

## Del1: En Webservice med Web Api och REST

### Introduktion

I detta projekt ska vi bygga en Web Service där vi genom ett API (application programming interface) hämta data samt alternera den. Temat på uppgiften är filmer och tanken är att det är en web service där man kan gå in och se information om filmer som ligger lagrade i databasen. På denna tutorial kommer det sedan en uppgift som är den obligatoriska uppgiften för veckan där ni ska utöka tjänsten samt om ni vill, även koppla på andra publika tjänster.

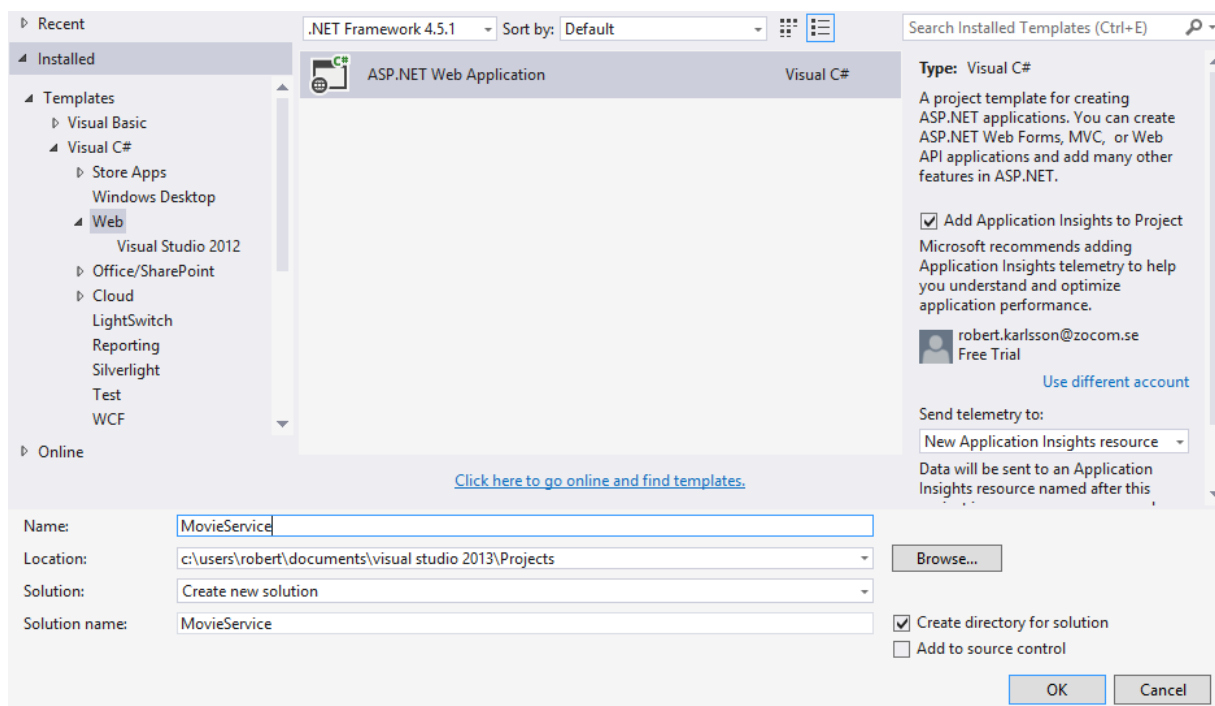
All kreativitet uppmuntras!

Denna del hanterar Backenddelen av vår webservice. Nästa del kommer att hantera Frontenddelen av vår webservice.





### Skapa projektet



Öppna Visual Studio och skapa ett nytt projekt av typen ASP.NET Web Application. Döp din applikation till MovieService och i nästa steg väljer du "Web API".

ASP.NET Web API är ett väldigt modernt ramverk som gör det lätt att bygga http services som blir tillgängliga för många olika typer av klienter. Web API är en perfekt plattform om vi vill bygga RESTful-applikationer vilken är anledningen till att vi använder det till denna applikation. Ett annat sätt att bygga dessa typer av tjänster på är med WCF men det ska vi titta mer på nästa vecka!



Select a template:

 Empty
  Web Forms
  MVC
  Web API


 Single Page Application
  Azure Mobile Service


A project template for creating RESTful HTTP services that can reach a broad range of clients including browsers and mobile devices.


[Learn more](#)

[Change Authentication](#)

Authentication: **Individual User Accounts**

 **Microsoft Azure**

 ☐ Host in the cloud

Website 

Signed in as robert.karlsson@zocom.se  
[Manage Subscriptions](#)

Add folders and core references for:

☐ Web Forms ☒ MVC ☒ Web API

☐ Add unit tests

Test project name:

OK Cancel

## Lägg till Models

I detta steg kommer vi att lägga till model-klasser vars uppgift är att definiera databasentiteterna. Vi kommer sedan att lägga till WEB API controllers som gör CRUD-operationerna på dessa entiteter.

Databasen kommer att skapas utefter Entity Frameworks "Code First" vilket betyder att vi skriver klasser som motsvarar databastabeller som sedan skapas av EF som en databas. För denna uppgift kommer vi att ha två stycken domain objects som ska skapas, Movie och Director.

I din solution explorer, högerklicka på models och lägg sedan till en ny klass "Director".

I klassen lägger vi till namespace:

```
using System.ComponentModel.DataAnnotations;
```

För att kunna använda kommentarer/noter om vad som krävs när en ny entitet skapas.

Vi lägger till ett Id och ett namn i klassen Director som ska representera en verklig regissör, t.ex. Christopher Nolan.

```
public class Director
{
    public int Id { get; set; }
    [Required]
    public string Name { get; set; }
}
```

Gör en till klass som heter Movie, även den i models.

Movie ska ha lite mer innehåll.

Ett unikt Id – En unik nyckel som skapas automatiskt vid inmatning av ny data för att skilja filmerna åt

En Titel – filmens namn

Ett År – det år den släpptes på bio

Ett Pris – vad filmen kostade att göra

Ett DirectorId – En främmande nyckel som refererar till en directors id

En Director – Som håller Director-objektet

```
public class Movie
{
    public int Id { get; set; }
    [Required]
    public string Title { get; set; }
    public int Year { get; set; }
    public decimal Price { get; set; }
    public string Genre { get; set; }

    // Foreign Key
    public int DirectorId { get; set; }
    // Navigation property
    public Director Director { get; set; }
}
```

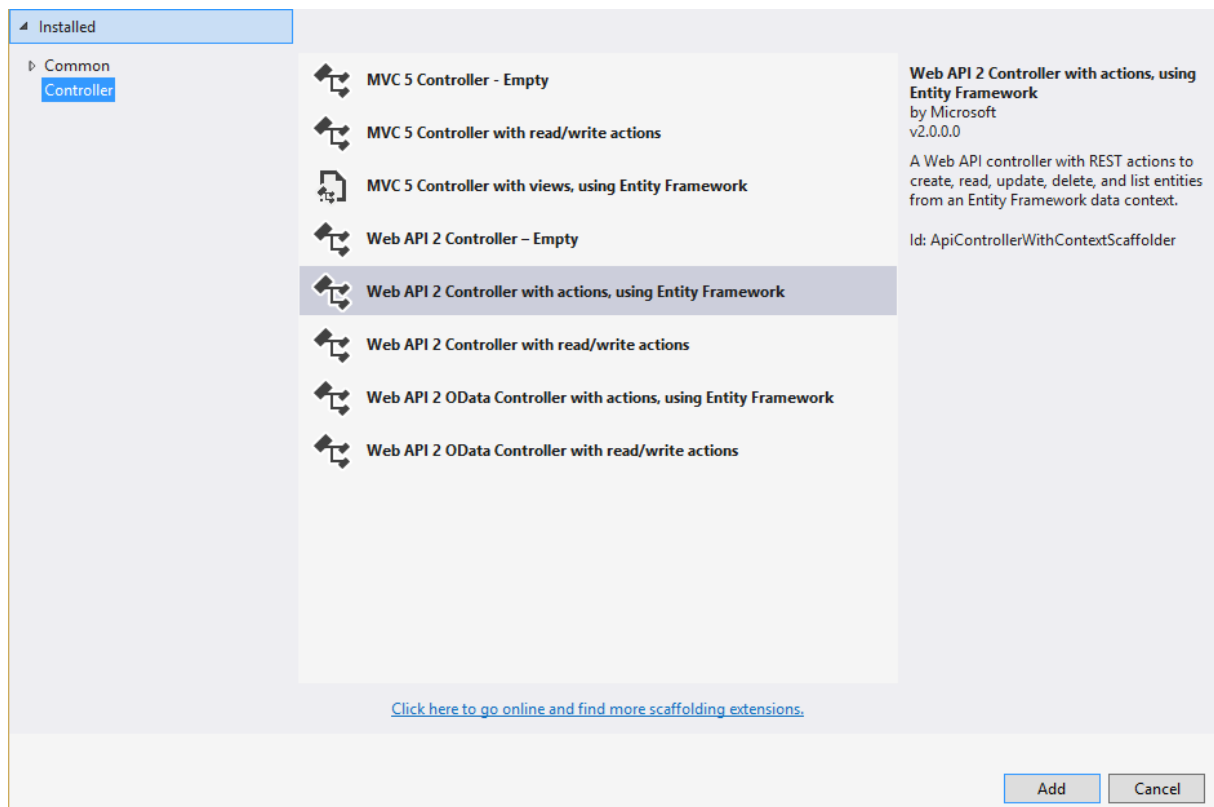
De två modellerna vi skapat kommer Entity Framework att använda för att skapa databastabellerna där Id är den primära nyckeln.

### Lägg till Web API controllers

I denna del kommer vi att lägga till API controllers som gör att vi kan utföra CRUD operationer med hjälp av http-requests mot vår URI i webbläsarfältet.

Öppna mappen Controllers i din solution explorer och ta bort ValuesController och bygg projektet.

Lägg sedan till en ny Controller och välj sedan "Web API 2 Controller with actions, using Entity Framework".



När du klickat på Add, Välj Director som din model class.  
 Klicka på plusset och sedan add för att lägga till context.  
 Bocka i "Use async controller actions".  
 Låt den heta DirectorsController

Model class: Director (MovieService.Models)

Data context class: MovieServiceContext (MovieService.Models) +

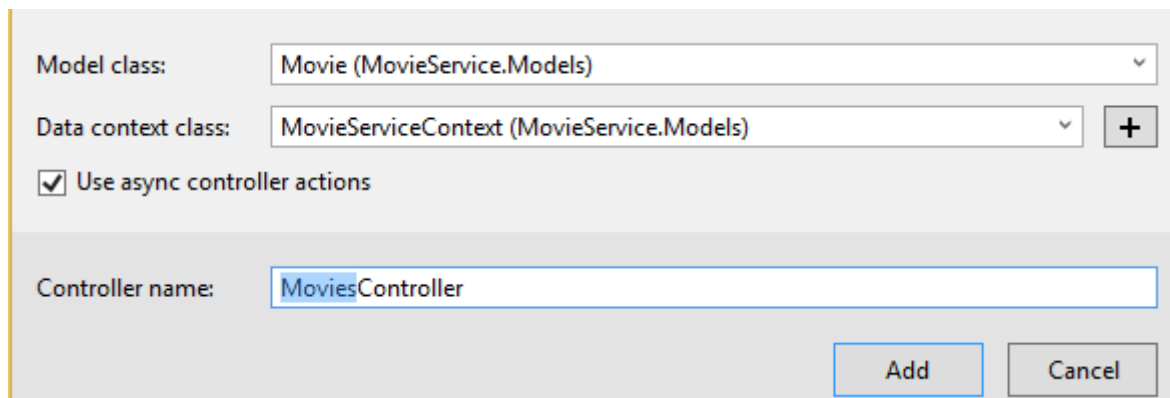
☒ Use async controller actions

Controller name: DirectorsController

Add Cancel

Bygg ditt projekt igen.

Upprepa processen fast med en MovieController. Använd MovieServiceContext för denna också.



Om ni nu tittar i filerna `MoviesController` och `DirectorsController` finns det ett färdig API där ni kan anropa bland annat

`api/Movies`

och

`api/Movies/id`

vilket då returnerar data. Dock inte för tillfället då vi inte har någon data, men det fixar vi i nästa steg!

### [Code First Migrations för att lägga en Seed i databasen](#)

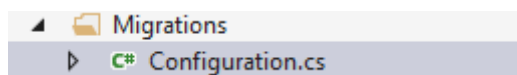
I denna del kommer du att skapa en seed för databasen med testdata.

I den tools meny, välj NuGet Package manager och sedan Package Manager Console. Här i ska vi skriva en kommando som skapar en mapp med namnet "Migrations" i projektet och en fil i denna som heter `Configuration.cs`. Här i kommer vi sedan att lägga vår seed till databasen.

I konsolen du har öppnat, skriv följande:

```
Enable-Migrations -ContextTypeName MovieService.Models.MovieServiceContext
```

Det ska nu finnas en mapp i ditt projekt som heter "Migrations", öppna filen `Configuration.cs` i denna.



Lägg till följande using:

```
using MovieService.Models;
```

Nedan följer en seed-kod för att fylla vår databas med testdata. Om du inte vill skriva av den kan du kopiera den från `snippet.txt` som ligger i samma mapp som denna tutorial. Om du läser wordversionen av detta dokument kan du bara dubbelklicka på bilden nedan så får du upp den i kod.

```

protected override void Seed(MovieService.Models.MovieServiceContext context)
{
    context.Directors.AddOrUpdate(x => x.Id,
new Director() { Id = 1, Name = "Tim Burton" },
new Director() { Id = 2, Name = "Christopher Nolan" },
new Director() { Id = 3, Name = "Michael Bay" }
);

    context.Movies.AddOrUpdate(x => x.Id,
new Movie()
{
    Id = 1,
    Title = "Corpse Bride",
    Year = 2005,
    DirectorId = 1,
    Price = 18M,
    Genre = "Animation"
},
new Movie()
{
    Id = 2,
    Title = "Batman Begins",
    Year = 1817,
    DirectorId = 2,
    Price = 100M,
    Genre = "Action"
},
new Movie()
{
    Id = 3,
    Title = "Gravity",
    Year = 2014,
    DirectorId = 2,
    Price = 200M,
    Genre = "Space"
},
new Movie()
{
    Id = 4,
    Title = "Transformers 3",
    Year = 2014,
    DirectorId = 3,
    Price = 200M,
    Genre = "Robots"
}
);
}

```

Skriv in följande två kommandon och låt magin ske!

1. Add-Migration Initial

2. update-database

När denna har exekverat färdigt kan du starta ditt projekt och välja API i menyn. Där inne kan du se dokumentation för ditt API och vilka URI som används för att göra CRUD-operationer.

Just nu mappar vårt API direkt till våra databasentiteter och det är inte alltid vi vill att det fungerar på det sättet. För att ändra på detta kommer vi att skapa en sorts "mellanhand" som kallas för DTO,

data transfer object. En DTO definierar hur data kommer att skickas över nätverket. I vår modelmapp, skapa en MovieDTO och en MovieDetailDTO.

Den första ska bestå av en tunnare information av en movie och den andra en mer detaljerad version. Se klasserna nedanför

```
namespace MovieService.Models
{
    public class MovieDTO
    {
        public int Id { get; set; }
        public string Title { get; set; }
        public string DirectorName { get; set; }
    }
}
```

```
namespace MovieService.Models
{
    public class MovieDetailDTO
    {
        public int Id { get; set; }
        public string Title { get; set; }
        public int Year { get; set; }
        public decimal Price { get; set; }
        public string DirectorName { get; set; }
        public string Genre { get; set; }
    }
}
```

Nästa steg är att ändra GET-metoderna i MoviesController klassen så att de istället returnerar DTOs. Vi använder LINQ för att konvertera Movie entiteter till DTOs. Det vi gör är att vi skapar ett objekt som kommer att skickas till användaren i den form vi vill istället för att använda det standardAPI som kommer med WEB API.

```

public IQueryable<MovieDTO> GetMovies()
{
    var movies = from m in db.Movies
                  select new MovieDTO()
                  {
                      Id = m.Id,
                      Title = m.Title,
                      DirectorName = m.Director.Name
                  };
    return movies;
}

// GET: api/Movies/5
[ResponseType(typeof(MovieDetailDTO))]
public async Task<IHttpActionResult> GetMovie(int id)
{
    var movie = await db.Movies.Include(m => m.Director)
                               .Select(m => new MovieDetailDTO()
                               {
                                   Id = m.Id,
                                   Title = m.Title,
                                   Year = m.Year,
                                   Price = m.Price,
                                   DirectorName = m.Director.Name,
                                   Genre = m.Genre
                               }).SingleOrDefaultAsync(m => m.Id == id);

    if(movie == null)
    {
        return NotFound();
    }

    return Ok(movie);
}

```

Sist men inte minst, modifiera PostMovie så att den returnerar en DTO.



```

// POST: api/Movies
[ResponseType(typeof(Movie))]
public async Task<IHttpActionResult> PostMovie(Movie movie)
{
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }

    db.Movies.Add(movie);
    await db.SaveChangesAsync();

    //Ny kod
    db.Entry(movie).Reference(x => x.Director).Load();

    var dto = new MovieDTO()
    {
        Id = movie.Id,
        Title = movie.Title,
        DirectorName = movie.Director.Name
    };

    return CreatedAtRoute("DefaultApi", new { id = movie.Id }, dto);
}

```

Ni kan nu gå in på er hemsida och test API:t genom att skriva in de URLs som ligger under den fliken på hemsidan för att testa att få ut data. Imorgon kommer vi att prata om applikationens Frontend och bygga en klient som kan prata med vårt nya API och utnyttja dess data!