Övning 15 – API för turneringar och matcher

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*.

Användaren ska kunna göra specifika API-anrop för att få information om turneringar och matcher inom varje turnering. 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:

- TournamentAPI.Api
  - Asp NET Core Web API
  - Innehåller våra Controllers samt Program och Startup-klasserna.
- TournamentAPI.Core
  - Klassbibliotek
  - Innehåller klasser för entiteterna, Data Transfer Objects (DTOs) och interfaces för repositories.
- TournamentAPI.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 TournamentAPI.Api, välj .NET 8.0 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 för att 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 controllern för WeatherForecast.

5. I Solution Explorer, högerklicka på Solution > Add > New Project… > Class Library. Döp den till TournamentAPI.Core', välj .NET 8.0.

6. Skapa ett till Class Library projekt som heter TournamentAPI.Data. Högerklicka på TournamentAPI.Data projektet > Manage NuGet Packages... > Browse > Sök efter och installera *Microsoft.EntityFrameworkCore och Microsoft.EntityFrameworkCore.SqlServer.*

7. I Solution Explorer, under TournamentAPI.Data högerklicka på Dependencies > Add Project Refence… > TournamentAPI.Core > OK. Gör samma sak för med TournamentAPI.Api men ge denproject reference till både TournamentAPI.Core och TournamentAPI.Data.

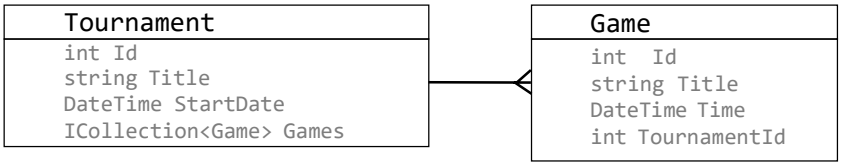
8. Lägg till NuGet-paketet *Microsoft.AspNetCore.Mvc.NewtonsoftJson* i Tournament.Api. Uppdatera Program.cs:

```
builder.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 TournamentAPI.Core och skapa två publika modeller (en-till-många-relation):

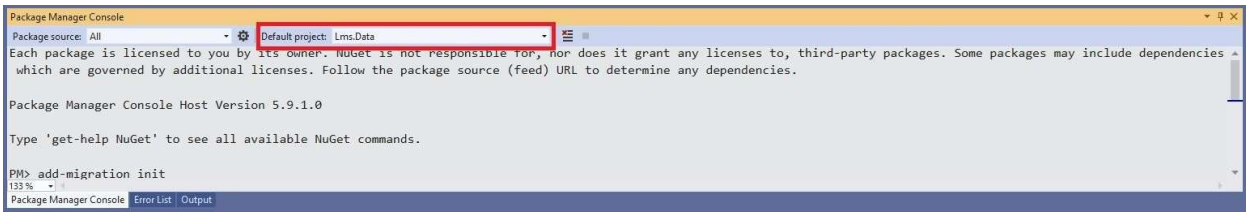


Generera kontroller och databaskontext

10. Högerklicka på TournamentAPI.Controllers > Add > Controller… > API > API Controller with actions, using Entity Framework > Add. (Ifall Controllers-mappen är osynlig testa att klicka på Show All Files) Välj Tournament som modell och klicka på plusknappen för att skapa ett nytt DbContext. Skapa en till kontroller på samma sätt för Game-modellen men välj den existerande DbContext.

11. Dra-and-droppa den genererade Data-foldern till TournamentAPI.Data-projektet så denkopieras där. Radera Data-foldern från TournamentAPI.Api. Ändra namespace i TournamentApiContext till TournamentAPI.Data.Data. Alla using-statements som är TournamentAPI.Api.Data ska även ändras till TournamentAPI.Data.Data.

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



Seed Data

13. Skapa mappen Extensions i TournamentAPI.Api. Lägg till den statiska klassen ApplicationBuilderExtensions.cs med den statiska metoden SeedDataAsync(). Kika på University-projektet i GitHub ifall ni behöver fräscha upp minnet.

14. Lägg till SeedData.cs i TournamentAPI.Data.Data. Skriv kod för att seeda några turneringarsom alla har några tillhörande matcher. Kalla på SeedData i Program.cs efter att applikationen byggts.

```
var app = builder.Build();

await app.SeedDataAsync();
```

15. Ctrl + F5 för att bygga programmet. Kolla i Swagger-dokumentationen att de seedade turneringarna och matcherna kommit fram genom att göra GET-requests (Try it out > ändra Media Type till application/json > Execute).

16. Nu när vi har all grundläggande CRUD-funktionalitet och dummy-data 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). Kom ihåg att PUT kräver att man anger en hel Body, även om man bara ska ändra ett fält. Dubbelkolla i SQL Server Object Explorer eller med GET-anrop i Postman att ändringarna gått igenom.

Repositories

17. Skapa foldern TournamentAPI.Core.Repositories och lägg in ett interface,ITournamentRepository, med följande metoder:

```
Task<IEnumerable<Tournament>> GetAllAsync();
Task<Tournament> GetAsync(int id);
Task<bool> AnyAsync(int id);
void Add(Tournament tournament);
void Update(Tournament tournament);
void Remove(Tournament tournament);
```

    Skapa sedan foldern TournamentAPI.Data.Repositories och lägg i klassen TournamentRepository som ärver från ITournamentRepository. Implementera metoderna från interfacet (glöm inte lägga till en konstruktor och injicera dbcontext).

18. Gör om föregående steg för Game-modellen.

Unit of Work

19. 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 TournamentAPI.Core.Repositories som har båda repositories som properties och en metod för att spara förändringar till databasen:

```
ITournamentRepository TournamentRepository { get; }
IGameRepository GameRepository { get; }

Task CompleteAsync();
```

    Glöm ej att göra interfacet publikt.

20. Lägg till en UoW-klass i TournamentAPI.Data.Repositories som implementerar IUoW. Klassenkommer även behöva få dbcontext injicerat i sin konstruktor så den kan skicka det vidaretill instanserna av TournamentRepository och GameRepository, samt använda den i CompleteAsync. Det enda som metoden CompleteAsync() gör är att asynkront spara förändringar i databasen.

21. Lägg till UoW som en service i Program.cs med `builder.Services.AddScoped<IUoW, UoW>();`

22. Lägg till ett fält för UoW-klassen i både Tournaments- och GamesController. Anropa alla CRUD-metoder från repositories i dessa kontrollers med hjälp av UoW-klassen istället för dbcontext.

23. 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

24. Skapa mappen TournamentAPI.Core.Dto och skapa en TournamentDto som har en titel,startdatum samt slutdatum som är tre månader efter startdatum (sök i Microsofts dokumentation efter en passande metod för att lägga till en månad på en befintlig DateTime-variabel). Skapa en GameDto med titel och startdatum.

25. Installera NuGet-paketet *AutoMapper.Extensions.Microsoft.DependencyInjection* i TournamentAPI.Data och TournamentAPI.Api.

    Skapa klassen TournamentMappings.cs i Tournament.Data.Data, låt den ärva från `Profile`, lägg till en mappning från båda entiteterna till deras Dto.

    Lägg till `builder.services.AddAutoMapper(typeof(TournamentMappings));` bland de andra services i Program.cs. Injicera mappern som ett IMapper-objekt via kontrollernas konstruktorer.

26. Skriv om alla CRUD-metoder i Tournaments- och GamesController 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/Tournaments
[HttpGet]
0 references
public async Task<ActionResult<IEnumerable<TournamentDto>>> GetTournament()
{
    var tournamentDto = mapper.Map//skriv din kod här...
    return Ok(tournamentDto);
}
```

27. 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 Tournaments- och GamesController.

28. Sätt med Data Annotation-attribut en gräns på antalet bokstäver som titlarna av turneringarna och matcherna får vara. Ange också att titlarna är required.

29. Skriv om metoderna i Tournaments- och GamesController 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.

30. Lägg till följande metod i TournamentsController:

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

Skriv respektive metod i GamesController.

Utökad funktionalitet

31. Ge GetTournament() alternativet att inkludera eller inte inkludera tillhörande matcher med hjälp av en ny input-parameter. Beroende på om parametern är true eller false ska GetTournament() returnera två olika listor av turneringar, en där matcher är inkluderade och en där de är exkluderade.

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

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

Extra

- Implementera sortering, dvs att man kan välja om GetTournament() och GetGame() 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 turneringar när man anropar GetTournament() och GetGame().

- Möjlighet att välja pagesize.

- Se till i SeedData att matchernas datum inte överlappar och att de hamnar innanför turneringens 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.