Web API övningsuppgifter, vidareutveckling av RövaraspråksAPI:t

I dem här övningsuppgifterna får du arbeta med att vidareutveckla API:t som du började på i tidigare övningsuppgifter: RövarspråksAPI.

Vi testar att lägga till fler endpoints, övar på att skicka med query parametrar, med mera. Vi kommer också att se till att våra endpoints är asynkrona (se till att vi inte ”låser ner” någon del av vår applikation för att vänta på att t.ex. ett databasanrop ska returnera ett resultat).

(Om du vill läsa om asynkron programmering med async await i .NET finns bland annat denna sida: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/async/task-asynchronous-programming-model).

Uppgift 1

I ett API har man ofta flera endpoints som används för att göra olika saker. Ofta vill man arbeta med data på ett mer varaktigt sätt än att man bara ”kastar bort” datan efter att en API-request har hanterats. Därför kommer du nu skapa en enklare databas med EF Core som lagrar Rövarspråks-översättningarna.

Öppna din existerande solution ”RobberLanguageAPI” i Visual Studio.

1. Öppna modellklassen som du skapade senast ”Translation” och lägg till följande properties: Id, CreationDate, ModificationDate.
2. Skapa en ny mapp **”Data”** i ditt projekt där du sedan skapar en ny klass **”RobberTranslationDBContext”**.
3. Lägg till den grundläggande kod som behövs i DB-context klassen för att skapa upp databasen.
4. Du behöver även lägga till följande registrering av databascontexten i Startup.cs i metoden **”ConfigureServices”** (lägger till databascontexten till DI-containern):

   ```
   services.AddDbContext<RobberTranslationDBContext>(options =>
   options.UseSqlServer(@"Server=(localdb)\mssqllocaldb;Database=Translations"));
   ```

5. Gör en databasmigration och uppdatera databasen.

Uppgift 2

Du har nu en modifierad modellklass och en databas som återspeglar detta. Nu ska du modifiera din tidigare skapade POST-endpoint. I en POST-endpoint skickas datavärden in som del av requestens body. Id behöver man inte ange något värde för eftersom det automatiskt genereras av databasen.

1. För att kunna anropa databasen direkt i vår controller lägger vi till en instans av vår databascontext i controllern, enligt nedan. Observera att det inte är helt idealt att instansiera databascontext-klassen på detta sätt men det duger för den här övningen.

```
public RobberLanguageController(RobberTranslationDBContext context)
{
        _context = context;
}
```

2. Modifiera den tidigare skrivna POST-endpoint så att den sparar ned responsen som man får genom att skicka en request till den (sparar ned översättningen till databasen). I samband med detta vill vi även ändra så att våra endpoints använder sig av **async await** och har returtypen **Task<TResult>** istället.

3. Du kan läsa mer om IActionResult och asynchronous actions på denna sida:

   https://docs.microsoft.com/en-us/aspnet/core/web-api/action-return-types?view=aspnetcore-5.0

   Mer om asynkron programmering med async och await:

   https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/async/

   Du ska göra om din endpoint enligt mönstret för hur koden är skriven i exemplet under rubriken **”Asynchronous action”** (se den första länken här ovanför):

4. Kontrollera i endpointen om inparametern **originalsentence** är ”nullorwhitespace”, i så fall – returnera **BadRequest()**; Det gör vi eftersom att vi inte vill att man ska kunna skapa ett tomt Translation-objekt och spara ned till databasen.

5. Skapa ett Translation-objekt (om indata inte är tom) och lägg till det till databasen. Notera att du måste **”awaita”** operationen att lägga till en ny post i databasen eftersom endpointen nu är asynkron. Du måste även göra ytterligare ett anrop (också med await): **savechangesasync** för att spara ned den nya översättningen till databasen.

   - Om det gick att lägga till (POST-operationen gick bra), returnera då ett så kallat **CreatedAtActionResult**-objekt.

   OBS! Notera att vi nu har bytt HTTP-statuskod som man får tillbaka ifall POST-requesten var lyckad. Det är mer lämpligt att använda statuskod 201 eftersom den är mer specifik och betyder ”Created”.

6. Dekorerar endpointen med följande (förutom rätt HTTP attribut som visar att det är en POST-endpoint, för att visa vilken typ av HTTP-response endpointen kan returnera:

```
[ProducesResponseType(StatusCodes.Status201Created)][ProducesResponseType(S
tatusCodes.Status400BadRequest)]
```

Uppgift 3

Vår modifierade POST-endpoint kommer dock inte fungera förrän vi har skapat en till endpoint, närmare bestämt en GET-endpoint som tar fram ett Translation-objekt baserat på Id. Det eftersom att vi returnerar vårt precis skapade objekt i POST-endpointen genom att anropa GET-endpointen.

1. Skapa en ny endpoint i din TranslationController, som tar fram en Translation utifrån inskickat id (inparameter id). På samma sätt som POST-endpointen ska den vara asynkron (använda sig av async och await) och returnera en Task<ActionResult<Translation>.
2. Dekorera endpointen med rätt HTTP Attribute som visar vad det är för typ av endpoint (den typen som används för att hämta ett objekt med inskickat Id samt dekorera endpointen med följande för att visa vilken typ av HTTP-response som endpointen kan returnera:

   ```
   [ProducesResponseType(StatusCodes.Status200OK)][ProducesResponseType(StatusCodes.
   Status404NotFound)]
   ```

3. Nu kan du testa att anropa din modifierade POST-endpoint via Swagger. När du har ”Postat” några translations kan du också prova din GET-BY-ID-endpoint och se så att den fungerar som väntat (returnerar translation med Id:et som du skickar med requesten).

Uppgift 4

I ett REST-API brukar man oftast även ha en endpoint som returnerar en lista som innehåller alla objekt av en typ: en GET-endpoint utan parametrar.

1. Skriv en GET-endpoint som hämtar alla existerande Translation-objekt och returnerar de som en lista (IEnumerable). Om det inte finns några objekt så returneras en tom lista (tom IEnumerable). Använd dig av metoden ToListAsync, eftersom endpointen ska vara asynkron (tillämpa async och await). Returtyp för metoden:

   ```
   Task<ActionResult<IEnumerable<Translation>>>
   ```

2. Dekorera endpointen med rätt HTTP Attribute som visar vad det är för typ av endpoint (den typen som används för att hämta alla objekt av en viss typ samt dekorera endpointen med ProduceResponseType-attributet för att visa vilken typer av HTTP-response som endpointen kan returnera.

Uppgift 5

I dem förra Rövarspråks-API övningarna du gjorde fick du prova på att arbeta lite med Attribute Routing. Du ska nu skriva en GET-endpoint där du använder dig av det som kallas för query string parameters och attributet FromQuery. Med query string parameters kan vi som en del av URL:en för endpoints filtrera vad som ska returneras som svar på våra request.

1. Skapa en GET-endpoint som söker fram alla Translation-objekt där originalsentence innehåller en viss sträng (inparameter till endpointen blir då string keyword), och returnerar

alla matchningar som en lista. Endpointen ska liksom övriga endpoints använda sig av async och await. Returtyp för metoden:

```
Task<ActionResult<IEnumerable<Translation>>>
```

2. För att vi ska kunna använda oss av en **request query string parameter** i vår endpoint behöver vi använda attributet **FromQuery**. Läs om hur man använder det på den här sidan: https://docs.microsoft.com/en-us/aspnet/core/web-api/?view=aspnetcore-5.0#binding-source-parameter-inference och se till att du i din endpoint använder FromQuery-attributet.

Om det inte finns några matchningar i databasen ska en tom lista returneras.

3. Dekorera endpointen med rätt **HTTP Attribute** som visar vad det är för typ av endpoint samt dekorera endpointen med attributet som visar vilken typ av HTTP-response som endpointen kan returnera.

4. **Bonusuppgift:** Det finns olika sätt att hantera t.ex. om användaren skickar in ett tomt värde (inte skickar med något värde på keyword i det här fallet), man kan välja att hantera det som ett fel och retunera BadRequest, eller så kan man välja att hantera det som en ”tom sökning” och returnera NotFound istället. Det är även möjligt att inte lägga till någon ”specialregel” utan bara returnera den tomma listan.

- Fundera över vilket sätt du tycker känns mest rimligt i det här fallet och välj att utforma din endpoint i enlighet med det. Varför väljer du just det sättet? Förklara.

5. **Bonusuppgift:** Beroende på vilket sätt du löste det på i steg 4, fyll vid behov på med ytterligare attribut ovanför endpointen som visar vilka fler ResponseTypes som endpointen har.

Uppgift 6

Det finns två ytterligare vanliga typer av endpoints som ingår i det som brukar kallas för CRUD (Create, read, update, delete): update och delete.

1. Skapa en endpoint som låter den som anropar endpointen skicka in uppdaterade värden för samtliga properties (dock ej Id). Samma regler som tidigare: använd **async-await**. HTTP-attributet som används för att markera att en endpoint hanterar uppdaterings-requests är PUT. För instruktioner om hur man gör för att skapa en PUT-endpoint kan du läsa t.ex. här: https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-web-api?view=aspnetcore-5.0&tabs=visual-studio
2. Tänk på att du behöver hantera möjliga fel som kan uppstå (se t.ex. ovan länkade tutorial), t.ex. om det inte finns något Translation-objekt med angivet Id i databasen.
3. Dekorera endpointen med rätt **HTTP Attribute** och med dem HTTP-response-attribut som visar vilka möjliga ResponseT ypes som endpointen har.

Uppgift 7

1. Skapa en endpoint som låter en ta bort ett givet Translation-objekt ur databasen. Om du behöver kolla upp hur man skapar en så kallad DELETE-endpoint i .Net Core Web API kan du t.ex. även här läsa i Microsofts tutorial (se länken i uppgift 6).
2. Tänk på att du behöver hantera ifall det inte finns något Translation-objekt med angivet Id i databasen.
3. Dekorera endpointen med rätt HTTP Attribute och med dem HTTP-responseType-attribut som visar vilka möjliga ResponseT ypes som endpointen har.