Övning 16 – API för ett Learning Management System

Intro/Teori

Vi ska bygga en egen API-backend som man kan kalla på som med Star Wars / Kortleken men vi ska även kunna göra annan CRUD funktionalitet. Vi kommer även öva på att bygga lite mer avancerad arkitektur och implementera ett *repository-pattern*.

Programmet är en påbörjan till ett Learning Management System (lärplattform). Användaren kunna göra specifika API-anrop för att få information om kurser och delar av kurser (moduler). De ska även med PUT-, PATCH-, POST- och DELETE-anrop kunna ändra göra ändringar i databasen.

Programmet kommer bestå av tre projekt i Visual Studio:

- Lms.Api
  - Asp NET Core Web API
  - Innehåller våra Controllers samt Program och Startup-klasserna.
- Lms.Core
  - Klassbibliotek
  - Innehåller klasser för entiteterna, Data Transfer Objects (DTOs) och interfaces för repositories.
- Lms.Data
  - Klassbibliotek
  - Innehåller klasser för databaskontextet, dataseedning samt repositories.

Instruktioner

Skapa alla projekt

1. Ladda ner och installera Postman (https://www.postman.com/downloads/) som används för att testa API:er under utveckling.

2. Skapa ett nytt projekt i Visual Studio och välj ASP.NET Core Web API, döp den till Lms.Api och behåll standardinställningarna.

3. Testa att köra programmet med ctrl + F5. Det som kommer upp i browsern är scaffoldad dokumentation baserat på Swagger. Bra dokumentation är nödvändigt för att folk ska kunna använda ens API.

4. Ta bort modellen och controllern för WeatherForecast.

5. I Solution Explorer, högerklicka på Solution > Add > New Project… > Class Library. Döp till ’Lms.Core’, välj .NET 5.0.

6. Skapa ett nytt Class Library projekt, döp till Lms.Data. Högerklicka på Lms.Data projektet > Manage NuGet Packages... > Browse > Sök efter och installera *Microsoft.EntityFrameworkCore, Microsoft.EntityFrameworkCore.SqlServer* och *Bogus*. Lägg till en mapp i projektet som heter Data.

7. Högerklicka på dess Dependencies > Add Project Refence… > Lms.Core > OK. Gör samma sak för med Lms.Api men ge den project reference till både Lms.Core och Lms.Data.

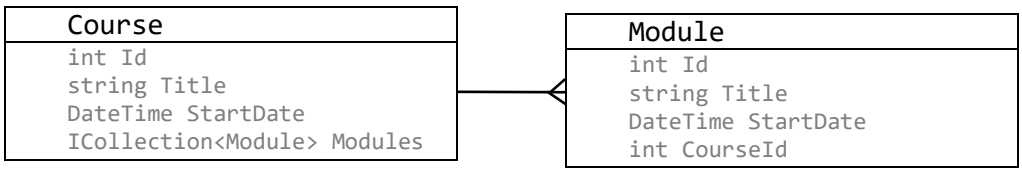
8. Lägg till NuGet-paketet *Microsoft.AspNetCore.Mvc.NewtonsoftJson* i Lms.Api. Uppdatera Startup.cs:

```
services.AddControllers(opt => opt.ReturnHttpNotAcceptable = true)
    .AddNewtonsoftJson()
    .AddXmlDataContractSerializerFormatters();
```

   Dessa tillägg hjälper oss att mappa mot Json och Xml.

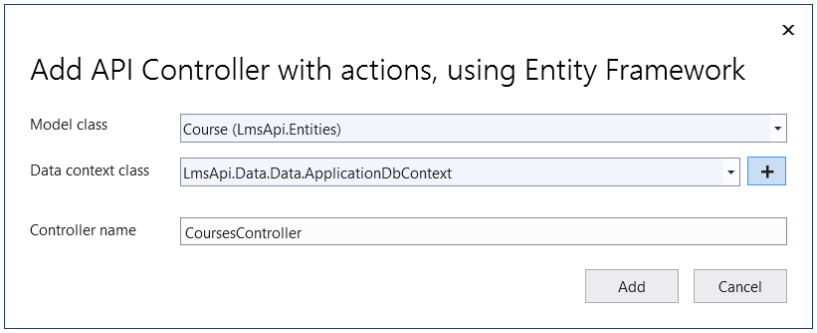
Modeller

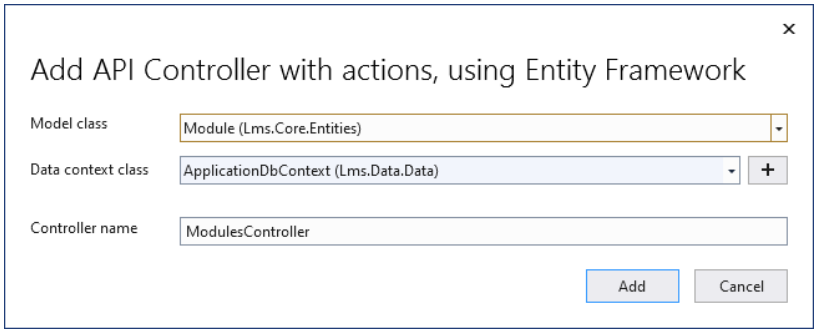
9. Lägg till en Entities-folder i Lms.Core och skapa två publika modeller (en-till-många-relation):



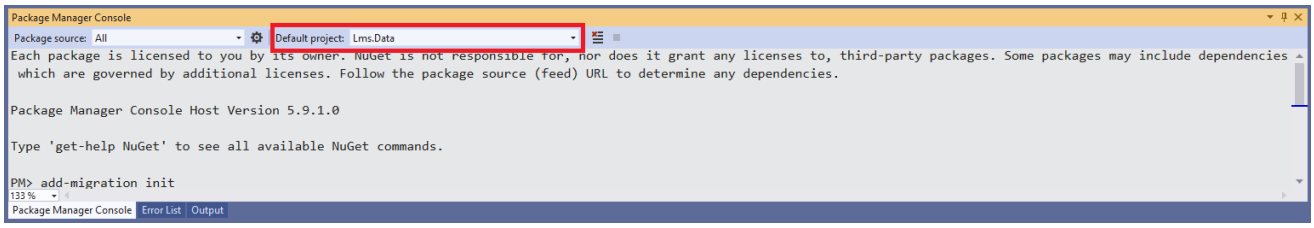
Generera kontroller och seeda data

10. Högerklicka på Lms.Api.Controllers > Add > Controller… > API > API Controller with actions, using Entity Framework > Add. Välj Course som modell och klicka på plusknappen för att skapa ett nytt DbContext. Se till att path:en pekar till Data-foldern i Lms.Data istället för den autogenererade. Skapa en till kontroller på samma sätt för Module-modellen men välj den existerande DbContext.





11. Öppna Package Manager Console, ändra Default Project till Lms.Data, kör `add-migration` och `update-database`.



Seed Data

12. Lägg till SeedData.cs i Lms.Data.Data. Skriv kod för att seeda några kurser som alla har några tillhörande moduler. Kalla på SeedData i Program.cs. Ta ett kik på Gymboknings-övningen ifall ni behöver fräscha upp minnet.

13. Ctrl + F5 för att bygga programmet. Kolla i Swagger-dokumentationen att de seedade kurserna och modulerna kommit fram.

Repositories

14. Skapa foldern Lms.Core.Repositories och lägg in ett interface, ICourseRepository. Lägg i följande metoder:

```
Task<IEnumerable<Course>> GetAllCourses();
Task<Course> GetCourse(int? Id);
Task<bool> SaveAsync();
Task AddAsync<T>(T added);
```

Skapa sedan foldern Lms.Data.Repositories och lägg i klassen CourseRepository som ärver från ICourseRepository. Ge den ett fält för DbContext och tillsätt fältet i en konstruktor. Flytta logiken för vardera metod från CoursesController till CourseRepository:

```
class CourseRepository : ICourseRepository
{
    private readonly ApplicationDbContext db;

    1 reference
    public CourseRepository(ApplicationDbContext db)
    {
        this.db = db;
    }

    0 references
    public Task<IEnumerable<Course>> GetAllCourses()
    {
        // Skriv kod här...
    }
    3 references
    public Task<Course> GetCourse(int? Id)
    {
        // Skriv kod här...
    }
    2 references
    public async Task<bool> SaveAsync()
    {
        return (await db.SaveChangesAsync()) >= 0;
    }
    1 reference
    public async Task AddAsync<T>(T added)
    {
        await db.AddAsync(added);
    }
```

15. Gör om föregående steg för Module-modellen.

Unit of Work

16. Vi vill inte att kontrollern ska kalla på repository-klasserna direkt. Det ska istället ske via en UoW-klass (*Unit of Work*). Lägg i ett interface i Lms.Core.Repositories som har båda repositories som properties och en metod för spara förändringar till databasen:

    ```
    ICourseRepository CourseRepository { get; }
    IModuleRepository ModuleRepository { get; }

    Task CompleteAsync();
    ```

    Glöm ej att göra interfacet public.

17. Lägg till en UoW-klass i Lms.Data.Repositories som implementerar IUoW. Det enda som metoden `CompleteAsync()` ska göra är att asynkront spara förändringar i databasen.

18. Lägg till ett fält för UoW-klassen i både Courses- och ModulesController. Anropa alla CRUD-metoder från repositories i dessa kontrollers med hjälp av UoW-klassen.

19. Nu när vi har all grundläggande CRUD-funktionalitet på plats så använder vi Postman för att se att allt funkar. Ni måste där ange rätt typ av anrop (GET, POST, PUT eller DELETE), rätt URI samt Body (för POST och PUT). Dubbelkolla i databasen att ändringarna gått igenom.

Data Transfer Objects och AutoMapper

20. Skapa mappen Lms.Api.Core.Dto och skapa en CourseDto som har en titel, startdatum samt slutdatum som är tre månader efter startdatum. Skapa en ModuleDto med Titel, StartDatum samt ett slutdatum som är en månad efter startdatum.

21. Installera NuGet-paketet *AutoMapper.Extensions.Microsoft.DependencyInjection* i Lms.Data och Lms.Core. Lägg till `services.AddAutoMapper(typeof(MapperProfile));` i ConfigureServices-metoden i Startup.cs. Skapa MapperProfile.cs i Lms.Data.Data:

```
public class MapperProfile : Profile
{
    0 references
    public MapperProfile()
    {
        CreateMap<Course, CourseDto>().ReverseMap();
        CreateMap<Module, ModuleDto>().ReverseMap();
    }
}
```

Kom ihåg att få in mappern som ett IMapper-objekt via kontrollernas konstruktorer:

```
private readonly ApplicationDbContext db;
private readonly IUoW uoW;
private readonly IMapper mapper;

0 references
public CoursesController(ApplicationDbContext db, IUoW uoW, IMapper mapper)
{
    this.db = db;
    this.uoW = uoW;
    this.mapper = mapper;
}
```

22. Skriv om alla CRUD-metoder i Courses- och ModulesController så att de returnerar Dtos i stället för entiteterna, samt mappar med hjälp av AutoMapper. Exempel:

```
// GET: api/Courses
[HttpGet]
0 references
public async Task<ActionResult<IEnumerable<CourseDto>>> GetAllCourses()
{
    var courses = await uoW.CourseRepository.GetAllCourses();
    var coursesDto = // Skriv kod här

    return Ok(coursesDto);
}
```

Statuskoder

Vi ska nu lägga in grundläggande errormeddelanden och statuskoder för varje metod i Courses- och ModulesController.

23. Sätt med Data Annotation-attribut en gräns på antalet bokstäver som titlarna av kurserna och modulerna får vara. Ange också att titlarna är required.

24. Skriv nu om metoderna i Courses- och ModulesController så de retunerar rätt statuskoder beroende på situation:

Situation	Respons
Ett element finns inte i databasen	`return NotFound();`
Validering misslyckas	`return BadRequest();`
Misslyckas att spara något till databasen	`return StatusCode(500);`
Allt går som det ska	`return Ok(dto);`

Patch

Den scaffoldade koden har ingen metod för att hantera PATCH-requests.

25. Installera NuGet-paketet *Microsoft.AspNetCore.JsonPatch* i Lms.Api.
26. Lägg till och avsluta följande metod i CourseController

```
[HttpPatch("{courseId}")]
0 references
public async Task<ActionResult<CourseDto>> PatchCourse(int courseId, JsonPatchDocument<CourseDto> patchDocument)
{
    // Kolla om kursen med courseId finns, returnera NotFound om den inte finns

    // Ta fram kursen mga UoW

    var model = mapper.Map<CourseDto>(course);

    patchDocument.ApplyTo(model, ModelState);

    // Försök validera modellen, returnera BadRequest om det inte går

    mapper.Map(model, course);

    if (await uoW.CourseRepository.SaveAsync())
    {
        return Ok(mapper.Map<CourseDto>(course));
    }
    else
    {
        // Returnera statuskod 500
    }
}
```

Utökad funktionalitet

27. Ge GetAllCourses() alternativet att inkludera eller inte inkludera kursernas tillhörande moduler. Detta fixas i CourseRepository och ICourseRepository genom att ge metoden en boolesk inputparameter. Beroende på om parametern är true eller false ska GetCourses() returnera två olika listor av kurser, en där moduler är inkluderade och en där de är exkluderade.

```
public async Task<IEnumerable<Course>> GetAllCourses(bool includeModules)
{

    return includeModules ? /* bara kurserna */ :
                            /* kurserna inklusive modulerna*/;
}
```

28. Skriv om GetModule() så användaren kan söka efter moduler med specifika titlar istället för id. Metoden ska då ha en sträng som inputparameter och ska endast returnera moduler med exakt den titeln.

29. Testa nu PATCH samt de utökade funktionaliteterna i Postman.

Extra

- Implementera sortering, dvs att man kan välja om GetAllCourses() och GetAllModules() ska returnera ett sorterat resultat (I queryn inte en separat endpoint).

- Implementera filtrering.

- Konsolidera alla inparametrar till ett objekt när antalet börjar växa. Kom ihåg att komplexa objekt förväntas komma från bodyn inte från query. Så här måste vi tala om för modelbindern med [FromQuery] attributet

- Implementera pagiering, dvs att det bara visas ett antal kurser när man anropar GetAllCourses() och GetAllModules().

- Möjlighet att välja pagesize.

- Se till i SeedData att Modulernas datum inte överlappar och att de hamnar innanför kursens period. Denna logik bör skrivas i separata metoder som sedan anropas i initieringsmetoden i SeedData.