

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 övrig 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 ska 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 bygga och köra programmet (Ctrl + F5). Det som kommer upp i browsern är scaffoldad dokumentation baserat på Swagger. Bra dokumentation är nödvändigt utomstående användare och utvecklare ska kunna nyttja API:et (tänk tillbaka på när ni satt med Star Wars och Deck-Of-Cards).
4. Ta bort modellen och kontrollern för WeatherForecast.
5. I Solution Explorer, högerklicka på Solution > Add > New Project... > Class Library. Döp den till 'Lms.Core', välj .NET 5.0.

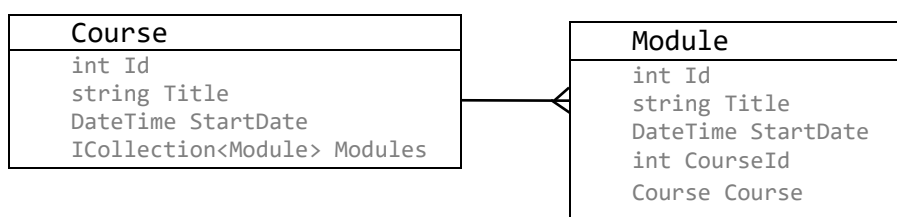
6. Skapa ett till Class Library projekt som heter **Lms.Data**. Högerklicka på Lms.Data projektet > **Manage NuGet Packages...** > Browse > Sök efter och installera **Microsoft.EntityFrameworkCore** och **Microsoft.EntityFrameworkCore.SqlServer**.
7. Högerklicka på dess **Dependencies** > Add Project Reference... > 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.

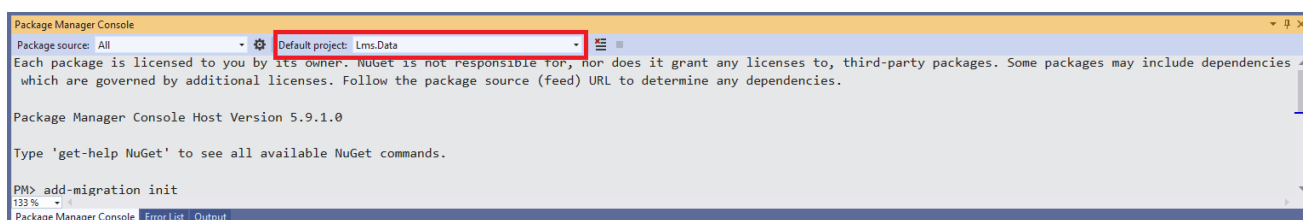
Modeller

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



Generera kontroller och databaskontext

10. Högerklicka på **Lms.Api.Controllers** > Add > Controller... > API > API Controller with actions, using Entity Framework > Add. (Ifall mappen är osynlig testa att klicka på Show All Files ) Välj **Course** som modell och klicka på **plusknappen** för att skapa ett nytt DbContext. Skapa en till kontroller på samma sätt för **Module**-modellen men välj den existerande DbContext.
11. Dra-and-droppa den genererade **Data**-foldern till Lms.Data-projektet så den kopieras där. Radera **Data**-foldern från Lms.Api. Ändra namespace i LmsApiController till Lms.Data.Data. Alla using-statements som är Lms.Api.Data ska även ändras till Lms.Data.Data.
12. Öppna Package Manager Console, ändra Default Project till Lms.Data, kör **add-migration** och **update-database**.



Seed Data

13. 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`. Kika på Gymboknings-övningen ifall ni behöver fräscha upp minnet.
14. `Ctrl + F5` för att bygga programmet. Kolla i Swagger-dokumentationen att de seedade kurserna och modulerna kommit fram genom att göra GET-requests (`Try it out` > ändra Media Type till `application/json` > `Execute`).

Repositories

15. Skapa foldern `Lms.Core.Repositories` och lägg in ett `interface`, `ICourseRepository`, med följande metoder:

```
Task<IEnumerable<Course>> GetAllCourses();
Task<Course> GetCourse(int? id);
Task<Course> FindAsync(int? id);
Task<bool> AnyAsync(int? id);
void Add(Course course);
void Update(Course course);
void Remove(Course course);
```

Skapa sedan foldern `Lms.Data.Repositories` och lägg i klassen `CourseRepository` som ärver från `ICourseRepository`. Implementera metoderna från interfacet (glöm inte lägga till en konstruktor och injicera `dbcontext`). Kolla på Booking-projektet om du fastnar.

16. Gör om föregående steg för `Module-modellen`.

Unit of Work

17. 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`.

18. Lägg till en UoW-klass i `Lms.Data.Repositories` som implementerar `IUoW`. Det enda som metoden `CompleteAsync()` gör är att asynkront spara förändringar i databasen.
19. Lägg till UoW som en service i `Startup.cs` med `services.AddScoped<IUoW, UoW>()`;

20. Lägg till ett fält för UoW-klassen i både Courses- och ModulesController. Anropa alla CRUD-metoder från repositories i dessa kontroller med hjälp av UoW-klassen istället för dbcontext.
21. 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 SQL Server Object Explorer eller med GET-anrop i Postman att ändringarna gått igenom.

Data Transfer Objects och AutoMapper

22. 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.
23. Installera NuGet-paketet AutoMapper.Extensions.Microsoft.DependencyInjection i Lms.Data och Lms.Api. Skapa MapperProfile.cs i Lms.Data.Data.
- Lägg till services.AddAutoMapper(typeof(MapperProfile)); i ConfigureServices-metoden i Startup.cs. Injicera mappern som ett IMapper-objekt via kontrollernas konstruktörer.
24. Skriv om alla CRUD-metoder i Courses- och ModulesController så att de returnerar Dto istället för entiteter, samt mappar med hjälp av AutoMapper (notera att du kan behöva modifiera befintliga Dtos eller lägga till nya). 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);
}
```

25. Testa alla metoder i Postman så de även funkar med Dtos.

Statuskoder

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

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

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

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

PATCH

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

28. Installera NuGet-paketet `Microsoft.AspNetCore.JsonPatch` i `Lms.Api`.
29. Lägg till följande `metod` i `CourseController` (repetera del 8 i *Building a RESTful API with ASP.NET Core 3* om det skulle behövas):

```
[HttpPatch("{courseId}")]
0 references
public async Task<ActionResult<CourseDto>> PatchCourse(int courseId,
    JsonPatchDocument<CourseDto> patchDocument)
{
    // Skriv kod här för PATCH, validering och statuskoder...
}
```

Skriv respektive metod i `ModulesController`.

Utökad funktionalitet

30. 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.
31. 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.
32. `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.
- Skapa ett testprojekt och skriv test för metoderna i controllers.