláním metody *subscribe* na tomto typu se pak počká na vrácení dat, a klient s nimi může pak dále pracovat. Volání metody *subscribe* v našem programu typicky probíhá až v komponentách.

3.2.2 ViewModels

V klientské části se také nacházejí ViewModely, které slouží ke komunikaci s API. Tyto objekty přesně odpovídají ViewModelům ze serverové části (tzn. obsahují stejné položky).

Na příkladu vidíme ViewModel, který reprezentuje odeslaný test (vidíme, že odpovídá ViewModelu ze serverové části – viz. sekce 3.1.3)

```
/**
* viewmodel for submitting a test
*/
export class SubmitTestVM {
    /**
    * id of the test submission
    */
```

```
    public testSubmissionId: string;

    /**
    * topic of the test
    */
    public testTopic: string;

    /**
    * check if the test has already been submitted
    */
    public isSubmitted: boolean;

    ...
}
```

Pro HTTP GET požadavky frontend služby typicky vrací objekty typu ViewModel, se kterými dále pracujeme v komponentách. Stejně tak například při HTTP POST dotazu je do těla požadavku vložena instance nějakého ViewModelu.

3.2.3 Komponenty

Komponenty představují UI aplikace a slouží k zobrazování dat. Nachází se ve složce *src/app/components*. Každá komponenta je uložená ve vlastním adresáři, který obsahuje minimálně dva soubory: šablonu a backend dané komponenty. V některých případech je ve složce i soubor s CSS styly.

Na obrázku 3.3 vidíme část UI s využitím několika komponent. Každá komponenta reprezentuje jeden logický celek uživatelského rozhraní. Konkrétně v uvedeném příkladu máme tyto komponenty:

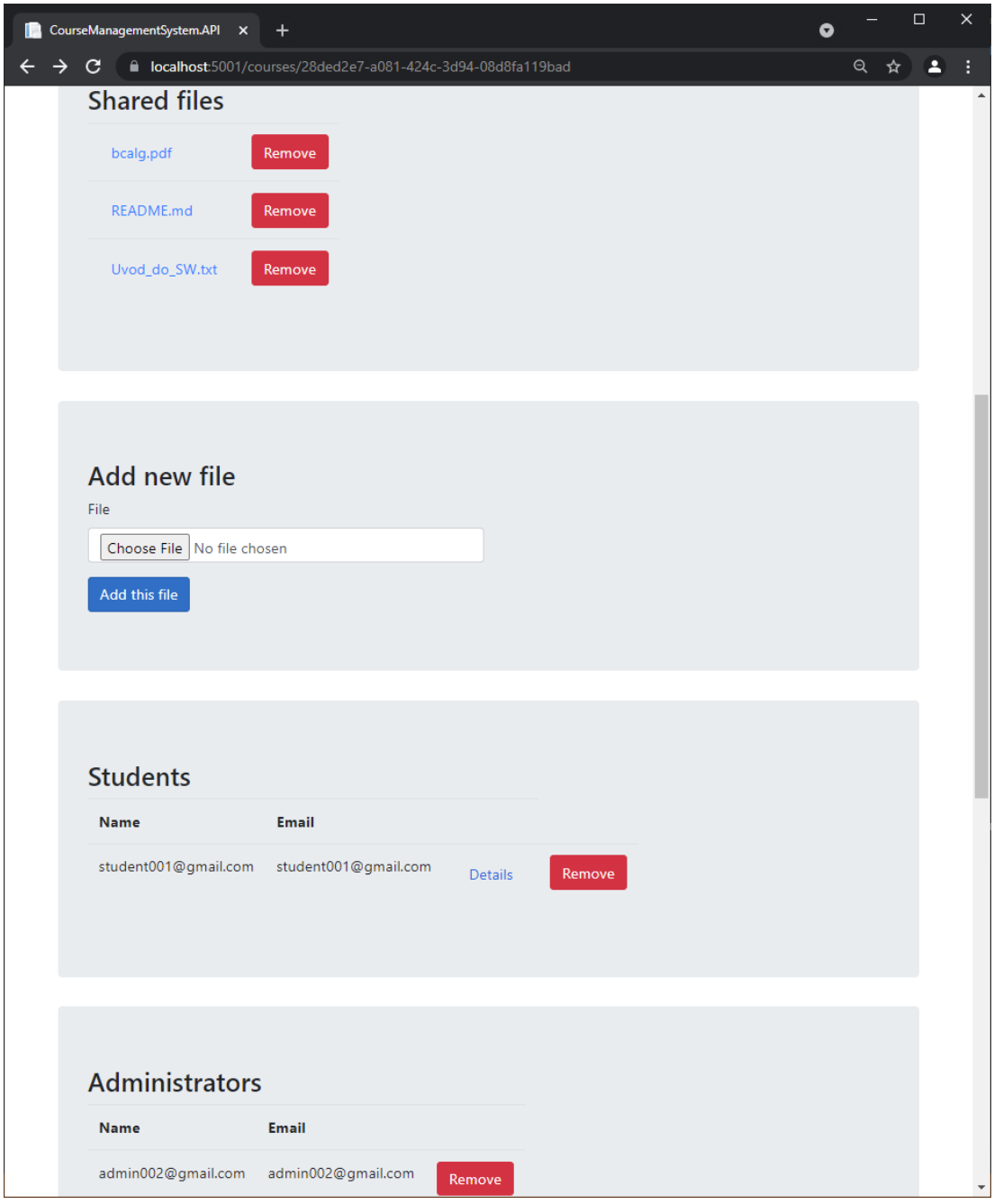
- Komponenta se seznamem souborů + formulářem pro přidání souboru
- Seznam studentů
- Seznam administrátorů

Šablona komponenty je typicky napsaná v HTML, zatímco backend je naprogramován v jazyce Typescript.

```
@Component({
    selector: 'app-student-list',
    templateUrl: './student-list.component.html',
    styleUrls: ['./student-list.component.css']
})
export class StudentListComponent implements OnInit {

    @Input()
    private courseId: string;

    /**
    * list of students
    */
```



Obrázek 3.3: Příklad UI s využitím komponent

```
    public students: CourseMemberOrAdminVM[] = [];

    private readonly courseService: CourseService;
    ...

    constructor(courseService: CourseService, ...) {
        this.courseService = courseService;
        ...
    }

    ngOnInit() {
        this.reloadData();
    }

    /**
    * reload student list
    * @private
    */
    private reloadData(): void {
        this.courseService.getAllMembers(this.courseId)
        .subscribe(result => {
            this.students = result;
        });
    }

    ...
```

Na výše uvedeném příkladu vidíme, že backend komponenty je klasická Typescript třída označená anotací *@Component*. U anotace je zároveň uvedený selektor, což je jméno HTML elementu, který slouží k vykreslení komponenty, a umístění souboru se šablonou a CSS styly.

Komponenty typicky používají služby k získání dat z API, podobně jako v příkladu. Můžeme si všimnout, že právě zde probíhá volání metody *subscribe*, pomocí které v tomto případě data uložíme do proměnné *students*.

Veřejné položky (například pole *students*) jsou pak dostupné v HTML šabloně. Některé položky (například *courseId*) jsou označené anotací *@Input()*, která označuje parametry šablony.

Pro ilustraci uvedeme také šablonu komponenty:

```
<div class="jumbotron">
    <h3>Students</h3>
    <table class="table table-striped table-responsive">
        <tr>
            <th>Username</th>
            <th></th>
            <th></th>
        </tr>
        <tr *ngFor="let person of students">
            <td class="align-middle">{{person.name}}</td>
            <td>
                <a class="btn btn-link"
                   (click)="pageNavigator.
```

```
                    navigateToStudentDetail(person.id)">
                    Details
                </a>
            </td>
            <td>
                <button class="btn btn-danger"
                (click)="removeMember(person)">Remove</button>
            </td>
        </tr>
    </table>
    <div class="my-2">
        <div>
            <button class="btn btn-info"
               (click)="pageNavigator.
                navigateToCourseEnrollmentRequests(courseId)">
                Enrollment requests
            </button>
        </div>
    </div>
</div>
```

Toto je šablona, která přísluší komponentě *StudentListComponent*, jenž slouží k zobrazení členů kurzu. Vidíme, že se jedná o klasické HTML, do kterého můžeme vkládat proměnné z backend části komponenty. Mezi složené závorky vložíme proměnnou, jež chceme vypsat, a framework Angular se sám postará o dosazení příslušné hodnoty.

Můžeme si všimnout atributu **ngFor* u elementu *<tr>*. Takto definujeme, že v tabulce bude pro každého studenta samostatný řádek. Na každém řádku následně vypíšeme jméno, email a odkaz na detail studenta.

V šablonách komponent můžeme také vykreslovat jiné šablony.

```
<app-test-list [courseId]="courseId"
   [isCourseAdmin]="isCourseAdmin"></app-test-list>
<app-file-list [courseId]="courseId"
   [isCourseAdmin]="isCourseAdmin"></app-file-list>
<app-student-list *ngIf="isCourseAdmin"
   [courseId]="courseId"></app-student-list>
...
```

Na tomto příkladu vidíme kus kódu ze šablony komponenty *CourseDetailsComponent*, která slouží k zobrazení informací o daném kurzu. Šablona tedy na daném místě vykreslí příslušné šablony (v tomto případě komponenty se seznamy testů, souborů a studentů).

Při volání šablon můžeme také uvést parametry (například parametr *courseId* v šabloně dané selektorem *app-test-list*). Tato hodnota se pak uloží do příslušné položky, která je označená anotací *@Input()*, v dané komponentě. To znamená, že komponenta se selektorem *app-test-list* má položku *courseId*, která je označená anotací *@Input()*. Pro lepší představu ještě uvedeme kód třídy s komponentou:

```
@Component({
    selector: 'app-test-list',
    templateUrl: './test-list.component.html',
```

```
    styleUrls: ['./test-list.component.css']
})
export class TestListComponent implements OnInit,
   OnChanges {

    @Input()
    public courseId: string;
    ...
}
```

Vzhled stránky se u některých komponent liší podle toho, v jaké je uživateli roli. Můžeme tedy mít komponenty nebo kusy HTML kódu, které se zobrazují například pouze administrátorům daného kurzu. Toto ve většině případů zajistíme pomocí proměnné, podle které poznáme, jestli je aktuální uživatel administrátor. Pomocí atributu **ngIf* pak zajistíme, že daná část kódu se zobrazí pouze vybraným uživatelům.

```
<app-student-list *ngIf="isCourseAdmin"
   [courseId]="courseId"></app-student-list>
```

V tomto příkladu používáme proměnnou *isCourseAdmin*, a pomocí atributu **ngIf* zajistíme, že komponenta určená selektorem *app-student-list* se zobrazí pouze administrátorům daného kurzu.

V komponentách také poměrně často využíváme obousměrný data binding, který je součástí frameworku Angular. Jedná se o techniku, která umožňuje synchronizaci dat mezi HTML a Typescript objekty.

Pokud tedy změníme data v HTML (například pole formuláře), framework se postará o změnu dat příslušné proměnné v backendu komponenty. Naopak, pokud dojde ke změně hodnoty dané proměnné, pak se framework postará o změnu HTML (například upraví text v poli formuláře).

Obousměrný data binding používáme typicky ve formulářích, podobně jako v následujícím příkladu.

```
export class AddGradeComponent implements OnInit {
    /**
    * grade that we will add to the student
    */
    public gradeToAdd: AddGradeVM;
    ...
}
```

Toto je komponenta, která slouží k přidávání známek.

```
...
<div class="form-group col-md-2">
    <label for="addGradeForm_Value">Score in %</label>
    <input type="number" min="0" class="form-control"
       name="value" id="addGradeForm_Value" required
    [(ngModel)]="gradeToAdd.percentualValue">
</div>
<div class="form-group col-md-2">
    <label for="addGradeForm_Weight">Weight</label>
```

```
    <input type="number" min="0" class="form-control"
       name="weight" id="addGradeForm_Weight" required
    [(ngModel)]="gradeToAdd.weight">
</div>
...
```

V šabloně pak máme (mimo jiné) políčka formuláře pro procentuální hodnotu a váhu známky. Pomocí atributu *[(ngModel)]* nastavíme proměnnou pro data binding.

Pokud tedy například změníme ve formuláři váhu známky, dojde automaticky ke změně hodnoty *gradeToAdd.weight*. Podobně, pokud dojde v kódu ke změně hodnoty proměnné *gradeToAdd.weight*, framework automaticky upraví hodnotu ve formuláři.

Ke stylování elementů používáme framework Bootstrap, který nám také zajišťuje, že uživatelské rozhraní aplikace je responzivní.

3.2.4 Routing

Frontend část aplikace mimo jiné provádí také routing. Samotný router je součástí frameworku Angular, nicméně potřebujeme nějak nakonfigurovat, jaká komponenta se zobrazí při dotazu na danou URL.

Tato konfigurace se nachází v souboru app.module.ts.

```
RouterModule.forRoot([
    {path: '', component: HomeComponent, pathMatch:
       'full'},
    {path: 'students/:id', component:
       StudentDetailComponent},
    {path: 'courses', component: CourseListComponent},
    ...
```

V konfiguraci máme vždy uvedenou cestu, a komponentu, která se zobrazí. Vidíme tedy, že na výchozí stránce bude komponenta *HomeComponent*. Druhý řádek nám říká, že při zadání URL *http://host/students/id*, kde *id* je parametr, se zobrazí komponenta *StudentDetailComponent*. K parametru *id* se poté můžeme dostat v komponentě. Poslední řádek definuje, že na stránce *http://host/courses* se zobrazí *CourseListComponent*.

K navigaci mezi stránkami nepoužíváme přímo vestavěný router, ale pomocnou třídu *PageNavigator*, která se nachází ve složce *src/app/tools*. Tato třída obsahuje referenci na třídu *Router* a metody, které slouží k navigaci na jednotlivé stránky. Pro každou stránku máme tedy jednu navigační metodu.

Na následujícím příkladu vidíme kód metody *navigateToCourseDetail*, která slouží k navigaci na stránku s detaily daného kurzu.

```
/**
* navigate to the page with course details
* @param courseId identifier of the course
*/
public navigateToCourseDetail(courseId: string): void {
    this.router.navigate(['/courses',
       courseId]).then(this.scrollToTop);
}
```

3.3 Další vybrané problémy

3.3.1 ViewModely x Databázové entity

Můžeme si všimnout, že v kódu používáme různé objekty pro ViewModely a databázové entity. Na následujícím příkladu můžeme vidět největší rozdíly mezi těmito objekty.

```
public class TestSubmissionAnswer : IGuidIdObject
{
    /// <summary>
    /// question which is answered
    /// </summary>
    [Required]
    public TestQuestion Question { get; set; }

    /// <summary>
    /// submitted text of the answer
    /// </summary>
    public string Text { get; set; }

    /// <summary>
    /// points obtained for the answer
    /// </summary>
    public int Points { get; set; }
    ...
}

public class SubmissionAnswerVM
{
    /// <summary>
    /// number of question that this answer belongs to
    /// </summary>
    [PositiveIntValue]
    public int QuestionNumber { get; set; }

    /// <summary>
    /// text of the question
    /// </summary>
    [RequiredWithDefaultErrorMessage]
    public string QuestionText { get; set; }

    /// <summary>
    /// answer submitted by the student
    /// </summary>
    public string AnswerText { get; set; }
    ...
}
```

Příklad obsahuje databázovou entitu *TestSubmissionAnswer* a ViewModel *SubmissionAnswerVM*, oba typy reprezentují odpověď studenta na danou otázku. Vidíme, že v tomto případě ViewModel obsahuje vlastnosti převzaté z více da-

tabázových entit, což je velmi častá situace. Vlastnosti *QuestionNumber* a *QuestionText* jsou převzaté z entity *TestQuestion*, naopak vlastnost *AnswerText* je z entity *TestSubmissionAnswer*.

Některé jiné ViewModely naopak obsahují jen vybrané vlastnosti příslušných databázových entit, ostatní data jsou v daném kontextu zbytečná.

Z toho vyplývá, že nemůžeme použít instance stejné třídy pro tyto typy objektů.

Pokud bychom se rozhodli použít dědičnost (tedy např. ViewModel by dědil z databázové entity, příp. naopak), narazili bychom na problém s atributy. Téměř všechny databázové entity totiž obsahují atributy, které slouží ke konfiguraci databázových tabulek, naopak ViewModely obsahují atributy pro validaci.

Vzhledem k výše uvedeným argumentům tedy používáme různé objekty.

3.3.2 ViewModely v serverové i klientské části

V kapitolách 3.1.3 a 3.2.2 můžeme vidět, že ViewModely se nachází jak v serverové, tak i v klientské části aplikace.

Tyto ViewModely jsou shodné (obsahují tedy stejné položky), akorát jsou zapsané v jiném jazyce. ViewModely v serverové části navíc často obsahují atributy, které slouží k validaci dat.

Typ položek je ve většině případů stejný, někdy se ovšem může lišit. To je většinou z toho důvodu, že neexistují typy, které si v jazycích C# a Typescript přesně odpovídají. Například jazyk Typescript používá pro všechna čísla typ *number*, zatímco v jazyce C# můžeme použít několik typů. Pro celá čísla používáme např. typy *int* a *long*, pro desetinná čísla *float* či *double*.

Toto řešení je nutné, abychom mohli s ViewModely pracovat v obou částech aplikace.

Závěr

V této práci jsem se zaměřil na vývoj webové aplikace pro správu různých typů výukových kurzů.

Oproti alternativám jako jsou např. aplikace Bakaláři a Moodle není výsledná aplikace zaměřená jen na školní kurzy, ale můžeme ji snadno použít i pro jiné typy kurzů, jako jsou např. jazykové či zájmové kurzy. Každý uživatel si může v aplikaci vytvářet vlastní kurzy.

Aplikace je rozdělená na serverovou a klientskou část, a naprogramovaná s použitím moderních technlologií, jako např. frameworky ASP.NET Core a Angular. Kód programu je uložený ve veřejném repozitáři na GitHubu.

Detailní popis funkcionality se nachází v uživatelské dokumentaci A.3.

Zdroje

[1] Dokumentace jazyka C#. https://docs.microsoft.com/en-us/dotnet/csharp. [Online; citováno 14.7.2021].

[2] Dokumentace frameworku ASP .NET Core. https://docs.microsoft.com/en-us/aspnet/core. [Online; citováno 14.7.2021].

[3] Dokumentace frameworku Entity Framework Core. https://docs.microsoft.com/en-us/ef/core. [Online; citováno 14.7.2021].

[4] Dokumentace frameworku xUnit.net. https://xunit.net/#documentation. [Online; citováno 14.7.2021].

[5] Dokumentace jazyka TypeScript. https://www.typescriptlang.org/docs. [Online; citováno 14.7.2021].

[6] Dokumentace frameworku Angular. https://angular.io/docs. [Online; citováno 14.7.2021].

[7] Dokumentace frameworku Bootstrap. https://getbootstrap.com/docs. [Online; citováno 14.7.2021].

[8] Dokumentace databáze SQL Server. https://docs.microsoft.com/en-us/sql/sql-server. [Online; citováno 14.7.2021].

[9] Dokumentace nástroje Git. https://git-scm.com/doc. [Online; citováno 14.7.2021].

[10] Dokumentace nástroje GitHub Actions. https://docs.github.com/en/actions. [Online; citováno 14.7.2021].

[11] Dokumentace nástroje CLI tools for Entity Framework Core. https://docs.microsoft.com/en-us/ef/core/cli/dotnet. [Online; citováno 14.7.2021].

Seznam obrázků

Seznam použitých zkratek

- HTTP – Hypertext Transfer Protocol
- API – Application Programming Interface
- CIL – Common Intermediate Language
- ORM – Objektově relační mapování
- IoC – Inversion of control
- CLR – Common Language Runtime
- OOP – Objektově orientované programování
- HTML – HyperText Markup Language
- CSS – Cascading Style Sheets
- CI – Continuous integration
- SQL – Structured Query Language
- URL – Uniform Resource Locator

A. Přílohy

A.1 Příručka k použití

Pro spuštění programu je třeba mít nainstalované tyto programy:

- .NET Core SDK verze 3.1
- Node.js verze 12 nebo 14
- lokální instance Microsoft SQL Server databáze
- nástroj CLI tools for Entity Framework Core

V kořenovém adresáři aplikace zadejte do příkazové řádky nejprve následující příkaz.

```
dotnet ef database update --project
   CourseManagementSystem.Data --startup-project
   CourseManagementSystem.API
```

Dojde k vytvoření databáze na lokální instanci SQL Serveru.
Pro spuštění aplikace zadejte tyto příkazy:

```
cd CourseManagementSystem.API
dotnet run
```

Tímto dojde ke spuštění aplikace. Ve výpisu příkazu *dotnet run* na příkazové řádce bychom nyní měli vidět URL, na které se aplikace nachází (typicky je to URL `https://localhost:5001`). Otevřeme webový prohlížeč na této URL a následně se zobrazí domovská stránka aplikace.

A.2 Generování testovacích dat

A.3 Uživatelská dokumentace

Tato kapitola obsahuje uživatelskou dokumentaci aplikace.

Aplikace umožňuje vytváření a správu výukových kurzů, kterýkoliv uživatel si může vytvořit vlastní kurz. Kurz může představovat vysokoškolskou přednášku, cvičení, předmět na střední škole nebo také jazykový, příp. jiný zájmový kurz.

V kurzu jsou dva typy uživatelů:

- Studenti
- Administrátoři (správci)

Role se vztahuje pouze k danému kurzu, uživatel tedy může být v jednom kurzu administrátor, v jiném naopak pouze student. Administrátoři mohou měnit obsah kurzu (přidávat, odebírat materiály, apod.) a vytvářet nové testy. Testy dělíme na dva typy:

- Hodnocené testy
- Nehodnocené testy (kvízy)

Rozdíl je v tom, že kvízy se nepočítají do celkového hodnocení studenta.

Studenti daného kurzu pak mohou testy vyplňovat a zobrazit si své známky i opravená řešení. Systém vyhodnocuje řešení testů automaticky, nicméně správci kurzu mohou následně body upravit ručně.

Testy se skládají z jednotlivých otázek, ty dělíme na tři typy podle druhu odpovědí.

- textová odpověď
- výběr z nabídky odpovědí – právě jedna z nabízených odpovědí je správná
- vícenásobný výběr z nabídky odpovědí – více nabízených odpovědí může být správných

V prvních dvou typech otázek probíhá vyhodnocení jednoduše, pokud se odeslaná a správná odpověď shodují, systém udělí za tuto otázku plný počet bodů, v opačném případě neudělí žádné body. V případě vícenásobného výběru z odpovědí systém nejprve spočítá počet správných voleb (tzn. vybraných odpovědí, které jsou správné a nevybraných odpovědí, které jsou chybné). Dále spočítá procentuální poměr správných voleb a udělí procentuálně daný počet bodů za tuto otázku. Pro ilustraci uvedeme příklad s otázkou, která je hodnocena šesti body, a obsahuje tři možné odpovědi, z nichž dvě jsou správné. Student zaškrtne pouze jednu správnou odpověď (ostatní možnosti nevybere), za otázku tedy dostane $\frac{2}{3} * 6 = 4$ body.

Dále mohou administrátoři přidělovat studentům i známky, které se nevážou k žádnému testu, což můžeme využít například k reprezentaci známek za aktivitu v hodině. Aplikace používá k hodnocení procenta, jednotlivé testy a známky mohou mít různou váhu.

Na obrázku A.1 vidíme uživatelské rozhraní aplikace.

Všechny akce, které zahrnují mazání jsou nevratné – pokud nějaký objekt smažeme, už jej není možné obnovit. Před každou takovou akcí se zobrazí potvrzovací formulář, který vidíme na obrázku A.17. Pro potvrzení akce stiskněte tlačítko *Confirm*, jež vyvolá danou akci. Kliknutím na tlačítko *Cancel* nebo zavřením formuláře se akce zruší, nedojde tedy ke smazání objektu.

Všechna data, která přichází na server jsou validována (pomocí atributů). V případě, že validace selže, aplikace zobrazí formulář s chybami. Příklad tohoto formuláře je na obrázku A.18, v tomto případě jsme se pokusili vytvořit známku se zápornou vahou a bez uvedeného tématu.

Následující podkapitoly obsahují jednotlivé uživatelské scénáře rozepsané v bodech a doplněné snímky obrazovky.

A.3.1 Registrace nového uživatele

- Pro registraci nového uživatele nejprve klikneme na tlačítko *Register* v horním menu.
- Uživatelé v aplikaci jsou identifikováni pomocí e-mailu. Do registračního formuláře zadáme tedy e-mail a heslo nového uživatele, a stiskneme tlačítko *Register*.
- Zobrazí se stránka s potvrzením registrace. Vzhledem k našim potřebám aktuálně neprovádíme ověření e-mailu. Pokud bychom chtěli tuto funkcionalitu přidat, můžeme postupovat například podle návodu v dokumentaci. [2] Klikneme na odkaz s textem *Click here to confirm your account*, a tím je registrace dokončena. Pomocí odkazu pak můžeme přejít na stránku s přihlášením.

A.3.2 Přihlášení uživatele

- Pro přihlášení uživatele nejprve stiskneme tlačítko *Login* v horním menu.
- Zobrazí se přihlašovací formulář, do kterého vyplníme e-mail a heslo, jak můžeme vidět na obrázku A.1
- Po odeslání formuláře nás aplikace přihlásí a přesměruje na stránku se seznamem kurzů.
- Pokud se následně chceme odhlásit, vybereme v menu volbu *Logout*.

A.3.3 Zobrazení stránky se seznamem kurzů

- Po kliknutí na odkaz *Courses* v menu se zobrazí stránka, na které se nachází seznam kurzů. (viz. obrázek A.2)
- Na této stránce se nachází několik sekcí. Sekce *Member courses* obsahuje všechny kurzy, jejichž jsme členy. Po kliknutí na odkaz s názvem kurzu se dostaneme na stránku s detaily kurzu.

- Sekce *Managed courses* obsahuje všechny kurzy, které spravujeme (tzn. jsme administrátoři). Kliknutím na odkaz s názvem kurzu se opět dostaneme na stránku s detaily kurzu. Každý z těchto kurzů můžeme také smazat – kliknutím na tlačíko *Delete* vedle názvu kurzu.
- Dále se na této stránce nachází sekce, které slouží k přidání nového kurzu a zápisu do kurzu (více viz. dále). Úplně dole je pak sekce, ve které můžeme vidět náš uživatelský identifikátor.

A.3.4 Vytvoření nového kurzu

- Na stránce se seznamem kurzů je sekce s názvem *Add new course*, která obsahuje formulář pro přidání nového kurzu.
- Do políčka *Name* vyplníme jméno kurzu a klikneme na tlačítko *Add*.
- Vytvoří se nový kurz, který má právě jednoho administrátora – nás.

A.3.5 Zápis do kurzu

- Nejprve přejdeme na stránku se seznamem kurzů a vybereme sekci *Enroll to a course*.
- K přihlášení do kurzu je nejprve třeba znát jeho identifikátor, ten nám může například zaslat nějaký z administrátorů. Po získání identifikátoru jej vložíme do políčka s názvem *Course ID* a formulář odešleme kliknutím na tlačítko *Enroll*.
- Tímto vytvoříme žádost o přihlášení do daného kurzu. Členem kurzu se staneme až poté, co nám žádost schválí některý z administrátorů.

A.3.6 Zobrazení detailu kurzu

- Po kliknutí na kurz v seznamu kurzů se dostaneme na stránku s detaily.
- Úplně nahoře vidíme sekci s názvem kurzu (viz. obrázek A.3) , ve které se zároveň nachází identifikátor tohoto kurzu, který se používá při zápisu do kurzu. Členové kurzu zde také vidí tlačítko s textem *My Grades*, pomocí kterého se dostanou na stránku s hodnocením studenta.
- Pod touto sekcí se nachází seznam testů, rozdělený do tří kategorií:
  - *Not Published* – tyto testy zatím nebyly publikované, mohou si je zobrazit pouze administrátoři.
  - *Active* – tyto testy jsou aktivní, aktuálně je lze odevzdávat.
  - *After deadline* – testy, které již není možné odevzdat.

  Pokud některá z kategorií neobsahuje žádný test, tak se na stránce vůbec nezobrazuje. Studenti kurzu vidí pouze kategorii *Active*, zatímco administrátorům se zobrazují všechny tři výše uvedené kategorie.

U každého testu je uvedena informace, jestli je hodnocený (*GRADED*, příp. *NOT GRADED*) a případně i deadline. Po kliknutí na název testu se student dostane buď na stránku s odevzdáním, anebo na stránku s vyhodnoceným řešením, pokud již test odevzdal. Administrátory kurzu aplikace přesměruje na stránku s detaily testu.

- Na stránce je dále sekce se sdílenými soubory, které si můžeme stáhnout. Správci kurzu zde vidí i formulář pro nahraní nového souboru.
- Dále se zde nachází komponenty se seznamy členů a administrátorů daného kurzu, které se zobrazují pouze administrátorům (více viz. dále).
- Na stránce dále můžeme vidět fórum s příspěvky. U každého příspěvku je uvedený text a jeho autor. Správci kurzu zde vidí také tlačítko pro smazání příspěvku.
- Pod fórem se nachází formulář, pomocí kterého lze přidávat nové příspěvky.
- Úplně dole je pak tlačítko, které slouží k opuštění kurzu. Tato akce je nevratná, pokud tedy kurz opustíme, nemůžeme původní účet v aplikaci nijak obnovit. Můžeme ovšem zažádat o nové zapsání do kurzu.

A.3.7 Zobrazení detailu studenta

- Na stránku s detaily studenta se můžeme dostat ze stránky s detaily kurzu. Studenti si stránku mohou zobrazit pomocí tlačítka *My Grades*, administrátoři se sem dostanou pomocí kliknutí na detail studenta v sekci s členy kurzu.
- Na obrázku A.4, vidíme, že na této stránce se nachází komponenty s hodnocením daného studenta.
- Nachází se zde tabulka s odevzdanými testy, po kliknutí na název testu se dostaneme na stránku s vyhodnoceným řešením. U každého testu pak vidíme získané skóre, váhu testu (tedy dopad na celkovou známku), datum a čas odevzdání. Poslední sloupec nám říká, jestli bylo odevzdané řešení již revidované
- Pod seznamem odevzdaných testů se nachází komponenta s dalšími známkami. Tyto známky nepatří k žádnému testu.
- Na stránce se dále nachází sekce s celkovým skóre studenta, ve které můžeme vidět vážený průměr všech jeho známek.
- Pod touto sekcí se nachází formulář pro přidávání známek, který se zobrazuje pouze administrátorům.
- Ve spodní části stránky je komponenta s odevzdanými kvízy (tzn. nehodnocenými testy). Po kliknutí na název kvízu nás aplikace přesměruje na detail odevzdaného řešení.

A.3.8 Odevzdání testu

- Nejprve přejdeme na stránku nějakého kurzu, jehož jsme členy.
- V seznamu testů vybereme ten, který chceme odevzdat.
- Pokud jsme tento test zatím neodevzdali, aplikace nás přesměruje na stránku s odevzdáním daného testu, jak můžeme vidět na obrázku A.5. V případě, že jsme test již odevzdali, zobrazí se stránka s vyhodnoceným řešením.
- V horní části vidíme název testu, deadline – čas, do kterého je třeba test odevzdat a několik dalších informací. Test následně vyplníme, u některých otázek máme na výběr z možností odpovědí.
- Stisknutím tlačítka *Save answers* dojde k uložení odpovědí. Můžeme tedy stránku opustit, příp. zavřít a při dalším načtení stránky se načtou i uložené odpovědi.
- Odevzdání testu provádíme pomocí tlačítka *Submit*. Tato akce je nevratná, po odevzdání testu jej není možné nijak upravovat ani znovu odevzdat.
- Po odevzdání testu jej systém vyhodnotí a aplikace nás přesměruje na stránku s vyhodnoceným řešením.

A.3.9 Zobrazení vyhodnoceného testu

- Na obrázku A.6 vidíme příklad stránky s vyhodnoceným řešením testu.
- Ve vrchní části stránky je komponenta s informacemi o testu. Nachází se zde název testu, informace o tom, jestli je test hodnocený a přesný čas odeslání. Pokud byl test již revidován, nachází se zde i řetězec *Reviewed*.
- Na stránce se dále nachází seznam otázek. U každé otázky se zobrazuje text, udělené body, odeslaná a správná odpověď, a případně i komentář opravujícího.
- Pozadí otázky může mít různé barvy, podle toho, kolik bodů student z otázky získal.
  - Zelená – otázka je zodpovězena správně.
  - Červená – otázka je zodpovězena špatně, student z této otázky tedy nezískal žádné body.
  - Oranžová – otázka je zodpovězena částečně správně, student z této otázky tedy získal nějaké body, ale méně než plný počet.
- Pod sekcí s otázkami se nachází komponenta, která slouží k revizi odeslaného řešení a je zobrazena pouze administrátorům.
- V dolní části stánky je sekce s hodnocením, ve které jsou uvedené získané body a skóre v procentech.

A.3.10 Postranní menu s názvem kurzu

- Můžeme si všimnout, že na některých stránkách (například detail testu, detail studenta, jak můžeme vidět na obrázku A.7) se zobrazuje postranní menu s názvem kurzu.
- Pomocí odkazu v tomto menu se můžeme kdykoliv vrátit zpět na stránku příslušného kurzu.

Následující akce mohou vykonávat pouze administrátoři příslušného kurzu.

A.3.11 Administrace členů kurzu

- Přejdeme na stránku daného kurzu.
- Pod sekcí se soubory se nachází komponenty, které obsahují seznamy studentů a administrátorů tohoto kurzu (viz. obrázek A.8).
- Pomocí tlačítka *Details* můžeme přejít na stránku s detaily tohoto studenta, naopak tlačítko *Remove* slouží k odstranění osoby z kurzu.
- Pod seznamem studentů se nachází tlačítko *Enrollment Requests*, které nám umožňuje zobrazit všechny žádostí o zapsání do tohoto kurzu.
- Komponenta s administrátory vypadá podobně, tlačítko *Remove* slouží k odebrání daného administrátora.
- Pod seznamem administrátorů se nachází formulář pro přidání nových administrátorů.

A.3.12 Přidání nového administrátora do kurzu

- Přejdeme na stránku daného kurzu a najdeme sekci s administrátory.
- Nejprve potřebujeme zjistit identifikátor daného uživatele, ten může uživatel získat například na stránce se seznamem kurzů a poté nám jej zaslat.
- Vyplníme tento identifikátor do jediného políčka formuláře a potvrdíme stisknutím tlačítka Add.
- Pokud je požadavek úspěšně dokončen, přidá aplikace tohoto uživatele mezi administrátory kurzu.

A.3.13 Sdílení souboru v kurzu

- Přejdeme na stránku daného kurzu a najdeme sekci se soubory.
- Ve formuláři pro přidání nového souboru klikneme na tlačítko s textem *Choose file*.
- Vybereme soubor, který chceme sdílet a formulář potvrdíme.
- Soubor se následně zobrazí v seznamu sdílených souborů, který je dostupný všem členům kurzu.
- Pokud bychom chtěli soubor následně smazat, klikneme na tlačítko *Remove*.

A.3.14 Potvrzení / zamítnutí žádosti o zápis do kurzu

- Přejdeme na stránku daného kurzu, najdeme sekci se studenty a klikneme na tlačítko *Enrollment requests*.
- Aplikace nás přesměruje na stránku, kde se nachází všechny žádosti o zapsání do tohoto kurzu (viz. obrázek A.9).
- U každé žádosti je uveden e-mail osoby a tlačítka pro potvrzení resp. zamítnutí žádosti.
- Pomocí těchto tlačítek můžeme žádost přijmout nebo zamítnout. Pokud žádost přijmeme, aplikace osobu automaticky přidá mezi studenty daného kurzu.

A.3.15 Vytvoření nového testu

- Přejdeme na stránku daného kurzu, najdeme sekci se seznamem testů a klikneme na tlačítko *Create new assignment*.
- Zobrazí se formulář sloužící k vytvoření testu.
- Do příslušných políček formuláře vyplníme téma, váhu, deadline a počet otázek testu. Je potřeba také zvolit, jestli se jedná o hodnocený test nebo o kvíz. V případě, že chceme vytvořit hodnocený test, zvolíme hodnotu *Yes* v otázce *Is this assignment graded?*.
- Po stisknutí tlačítka *Apply* se zobrazí formuláře pro jednotlivé otázky, jak můžeme vidět na obrázku A.10.
- V případě, že následně chceme změnit počet otázek, změníme hodnotu políčka *Question count* a potvrdíme stiskem tlačítka *Apply*.
- U každé otázky je nejprve potřeba vybrat typ odpovědí. Máme na výběr z těchto hodnot:
  - textová odpověď
  - výběr z nabídky odpovědí
  - vícenásobný výběr z nabídky odpovědí
- Do příslušného políčka vyplníme text otázky. V případě, že se jedná o otázku s výběrem odpovědí, je potřeba vybrat počet možných odpovědí a potvrdit pomocí tlačítka *Apply*. Poté se zobrazí políčka pro jednotlivé možné odpovědi, označené písmenem. Do těchto políček vyplníme text možných odpovědí. Maximální počet možných odpovědí je deset.
- Vyplníme správnou odpověď otázky a počet bodů za tuto otázku. V případě, že se jedná o otázku s výběrem odpovědi, jako správnou odpověď zadáme jedno písmeno – identifikátor možnosti, která je správná.

  Pokud jde o otázku s vícenásobným výběrem odpovědí, do políčka pro správnou odpověď zadáme seznam písmen pravdivých možností, oddělených

čárkou. Tedy například, pokud máme otázku se správnými odpověďmi A a C, do políčka pro správnou odpověď zadáme řetězec *A,C*. Pokud je pravdivá pouze jedna odpověď, zadáme pouze jedno písmeno, naopak pokud není pravdivá žádná z odpovědí, necháme políčka pro správnou odpověď prázdné. Ilustrační příklad je na obrázku A.12.

- Vytvoření testu potvrdíme stisknutím tlačítka *Create* v dolní části stránky.
- Aplikace nás poté přesměruje zpět na stránku příslušného kurzu.

A.3.16 Zobrazení detailu testu

- Přejdeme na stránku daného kurzu, najdeme sekci s testy a klikneme na požadovaný test.
- Pokud jsme administrátorem daného kurzu, aplikace nás přesměruje na stránku s detailem testu.
- Na obrázku A.11 vidíme, že stránka obsahuje informace o daném testu a seznam otázek.
- V případě, že tento test ještě nebyl publikován, se zobrazí tlačítko, které umožňuje editaci.
- Pokud je test již publikován, ve spodní části stránky se zobrazí sekce *Submitted solutions*, která obsahuje seznam odeslaných řešení tohoto testu.
- Na obrázku A.11 vidíme, že u každého odeslaného řešení se zobrazuje e-mail studenta, který řešení odevzdal, skóre, čas odeslání a informace o tom, jestli bylo dané řešení již revidované. Kliknutím na tlačítko *Detail* se dostaneme na stránku s vyhodnoceným řešením.

A.3.17 Editace testu

- Přejdeme na stránku s detailem daného testu a klikneme na tlačítko s textem *Edit*. Testy, které již byly publikované, editovat nelze.
- Zobrazí se formulář, sloužící k úpravě testu. Na obrázcích A.13 a A.14 vidíme, že vypadá téměř stejně jako formulář pro vytvoření testu.
- V případě, že chceme upravit hodnotu deadline, klikneme na tlačítko Set new deadline a vyplníme příslušné hodnoty. Pomocí tlačítka *Cancel* aktualizaci hodnoty deadline zrušíme.
- Upravíme požadované hodnoty, pro přidání nové otázky stiskneme tlačítko *Add new question*, jež se nachází ve spodní části stránky. Pokud chceme nějakou z otázek odebrat, klikneme na tlačítko *Delete question*.
- Úplně dole se nachází tlačítka pro potvrzení, resp. zrušení změn. Pokud chceme změny potvrdit, stiskneme tlačítko *Save*, pro zahození změn stiskneme tlačítko *Discard*.
- V obou případech nás aplikace přesměruje zpět na stránku s daným testem.

A.3.18 Publikace testu

- Přejdeme na stránku daného kurzu do sekce s testy.
- U vybraného testu klikneme na tlačítko *Publish*. Tato akce je nevratná, publikaci testu nelze vrátit.
- Po dokončení požadavku se test přidá do skupiny aktivních testů a studenti mohou začít odevzdávat svá řešení.

A.3.19 Odstranění testu

- Přejdeme na stránku daného kurzu a najdeme sekci s testy.
- U vybraného testu klikneme na tlačítko *Delete*.
- Tím dojde ke smazání daného testu.

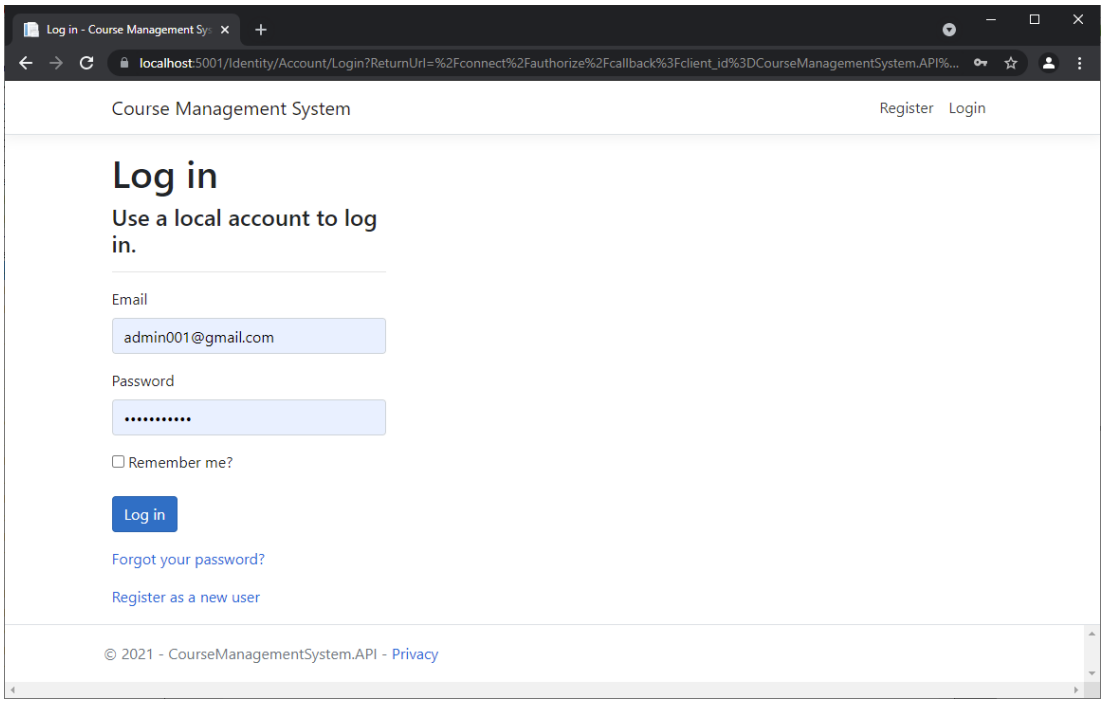
A.3.20 Manuální oprava testu

- Přejdeme na stránku s detailem testu, ve spodní části stránky nalezneme sekci s odeslanými řešeními. Pokud ještě žádný student řešení neodeslal, tato sekce se nezobrazí.
- U vybraného řešení klikneme na tlačítko Detail, tím se dostaneme na stránku s vyhodnoceným řešením.
- Ve spodní část stránky se nachází sekce *Review*.
- V případě, že nechceme upravit body, ani přidat žádné komentáře, klikneme na tlačítko *Mark as reviewed*, tím označíme test jako revidovaný.
- Pokud chceme upravit body, či přidat komentáře, stiskneme tlačítko *Update*. U každé otázky se zobrazí formuláře sloužící k úpravě bodů a přidání komentáře, jak můžeme vidět na obrázku A.15.
- Vyplníme požadované změny (např. změníme počet získaných bodů u vybrané otázky).
- Pod otázkami se nachází tlačítka pro potvrzení, resp. zrušení změn. Pokud chceme změny potvrdit, stiskneme tlačítko *Save*, pro zahození změn klikneme na tlačítko *Discard*.
- Aplikace provede požadovanou akci a změny uloží, resp. zahodí.

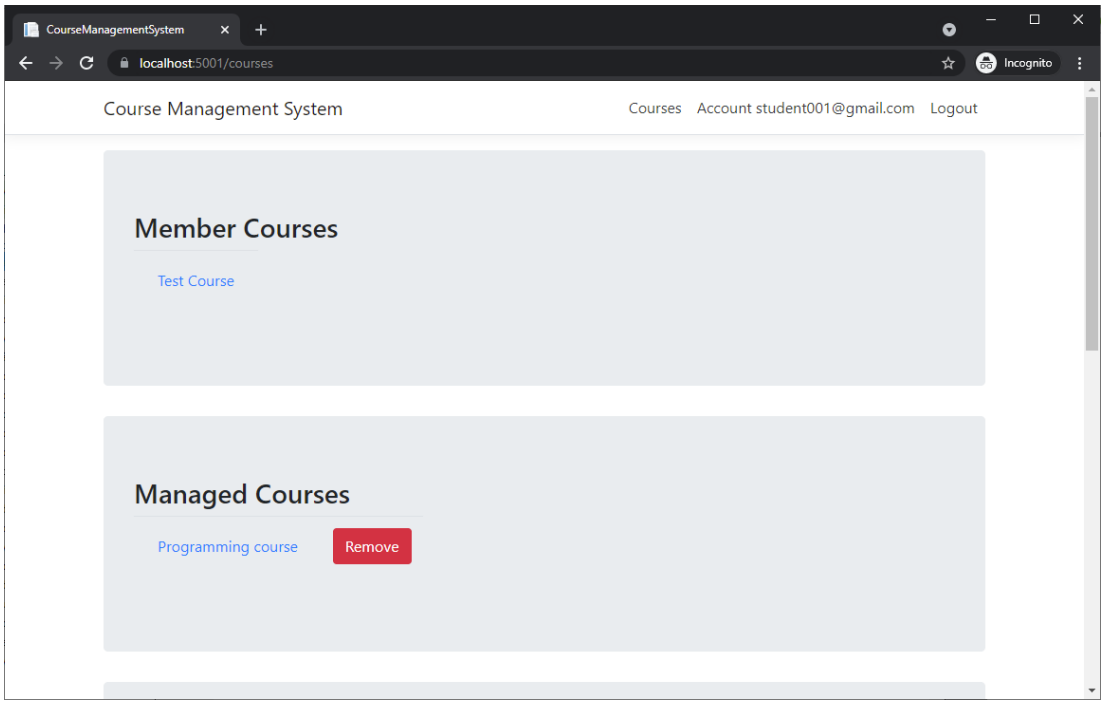
A.3.21 Vytvoření nové známky

- Přejdeme na stránku s detailem studenta.
- Pod komponentou s celkovým skóre se nachází formulář pro přidání nové známky (viz. obrázek A.16).
- Vyplníme téma, skóre, váhu a případně i komentář a následně formulář potvrdíme.

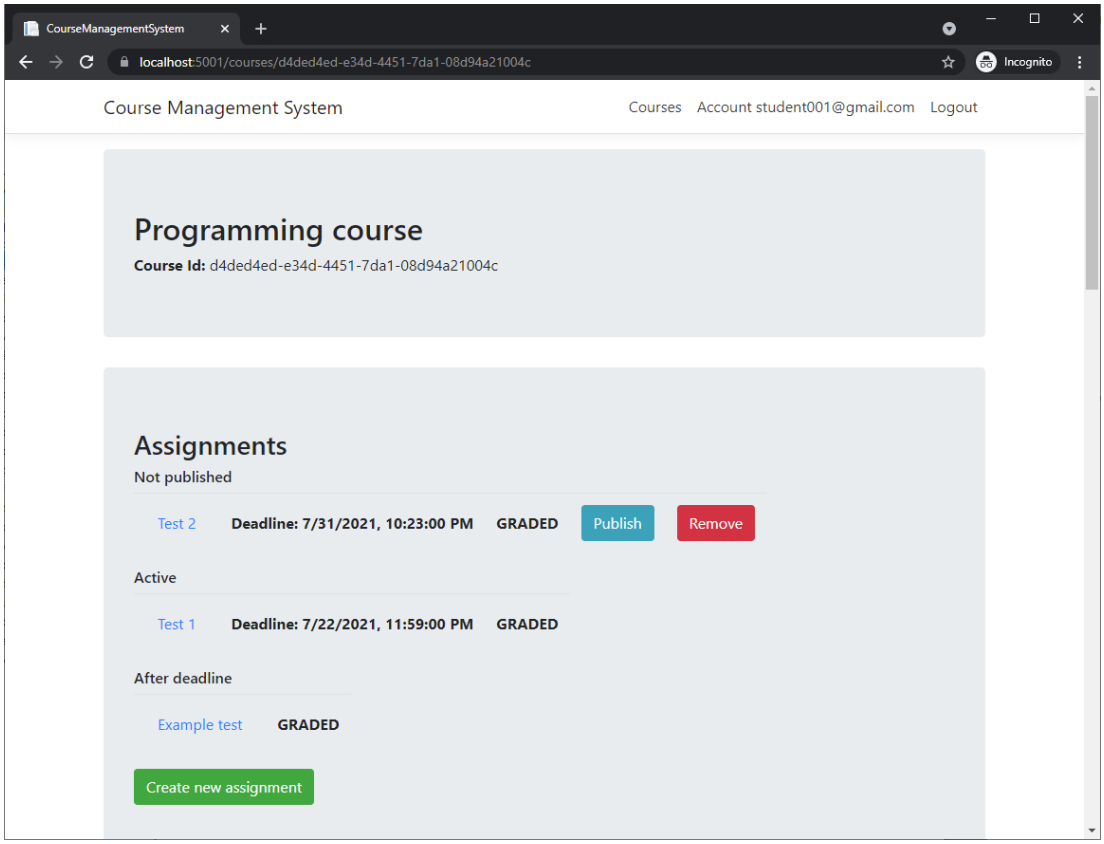
- Dojde k vytvoření nové známky, která se přidá do tabulky známek.
- Pokud chceme nějakou ze známek následně smazat, stiskneme příslušné tlačítko *Remove*.



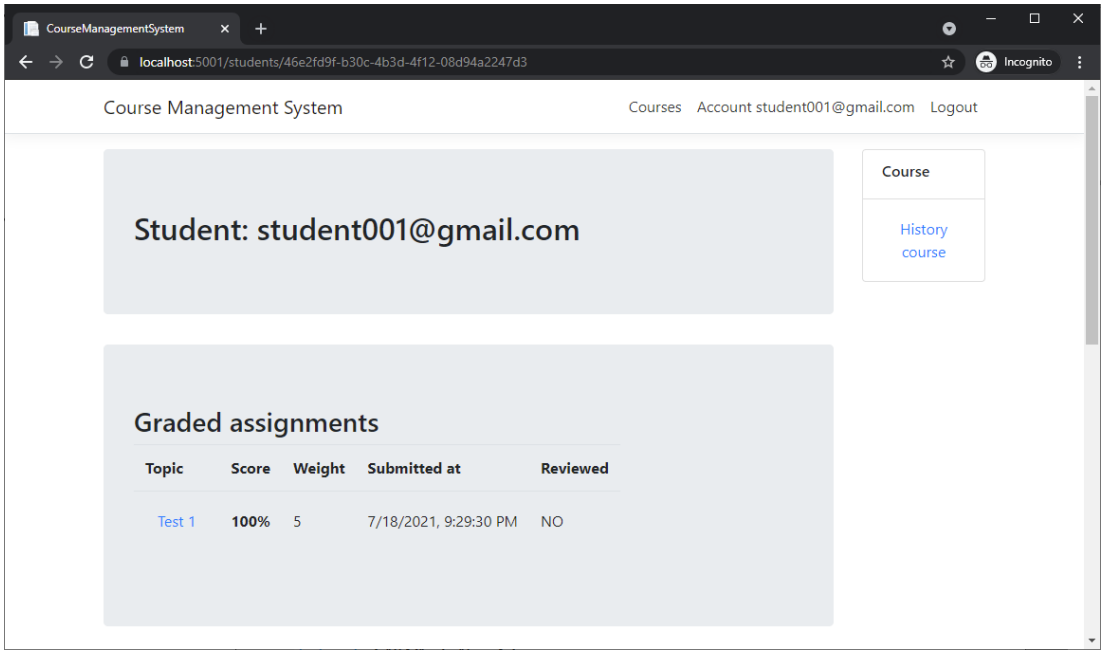
Obrázek A.1: Stránka s přihlášením



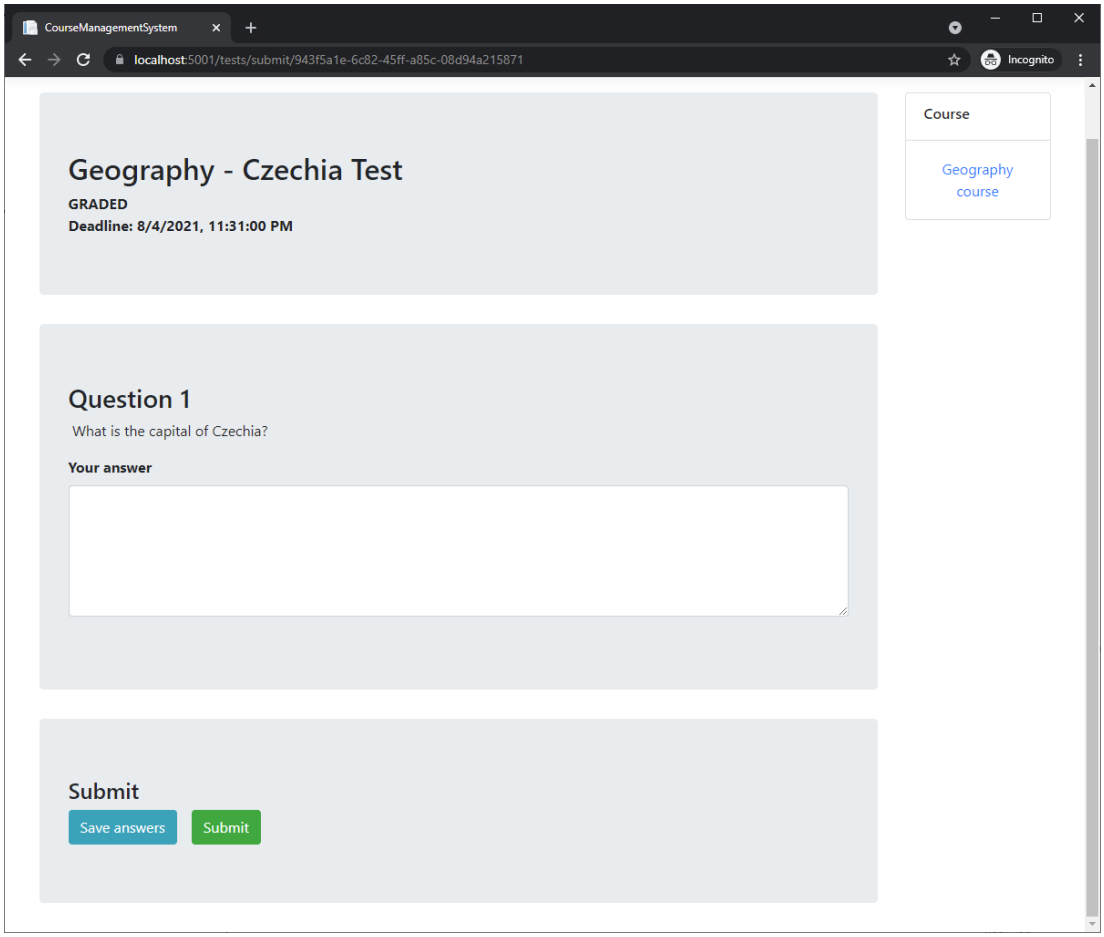
Obrázek A.2: Stránka se seznamem kurzů



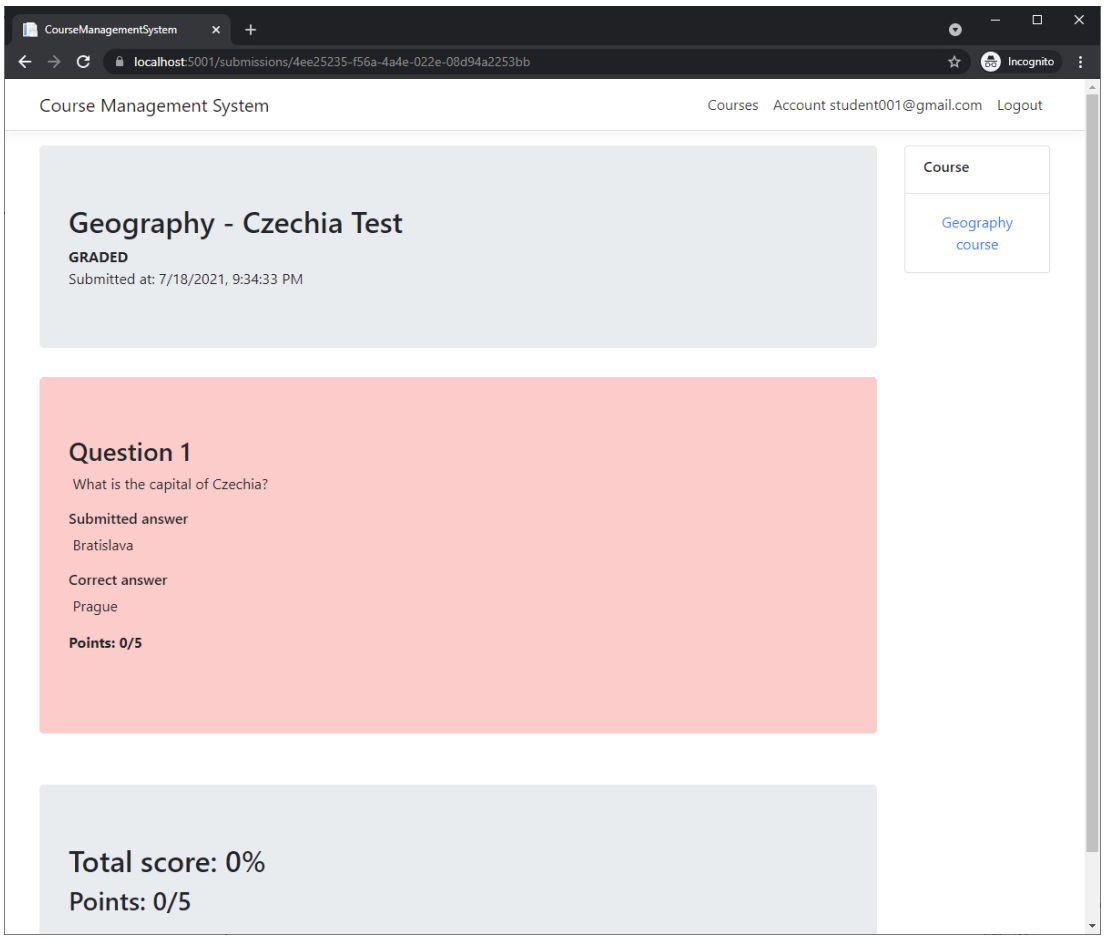
Obrázek A.3: Stránka s detailem kurzu



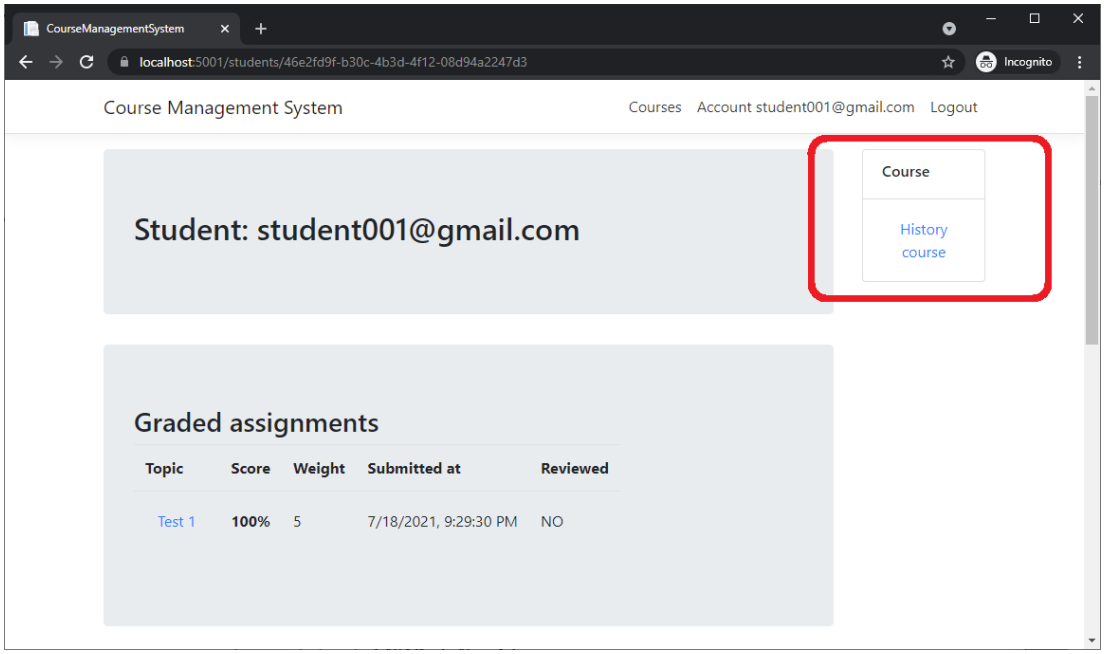
Obrázek A.4: Stránka s detailem studenta



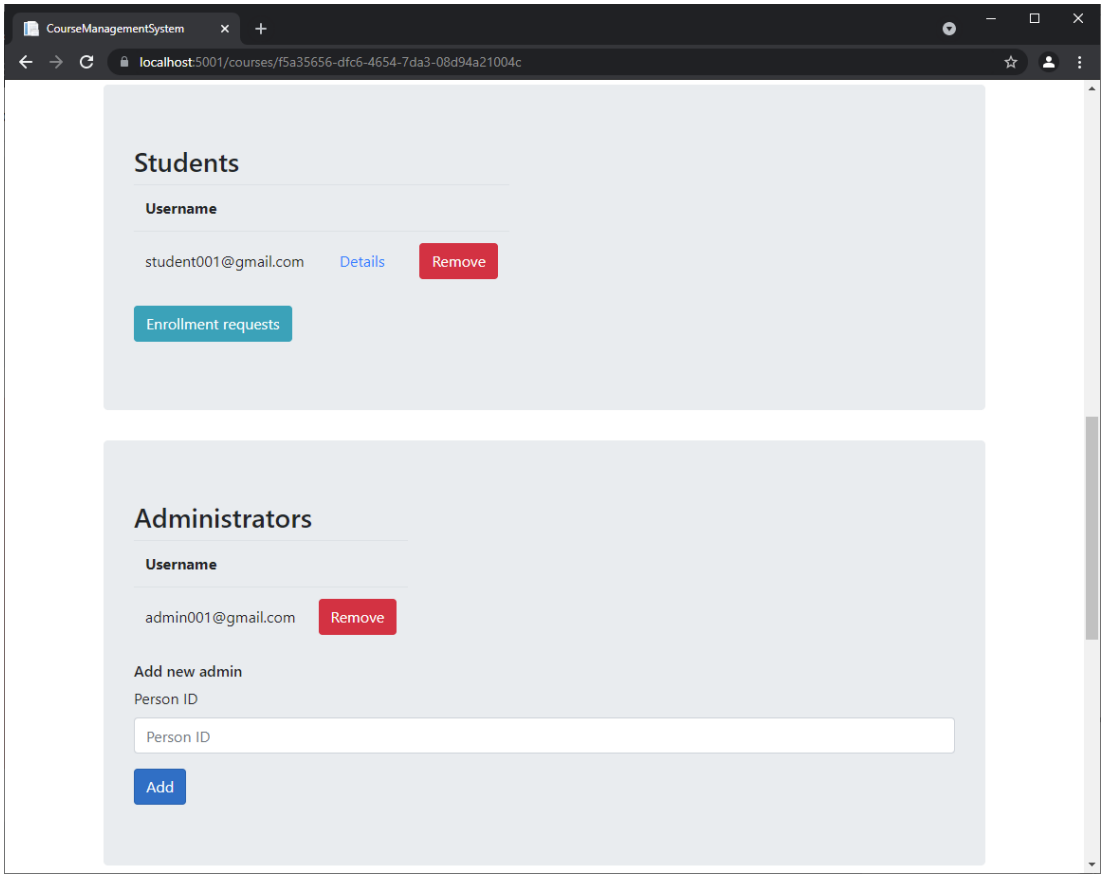
Obrázek A.5: Stránka s odevzdáním testu



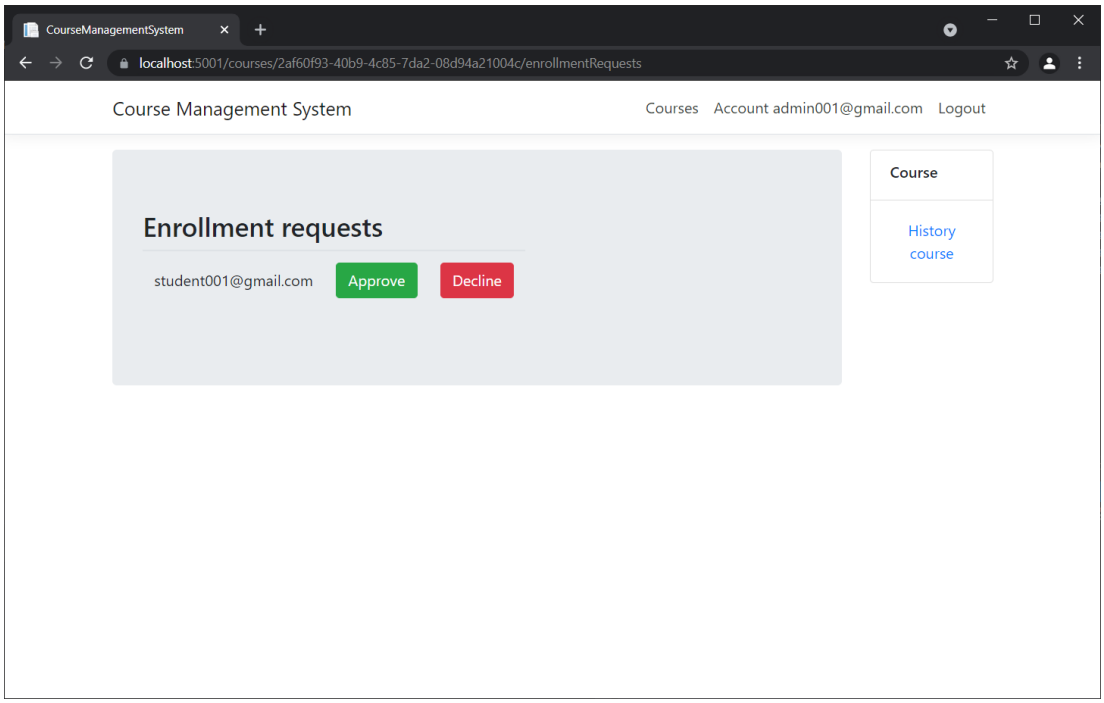
Obrázek A.6: Zobrazení vyhodnoceného testu



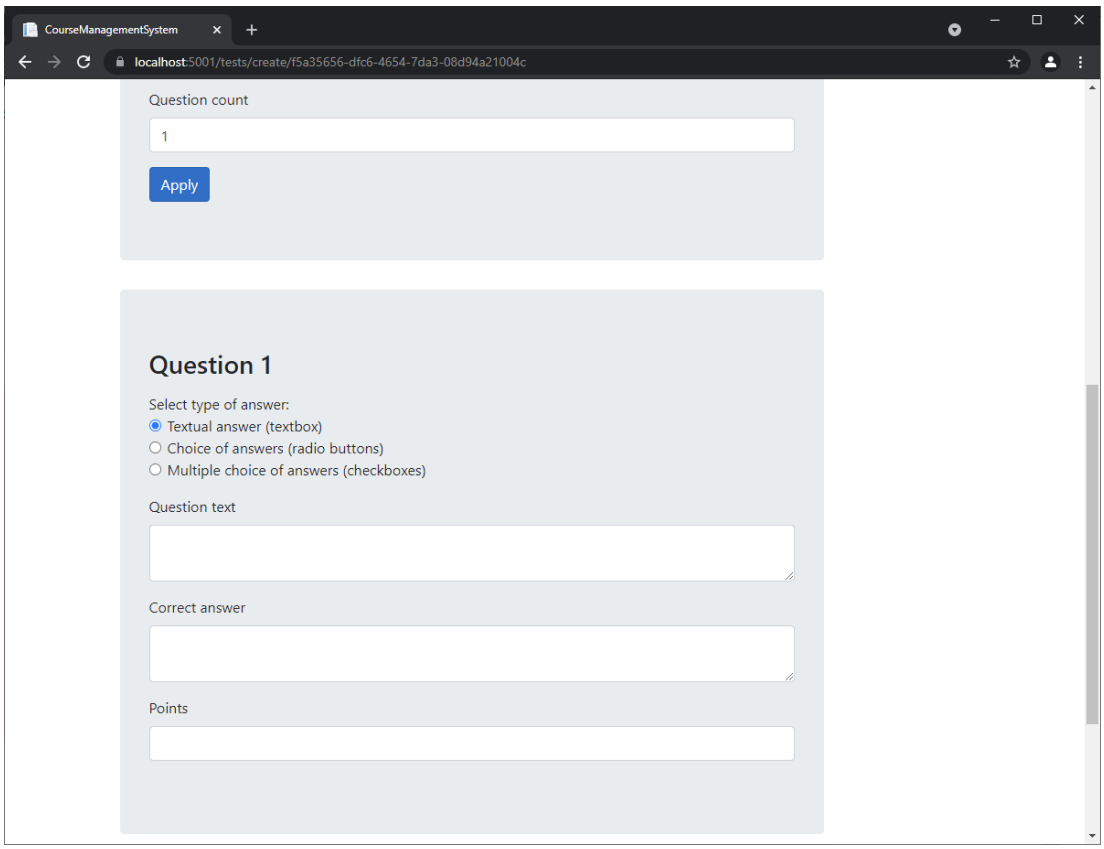
Obrázek A.7: Postranní menu



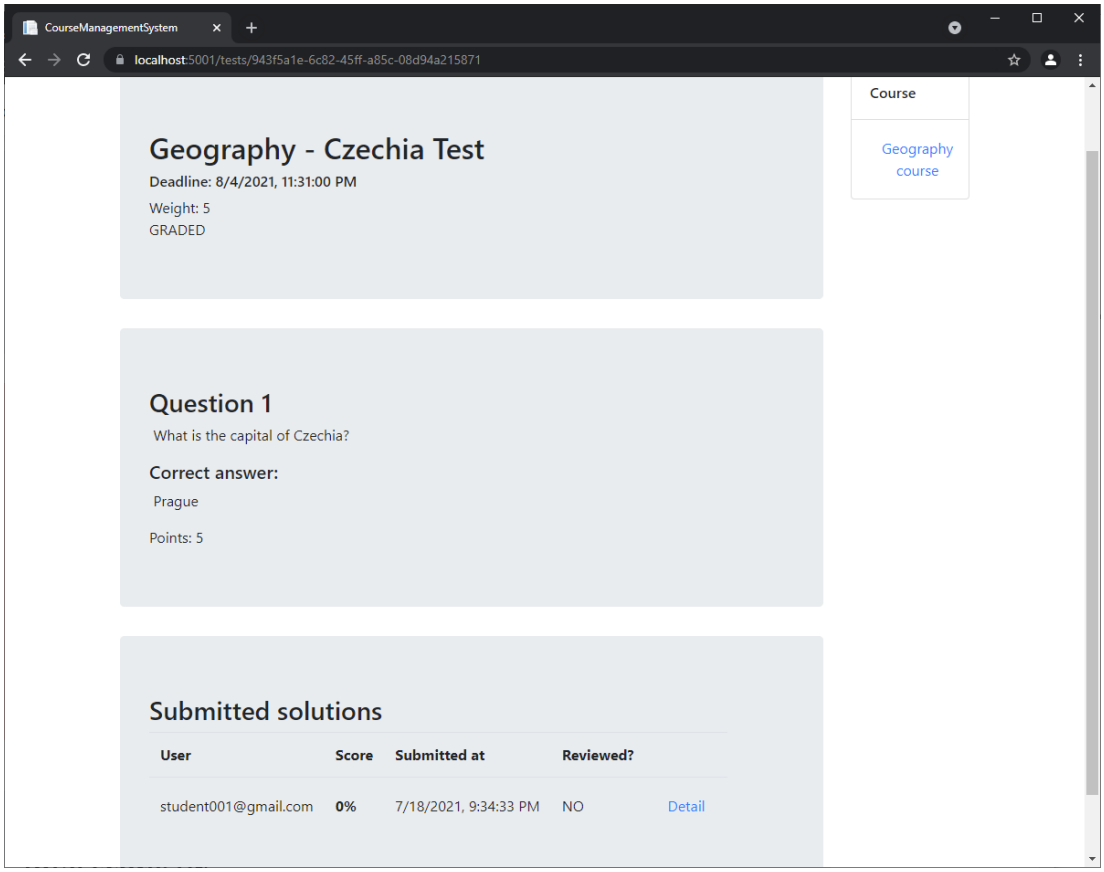
Obrázek A.8: Seznam studentů a administrátorů



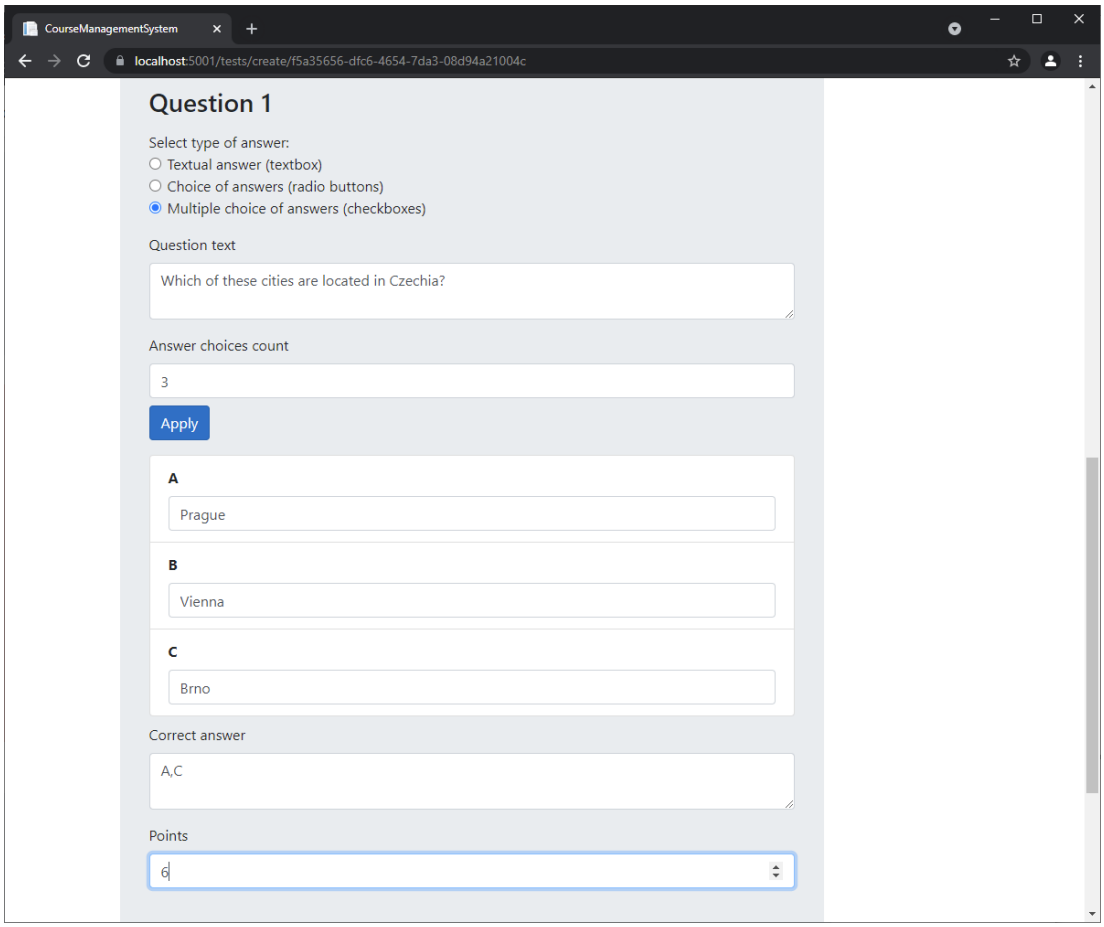
Obrázek A.9: Seznam žádostí o zápis do daného kurzu



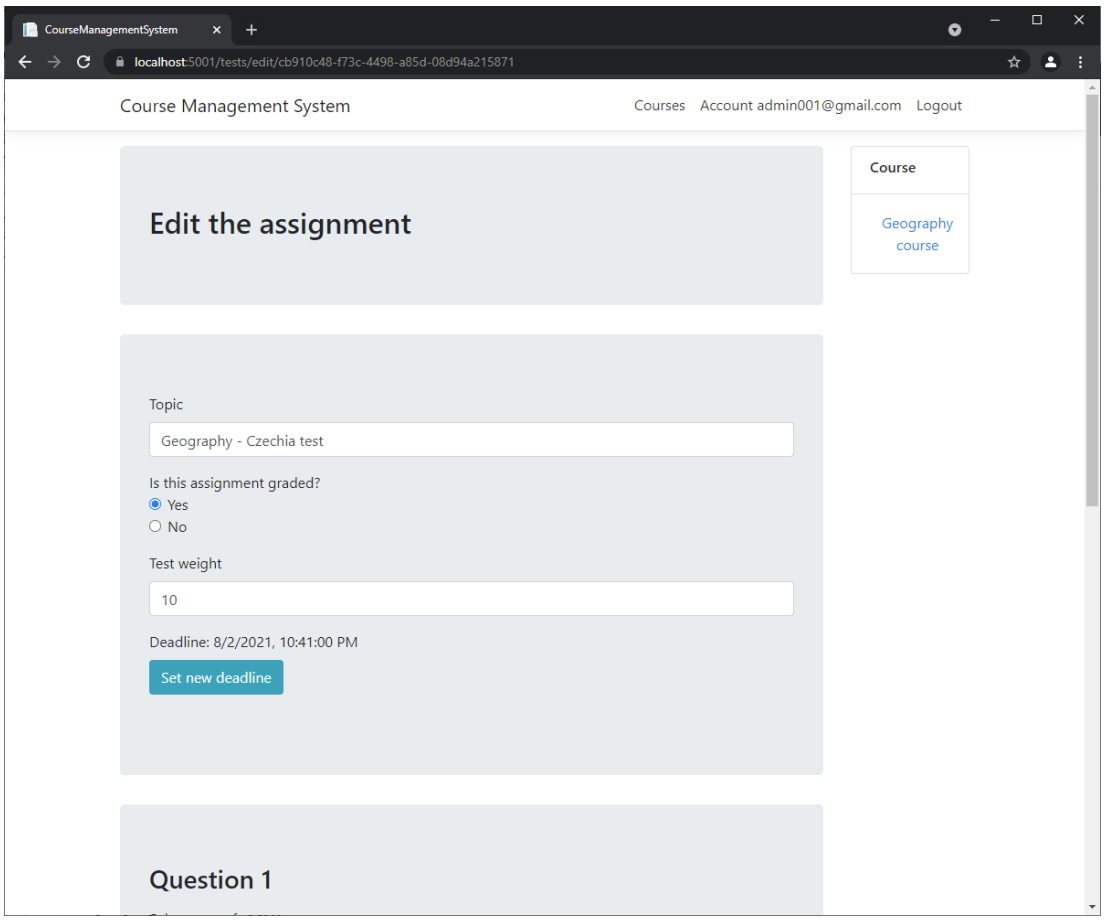
Obrázek A.10: Vytvoření testu



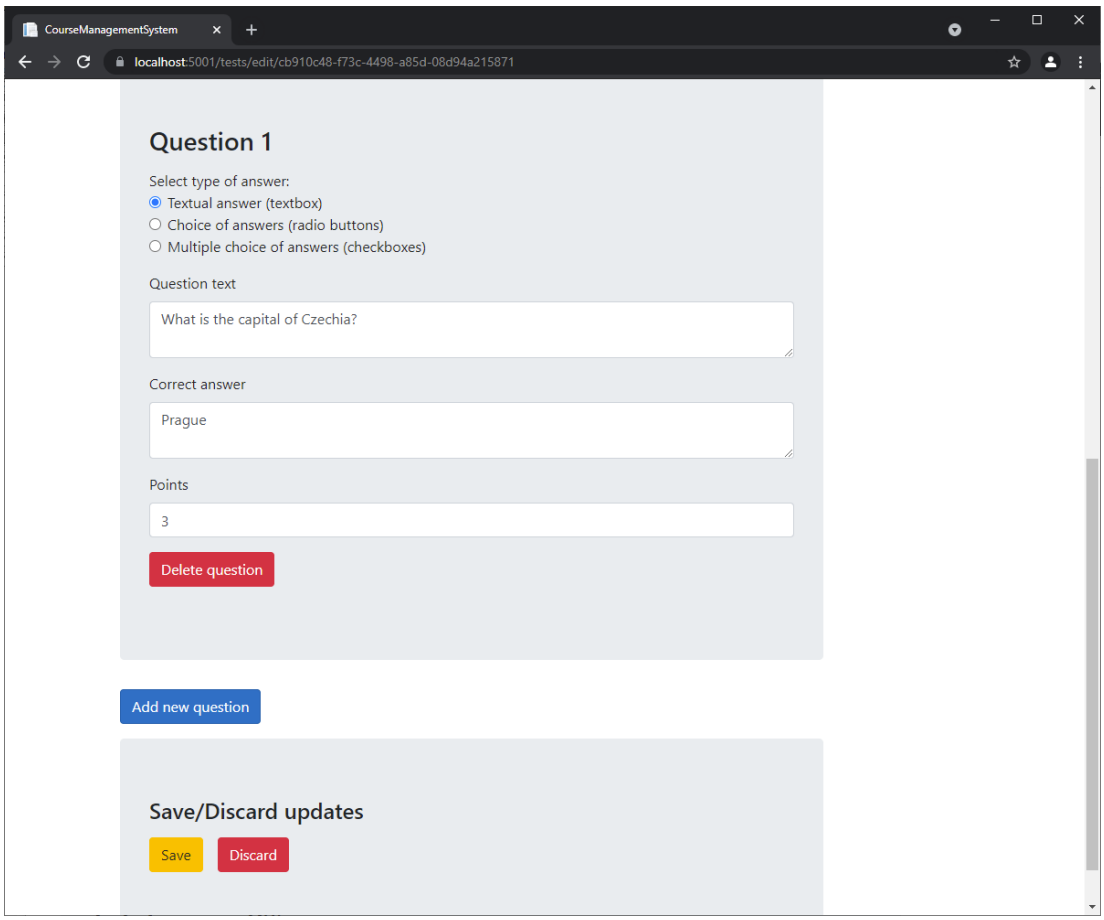
Obrázek A.11: Stránka s detaily testu



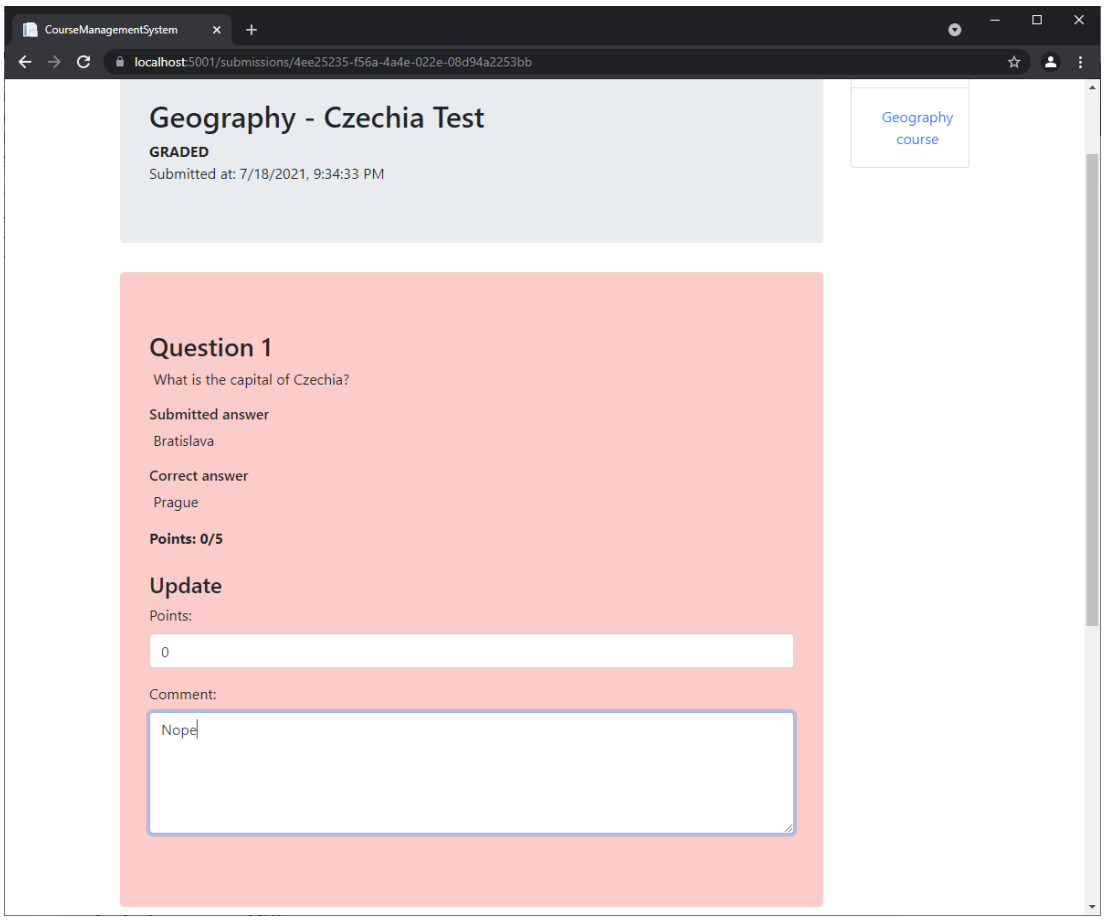
Obrázek A.12: Vzorová otázka



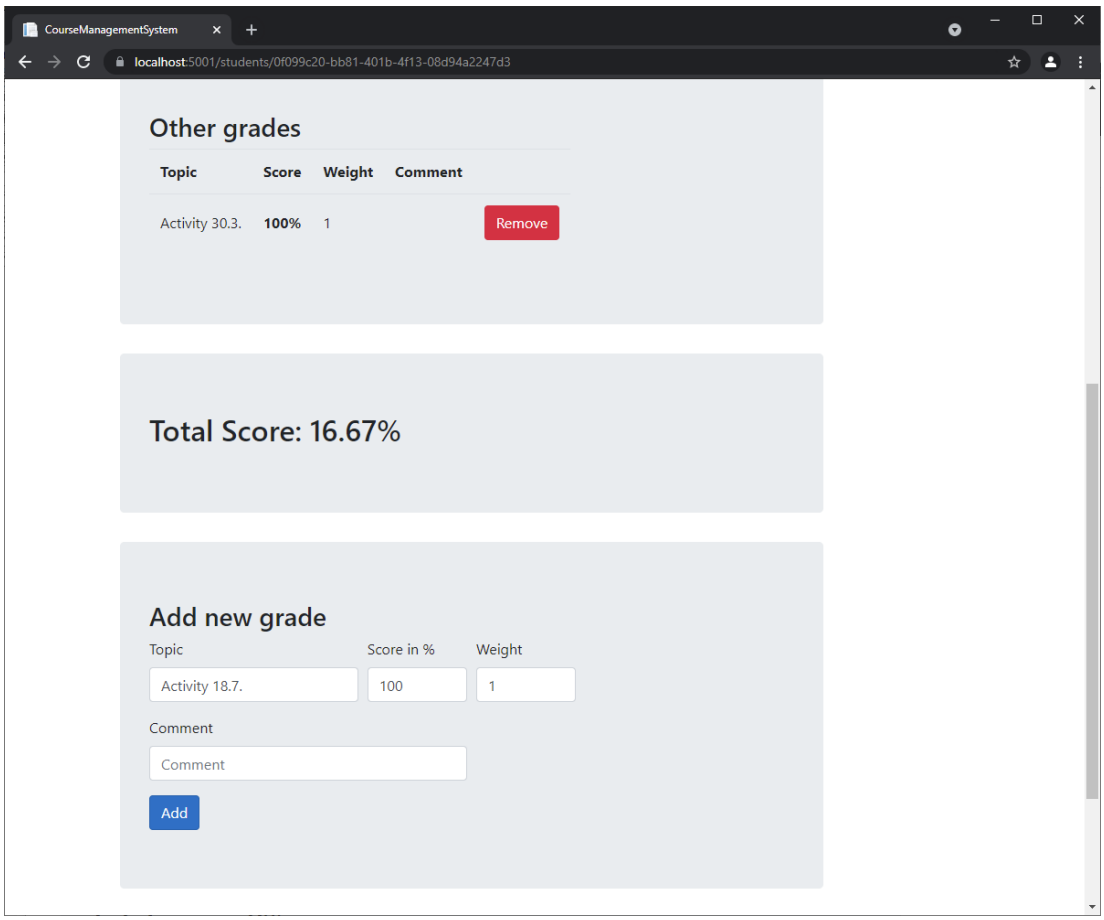
Obrázek A.13: Editace testu, část 1



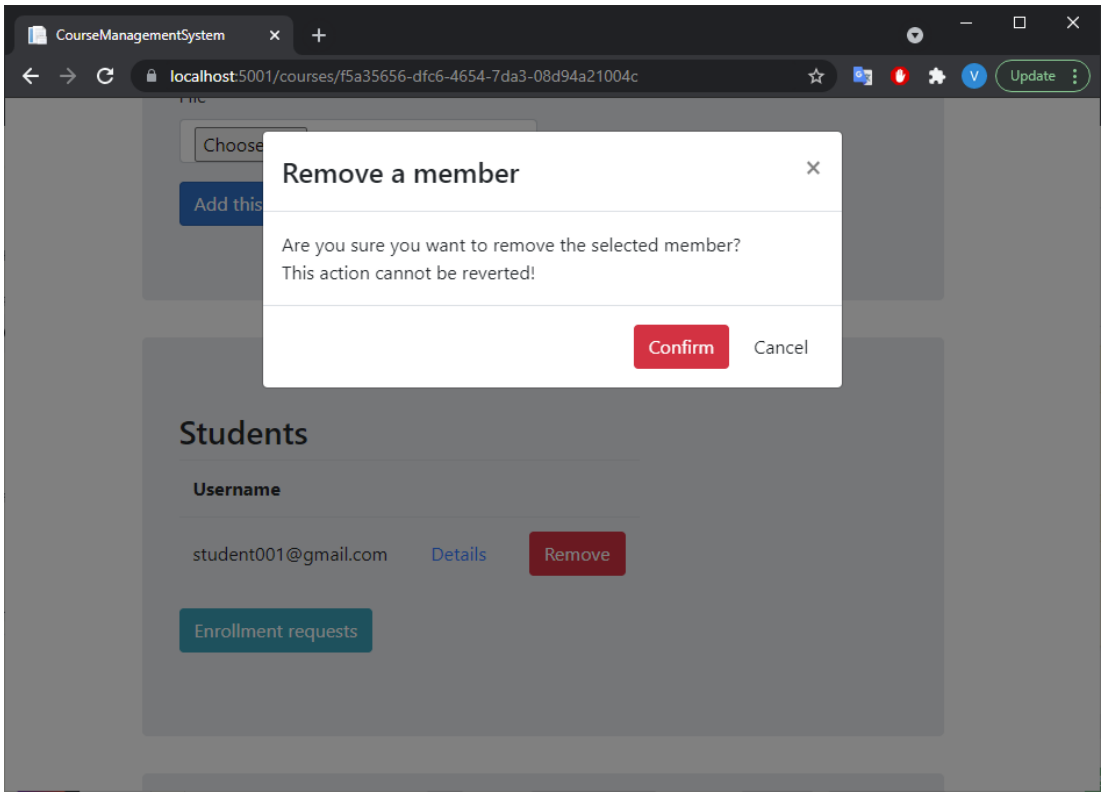
Obrázek A.14: Editace testu, část 2



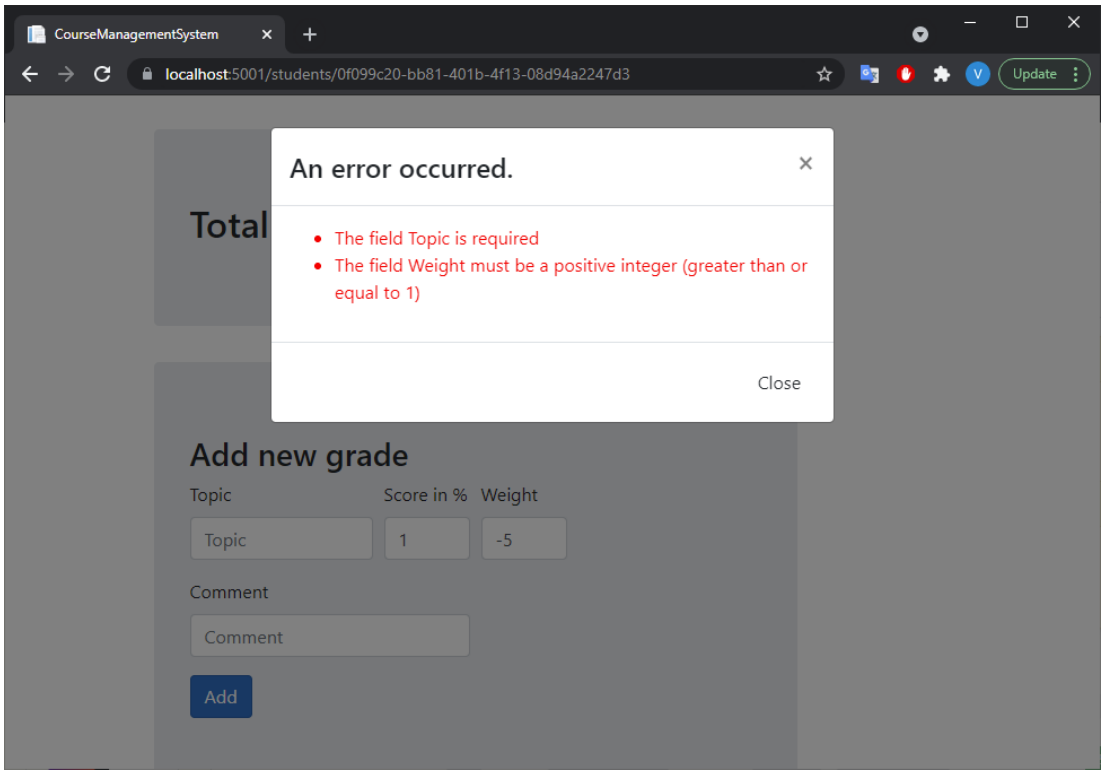
Obrázek A.15: Manuální oprava odeslaného řešení



Obrázek A.16: Formulář pro přidání známky



Obrázek A.17: Potvrzovací formulář



Obrázek A.18: Formulář s chybami