Lesson 8

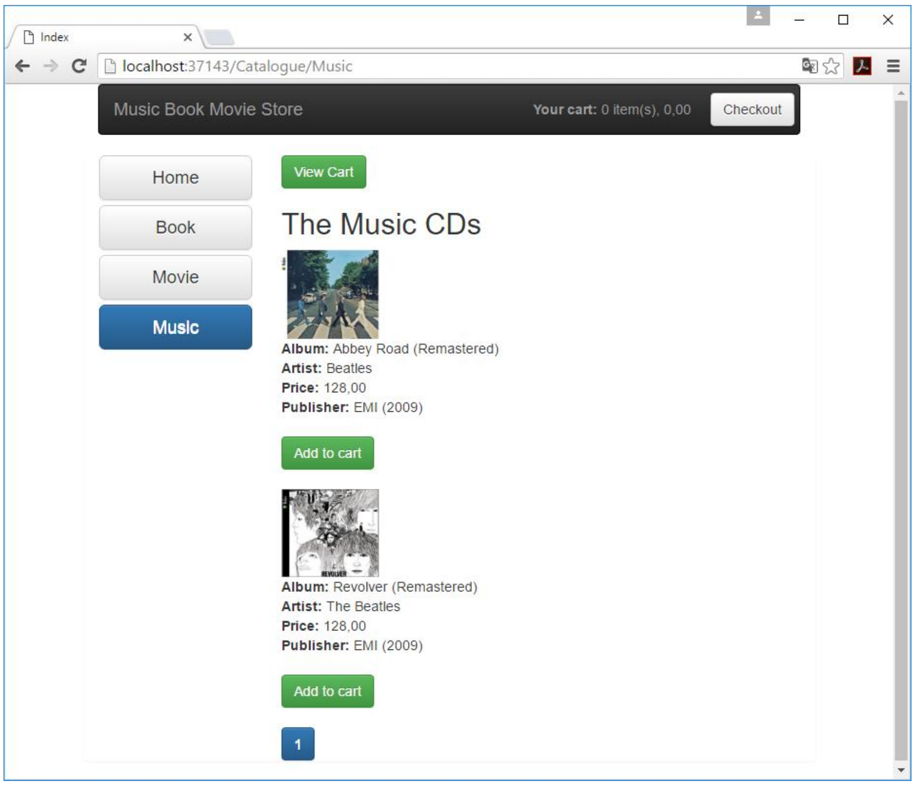
In these and the upcoming lesson 9 exercises we'll use Entity Framework (EF) to add a database to the project and store data in the database and display data dynamically from the database. We will continue to work with the **MusicBookMovieStore** project and modify the selected parts of the code necessary to connect and communicate with the database.

The exercises are based on Dykstra, Tom and Rick Anderson: Getting Started with Entity Framework 6 Code First using MVC 5 and adapted to the MusicBookMovieStore project.

Exercise 1, Install Entity Framework

To get started, I recommend that you download the 1st semester > Backendprogrammering > Exercises > Lesson08_MbmStore_startup.zip file from Fronter. By doing that you are sure to have a code base that exactly match the examples the exercises.

Unzip the folder and open the project in Visual Studio. Build (Ctrl+Shift+B) and Run the project (Ctrl+ F5):



Album: Abbey Road (Remastered)
Artist: Beatles
Price: 128,00
Publisher: EMI (2009)

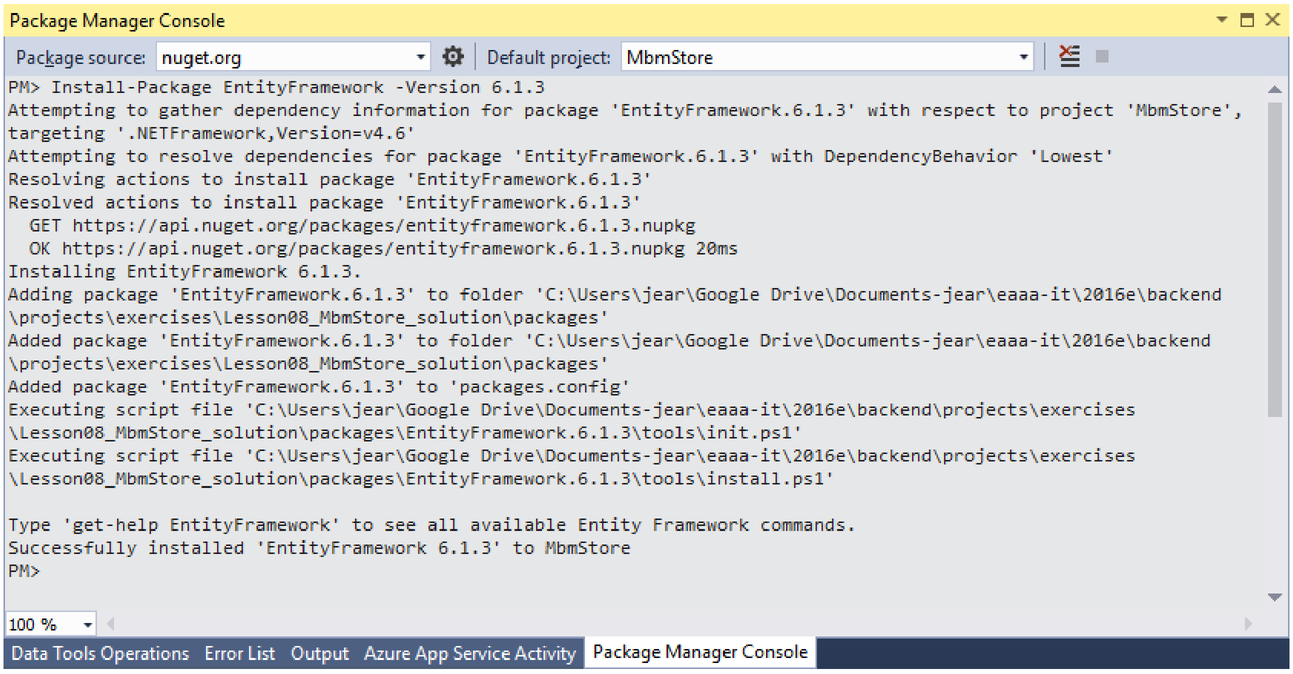
Album: Revolver (Remastered)
Artist: The Beatles
Price: 128,00
Publisher: EMI (2009)

Exercise 2, Install Entity Framework 6

From the Tools menu click Library Package Manager and then click Package Manager Console.

In the **Package Manager Console** window enter the following command:

```
Install-Package EntityFramework -Version 6.1.3
```



```
PM> Install-Package EntityFramework -Version 6.1.3
Attempting to gather dependency information for package 'EntityFramework.6.1.3' with respect to project 'MbmStore',
targeting '.NETFramework,Version=v4.6'
Attempting to resolve dependencies for package 'EntityFramework.6.1.3' with DependencyBehavior 'Lowest'
Resolving actions to install package 'EntityFramework.6.1.3'
Resolved actions to install package 'EntityFramework.6.1.3'
  GET https://api.nuget.org/packages/entityframework.6.1.3.nupkg
  OK https://api.nuget.org/packages/entityframework.6.1.3.nupkg 20ms
Installing EntityFramework 6.1.3.
Adding package 'EntityFramework.6.1.3' to folder 'C:\Users\jear\Google Drive\Documents-jear\eaaa-it\2016e\backend
\projects\exercises\Lesson08_MbmStore_solution\packages'
Added package 'EntityFramework.6.1.3' to folder 'C:\Users\jear\Google Drive\Documents-jear\eaaa-it\2016e\backend
\projects\exercises\Lesson08_MbmStore_solution\packages'
Added package 'EntityFramework.6.1.3' to 'packages.config'
Executing script file 'C:\Users\jear\Google Drive\Documents-jear\eaaa-it\2016e\backend\projects\exercises
\Lesson08_MbmStore_solution\packages\EntityFramework.6.1.3\tools\init.ps1'
Executing script file 'C:\Users\jear\Google Drive\Documents-jear\eaaa-it\2016e\backend\projects\exercises
\Lesson08_MbmStore_solution\packages\EntityFramework.6.1.3\tools\install.ps1'

Type 'get-help EntityFramework' to see all available Entity Framework commands.
Successfully installed 'EntityFramework 6.1.3' to MbmStore
PM>
```

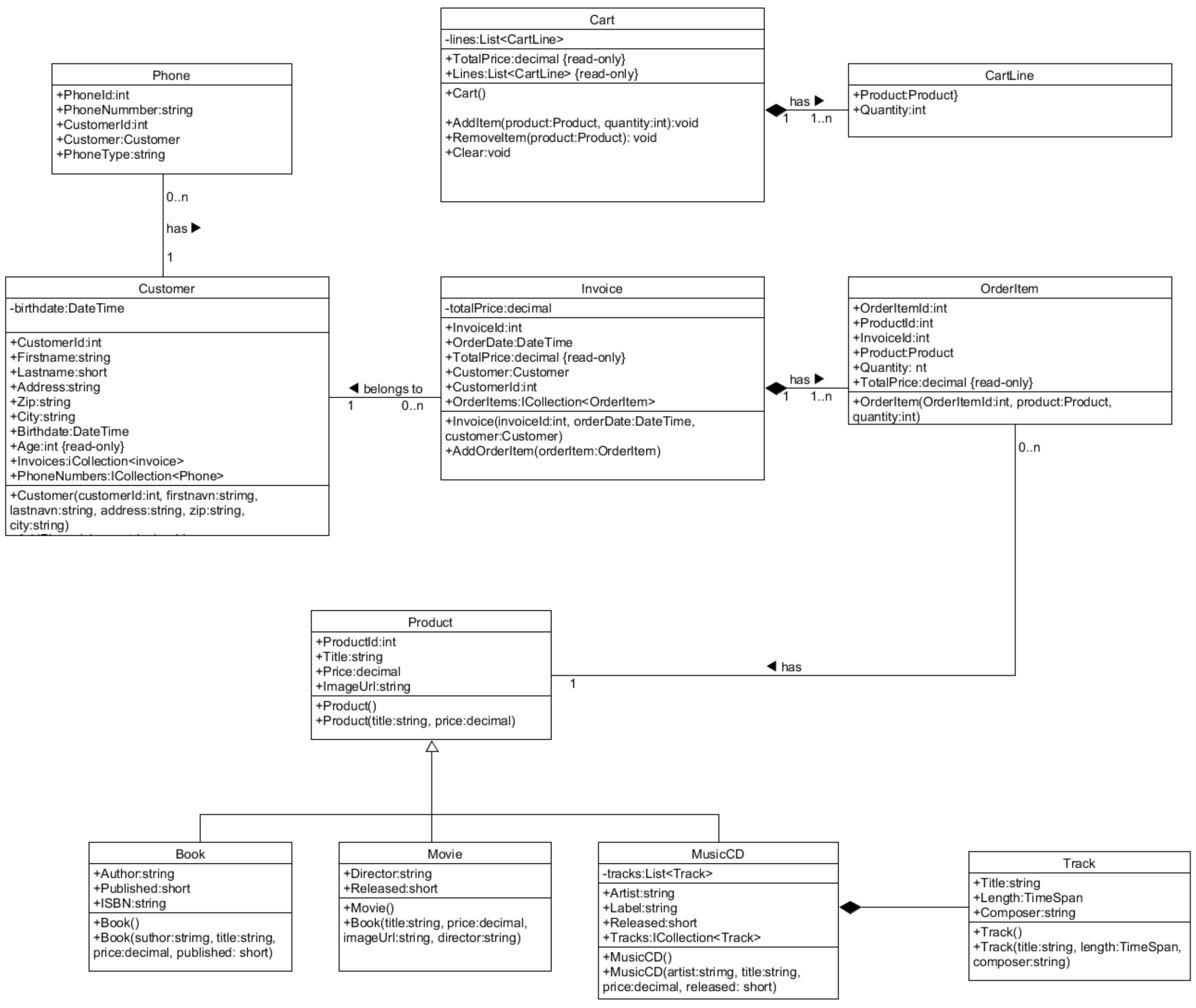
The image shows version 6.1.3 being installed. If you write:

```
Install-Package EntityFramework
```

NuGet will install the latest version of EF (excluding pre-release versions), which as of the most recent update to these exercises is 6.1.3.

This step is one of a few steps that this tutorial has you do manually, but which could have been done automatically by the ASP.NET MVC scaffolding feature. You're doing them manually so that you can see the steps required to use the EF. You'll use scaffolding later to create the MVC controller and views. An alternative is to let scaffolding automatically install the EF NuGet package, create the database context class, and create the connection string. When you're ready to do it that way, all you have to do is skip those steps and scaffold your MVC controller after you create your entity classes.

Exercise 3, Decorate the Data Model


Cart
-lines:List<CartLine>
+TotalPrice:decimal {read-only}
+Lines:List<CartLine> {read-only}
+Cart()
+AddItem(product:Product, quantity:int):void
+RemoveItem(product:Product): void
+Clear:void
has
1
1..n
CartLine
+Product:Product}
+Quantity:int
Phone
+PhoneId:int
+PhoneNummber:string
+CustomerId:int
+Customer:Customer
+PhoneType:string
0..n
has
1
Customer
-birthdate:DateTime
+CustomerId:int
+Firstname:string
+Lastname:short
+Address:string
+Zip:string
+City:string
+Birthdate:DateTime
+Age:int {read-only}
+Invoices:iCollection<invoice>
+PhoneNumbers:ICollection<Phone>
+Customer(customerId:int, firstnavn:strimg, lastnavn:string, address:string, zip:string, city:string)
belongs to
1
0..n
Invoice
-totalPrice:decimal
+InvoiceId:int
+OrderDate:DateTime
+TotalPrice:decimal {read-only}
+Customer:Customer
+CustomerId:int
+OrderItems:ICollection<OrderItem>
+Invoice(invoiceId:int, orderDate:DateTime, customer:Customer)
+AddOrderItem(orderItem:OrderItem)
has
1
1..n
OrderItem
+OrderItemId:int
+ProductId:int
+InvoiceId:int
+Product:Product
+Quantity: nt
+TotalPrice:decimal {read-only}
+OrderItem(OrderItemId:int, product:Product, quantity:int)
0..n
has
1
Product
+ProductId:int
+Title:string
+Price:decimal
+ImageUrl:string
+Product()
+Product(title:string, price:decimal)
Book
+Author:string
+Published:short
+ISBN:string
+Book()
+Book(suthor:strimg, title:string, price:decimal, published: short)
Movie
+Director:string
+Released:short
+Movie()
+Book(title:string, price:decimal, imageUrl:string, director:string)
MusicCD
-tracks:List<Track>
+Artist:string
+Label:string
+Released:short
+Tracks:ICollection<Track>
+MusicCD()
+MusicCD(artist:strimg, title:string, price:decimal, released: short)
Track
+Title:string
+Length:TimeSpan
+Composer:string
+Track()
+Track(title:string, length:TimeSpan, composer:string)

If we – to start with – look at the Customer UML class the corresponding *class.css* has the following code:

```
using System;
using System.Collections.Generic;

namespace MbmStore.Models
{
    public class Customer
    {
        public int CustomerId { get; set; }
        public string Firstname { get; set;  }
        public string Lastname { get; set; }
        public string Address { get; set; }
        public string Zip { get; set; }
        public string City { get; set; }
        public string Email { get; set; }
        public DateTime Birthdate { get; set; }
        // read only property
        public int Age
        {
            get
            {
                DateTime now = DateTime.Now;
                int age = 0;
                age = now.Year - Birthdate.Year;
                if (now.Month < Birthdate.Month ||
                    (now.Month == Birthdate.Month && now.Day < Birthdate.Day))
                {
                    age--;
                }
                return age;
            }
        }

        public virtual ICollection<Invoice> Invoices { get; set; }
        public virtual ICollection<Phone> PhoneNumbers { get; set; }



         // constructor
         public Customer() {}

         public Customer(int customerId, string firstnavn, string lastnavn, string
         address, string zip, string city)
        {
            CustomerId = customerId;
            Firstname = firstnavn;
            Lastname = lastnavn;
            Address = address;
            Zip = zip;
            City = city;
        }

        // method
        public void AddPhone(Phone phone)
        {
            PhoneNumbers.Add(phone);
        }

    }
}
```

The `CustomerId` property will become the primary key column of the database table that corresponds to this class. By default, the EF interprets a property that's named `ID` or *classname*`ID` as the primary key.

The `Invoices` and PhoneNumbers properties are *navigation properties*. Navigation properties hold other entities that are related to this entity. In this case, the PhoneNumbers property of a `Customer` entity will hold all of the `Phone` entities that are related to that `Customer` entity. In other words, if a given `Customer` row in the database has two related Phone rows (rows that contain that customer's primary key value in their `CustomerId` foreign key column), that `Customer` entity's PhoneNumbers navigation property will contain those two `Phone` entities.

Navigation properties are typically defined as `virtual` so that they can take advantage of certain EF functionality such as *lazy loading*. (Lazy loading will be explained later, in the next lesson.)

If a navigation property can hold multiple entities (as in many-to-many or one-to-many relationships), its type must be a list in which entries can be added, deleted, and updated, such as `ICollection`.

Let's take a look at the Phone class:

```
namespace MbmStore.Models
{
    public class Phone
    {
        public int PhoneId { get; set; }
        public string Number { get; set; }
        public int CustomerId { get; set; }

        public Customer Customer { get; set; }
    }
}
```

The `CustomerId` property becomes a foreign key in the Phone table, and the corresponding navigation property is `Customer`. A `Phone` entity is associated with one `Customer` entity, so the property can only hold a single `Single` entity (unlike the `Customer.PhoneNumbers` navigation property you saw earlier, which can hold multiple `Enrollment` entities).

EF interprets a property as a foreign key property if it's named *<navigation property name><primary key property name>* (for example, `CustomerCustomerId` for the `Customer` navigation property since the `Customer` entity's primary key is `CustomerId`). Foreign key properties can also be named the same simply *<primary key property name>* (for example, `CustomerId` since the `Customer` entity's primary key is `CustomerId`).

Inheritance

Datetime2 is the recommended type for dates and times in SQL Server 2008 onwards. You will need to explictly map the relevant columns to datetime2 since EF will always map `.Net DateTimes` to the SQL Server `datetime` type. You can do this via the fluent API as we'll see in the next lesson, or you can use attributes on your model's properties as you see for the `Customer` and `Invoice` classes:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations.Schema;

namespace MbmStore.Models
{
    public class Customer
    {
        public int CustomerId { get; set; }
        public string Firstname { get; set;  }
        public string Lastname { get; set; }
        public string Address { get; set; }
        public string Zip { get; set; }
        public string City { get; set; }
        public string Email { get; set; }

        [Column(TypeName = "datetime2")]
        // code removed for brevity
        ...

    }
}
```

and

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations.Schema;
using System.Linq;

namespace MbmStore.Models
{
    public class Invoice
    {

        private decimal totalPrice;
        private List<OrderItem> orderItems = new List<OrderItem>();

        public int InvoiceId { get; set; }
        [Column(TypeName = "datetime2")]
        public DateTime OrderDate { get; set; }
        // code removed for brevity
        ...
    }
}
```

For more information on this subject see: http://www.mikesdotnetting.com/article/229/conversion-of-a-datetime2-data-type-to-a-datetime-data-type-resulted-in-an-out-of-range-value.

Inheritance

Inheritance in EF provides a way to create the required logical entities to act on a set of database tables and also to create a more meaningful relationship between entities using inheritance.

There are three type of inheritance relationships in the EF:

- Table per hierarchy (TPH)
- Table per type (TPT)
- Table per concrete type (TPC)

Table per hierarchy is the default behavior and it will create one single table for one class hierarchy. For the MbmStore project that means there will be one single product table:

Product table

Attribute	Attribute from class
ProductId	Product
Title	Product
Price	Product
ImageUrl	Product
Category	Product

Artist	MusicCD
Label	MusicCD
Released	MusicCD
Author	Book
Publisher	Book
Published	Book
ISBN	Book
Director	Movie
Released1	Movie

TPH inheritance patterns generally deliver better performance in the EF than TPT inheritance patterns, because TPT patterns can result in complex join queries. From a database model perspective however, the TPT patterns are preferable because it does not produce null values, and we'll stick with that pattern in these exercises.

To instruct EF to use this patterns we have 2 options. We can annotate the types with the `[Table()]` attribute or we can use the Fluent API to instruct EF to map the Product entity to the specific tables. We'll introduce the Fluent API in the next lesson and use annotations:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations.Schema;


namespace MbmStore.Models
{

    [Table("MusicCD")]
    public class MusicCD : Product {

        public string Artist { get; set; }
        public string Label { get; set; }
        public short Released { get; set; }
        private List<Track> tracks = new List<Track>();
        // code removed for brevity
        ...
    }
}
```

Add the same kind of table annotations for Book and `Movie` subclasses.

Exercise 4, Create the Database Context

The main class that coordinates EF functionality for a given data model is the *database context* class. You create this class by deriving from the System.Data.Entity.DbContext class. In your code you specify which entities are included in the data model. You can also customize certain EF behavior. In this project, the class is named `SchoolContext`.

To create a folder in the MbmStore project, right-click the project in **Solution Explorer** and click **Add**, and then click **New Folder**. Name the new folder *DAL* (for Data Access Layer). In that folder create a new class file named *MbmStoreContext.cs*, and replace the template code with the following code:

```
using MbmStore.Models;
using System;
using System.Collections.Generic;
using System.Data.Entity;
using System.Data.Entity.ModelConfiguration.Conventions;

namespace MbmStore.DAL
{
    public class MbmStoreContext : DbContext
    {
        public MbmStoreContext() : base("MbmStoreContext")
        {
        }

        public DbSet<Customer> Customers {get; set;}
        public DbSet<Invoice> Invoices {get; set;}
        public DbSet<OrderItem> OrderItems {get; set;}
        public DbSet<Phone> Phones {get; set;}
        public DbSet<Product> Products {get; set;}
        public DbSet<Book> Books { get; set; }
        public DbSet<MusicCD> MusicCDs { get; set; }
        public DbSet<Movie> Movies { get; set; }

        public DbSet<Track> Tracks {get; set;}

        protected override void OnModelCreating(DbModelBuilder modelBuilder)
        {
            modelBuilder.Conventions.Remove<PluralizingTableNameConvention>();
        }
    }
}
```

Specifying entity sets

This code creates a DbSet property for each entity set. In EF terminology, an *entity set* typically corresponds to a database table, and an *entity* corresponds to a row in the table.

You could have omitted the DbSet<Phone>, DbSet<Invoice> and DbSet<OrderItem> statements and it would work the same. The EF would include them implicitly because the Customer entity references the Phone entity and the Invoice entity and the Invoice entity references the OrerItem entity.

Specifying the connection string

The name of the connection string (which you'll add to the Web.config file later) is passed in to the constructor.

```
public MbmMbmStoreContext() : base("MbmMbmStoreContext")
{
}
```

You could also pass in the connection string itself instead of the name of one that is stored in the Web.config file. For more information about options for specifying the database to use, see Entity Framework - Connections and Models.

If you don't specify a connection string or the name of one explicitly, EF assumes that the connection string name is the same as the class name. The default connection string name in this example would then be `MbmMbmStoreContext`, the same as what you're specifying explicitly.

Specifying singular table names

The `modelBuilder.Conventions.Remove` statement in the OnModelCreating method prevents table names from being pluralized. If you didn't do this, the generated tables in the database would be named `Customers`, `Phones`, and `Products` etc. Instead, the table names will be `Student`, `Phone`, and `Product`. Developers disagree about whether table names should be pluralized or not. These exercises use the singular form, but the important point is that you can select whichever form you prefer by including or omitting this line of code.

Set up EF to initialize the database with test data

The EF can automatically create (or drop and re-create) a database for you when the application runs. You can specify that this should be done every time your application runs or only when the model is out of sync with the existing database. You can also write a `Seed` method that the EF automatically calls after creating the database in order to populate it with test data.

The default behavior is to create a database only if it doesn't exist (and throw an exception if the model has changed and the database already exists). In this section you'll specify that the database should be dropped and re-created whenever the model changes. Dropping the database causes the loss of all your data. This is generally OK during development, because the `Seed` method will run when the database is re-created and will re-create your test data. But in production you generally don't want to lose all your data every time you need to change the database schema. Later you'll see how to handle model changes by using Code First Migrations to change the database schema instead of dropping and re-creating the database.

In the DAL folder, create a new class file named `MbmStoreInitializer.cs` and replace the template code with the following code, which causes a database to be created when needed and loads test data into the new database.

```
using MbmStore.Models;
using System;
using System.Collections.Generic;

namespace MbmStore.DAL
{
    public class MbmStoreInitializer :
System.Data.Entity.DropCreateDatabaseIfModelChanges<MbmMbmStoreContext>
    {
        protected override void Seed(MbmMbmStoreContext context)
        {

            // Products
            var products = new List<Product>
                {
                    new Book {ProductId=1, Title="A Hard Day's Write: The Stories Behind
Every Beatles Song ", Author="Steve Turner", Price=150.00M, Publisher="It Books",
Published=2005, ISBN="978-0060844097", ImageUrl="A_Hard_Days_Write.jpg", Category="Book"},

                    new Book {ProductId=2, Title="With a Little Help from My Friends: The
Making of Sgt. Pepper", Author="Georg Martin", Price=180.00M, Publisher="Little Brown & Co",
Published=1995, ISBN="0316547832", ImageUrl="The Making of Sgt. Pepper.jpg",
Category="Book"},
                    new MusicCD { ProductId=3, Title="Abbey Road (Remastered)",
Artist="Beatles", Price=128.00M, Released=2009, Label="EMI", ImageUrl="abbey_road.jpg",
Category="Music", Tracks=new List<Track> {
                        new Track {Title="Come Together", Length=new TimeSpan(0, 4, 20),
Composer="Lennon, McCartney"},
                        new Track {Title="Something", Length=new TimeSpan(0, 3, 03),
Composer="Harrison"},
                        new Track {Title="Maxwell's Silver Hammer", Length=new TimeSpan(0,
3, 29), Composer="Lennon, McCartney"},
                        new Track {Title="Oh! Darling", Length=new TimeSpan(0, 3, 26),
Composer="Lennon, McCartney"},
                        new Track {Title="Octopus's Garden", Length=new TimeSpan(0, 2, 51),
Composer="Starkey"},
                        new Track {Title="I Want You (She's So Heavy)", Length=new
TimeSpan(0, 7, 47), Composer="Lennon, McCartney"},
                        new Track {Title="Here Comes The Sun", Length=new TimeSpan(0, 3,
05), Composer="Harrison"},
                        new Track {Title="Because", Length=new TimeSpan(0, 2, 45),
Composer="Lennon, McCartney"},
                        new Track {Title="You Never Give Me Your Money", Length=new
TimeSpan(0, 4, 02), Composer="Lennon, McCartney"},
                        new Track {Title="Sun King", Length=new TimeSpan(0, 2, 26),
Composer="Lennon, McCartney"},
                        new Track {Title="Mean Mr. Mustard", Length=new TimeSpan(0, 1, 6),
Composer="Lennon, McCartney"},
                        new Track {Title="Polythene Pam", Length=new TimeSpan(0, 1, 12),
Composer
                       ="Lennon, McCartney"},
                        new Track {Title="She Came In Through The Bathroom Window",
Length=new TimeSpan(0, 1, 57), Composer="Lennon, McCartney"},
                        new Track {Title="Golden Slumbers", Length= new TimeSpan(0, 1,
31),Composer="Lennon, McCartney"},
                        new Track {Title="Carry That Weight", Length=new TimeSpan(0, 1, 36),
Composer="Lennon, McCartney"},
                        new Track {Title="The End", Length=new TimeSpan(0, 2, 19),
Composer="Lennon, McCartney"},
                        new Track {Title="Her Majesty", Length=new TimeSpan(0, 0, 23),
Composer="Lennon, McCartney"}
                       }
                   },
                    new MusicCD {ProductId=4, Title="Revolver (Remastered)", Artist="The
Beatles", Price=128.00M, Released=2009, Label="EMI", ImageUrl="revolver.jpg",
Category="Music", Tracks=new List<Track> {
                         new Track {Title="Taxman", Length=new TimeSpan(0, 2, 28),
Composer="Harrison"},
                         new Track {Title="Eleanor Rigby", Length=new TimeSpan(0, 2, 6),
Composer="Lennon, McCartney"},
                         new Track {Title="I'm Only Sleeping", Length=new TimeSpan(0, 3, 0),
Composer="Lennon, McCartney"},
                         new Track {Title="Love You To", Length=new TimeSpan(0, 2, 59),
Composer="Harrison"},
                         new Track {Title="Here, There And Everywhere", Length=new
TimeSpan(0, 2, 23), Composer="Harrison"},
                        new Track {Title="Yellow Submarine", Length=new TimeSpan(0, 2, 38),
Composer="Lennon, McCartney"},
                        new Track {Title="She Said She Said", Length=new TimeSpan(0, 2, 36),
Composer="Lennon, McCartney"},
                        new Track {Title="Good Day Sunshine", Length=new TimeSpan(0, 2, 9),
Composer="Lennon, McCartney"},
                        new Track {Title="And Your Bird Can Sing", Length=new TimeSpan(0, 2, 
0), Composer="Lennon, McCartney"},
                        new Track {Title="For No One", Length=new TimeSpan(0, 1, 59), 
Composer="Lennon, McCartney"},
                        new Track {Title="Doctor Robert", Length=new TimeSpan(0, 1, 14), 
Composer="Lennon, McCartney"},
                        new Track {Title="I Want To Tell You", Length=new TimeSpan(0, 2, 
27), Composer="Harrison"},
                        new Track {Title="Got To Get You Into My Life", Length=new 
TimeSpan(0, 2, 29), Composer="Lennon, McCartney"},
                        new Track {Title="Tomorrow Newer Knows", Length=new TimeSpan(0, 3, 
01), Composer="Lennon, McCartney"}
                     }
                    },
                    new Movie {ProductId=5, Title="Jungle Book", Price=160.50M, 
ImageUrl="junglebook.jpg", Director="Jon Favreau", Category="Movie"},
                    new Movie {ProductId=6, Title="Gladiator", Price=49.95M, 
ImageUrl="gladiator.jpg", Director="Ridley Scott", Category="Movie"},
                    new Movie {ProductId=7, Title="Forrest Gump", Price=160.50M, 
ImageUrl="forrest-gump.jpg", Director="Robert Zemeckis", Category="Movie"}
                };

            // populate the database
            products.ForEach(p => context.Products.Add(p));
            context.SaveChanges();

            // Customers
            var customers = new List<Customer>
                {
                    new Customer {CustomerId=1, 
Firstname="Tina",Lastname="Petterson",Address="Irisdahlsvej 32", Zip="8200", City="Århus 
N"},
                    new Customer{CustomerId=2, Firstname="Thomas", Lastname="Larsson", 
Address="Solsikkevej 32", Zip="8000", City="Århus C"}
                };

            // populate the database
            customers.ForEach(c => context.Customers.Add(c));
            context.SaveChanges();

            var invoices = new List<Invoice>
                {
                    new Invoice {InvoiceId=1, OrderDate=new DateTime(2016, 09, 12), 
CustomerId=1,
                        OrderItems=new List<OrderItem> {
                            new OrderItem {ProductId=7, Quantity=1},
                            new OrderItem {ProductId=2, Quantity=1}
                        }
                    },
                    new Invoice {InvoiceId=2, OrderDate=new DateTime(2016, 09, 18), 
CustomerId=2,
                        OrderItems=new List<OrderItem> {
                            new OrderItem {OrderItemId=1, ProductId=1, Quantity=1},
                            new OrderItem {OrderItemId=2, ProductId=3, Quantity=1}
                        }
                    }
                };

            // populate the database
            invoices.ForEach(i => context.Invoices.Add(i));
```

```
                context.SaveChanges();
            }
        }
}
```

The `Seed` method takes the database context object as an input parameter, and the code in the method uses that object to add new entities to the database. For each entity type, the code creates a collection of new entities, adds them to the appropriate `DbSet` property, and then saves the changes to the database. It isn't necessary to call the `SaveChanges` method after each group of entities, as is done here, but doing that helps you locate the source of a problem if an exception occurs while the code is writing to the database.

To tell EF to use your initializer class, add an element to the `entityFramework` element in the application *Web.config* file (the one in the root project folder), as shown in the following example:

```
  <appSettings>
    <add key="webpages:Version" value="3.0.0.0" />
    <add key="webpages:Enabled" value="false" />
    <add key="ClientValidationEnabled" value="true" />
    <add key="UnobtrusiveJavaScriptEnabled" value="true" />
    <add key="DatabaseInitializerForType MbmStore.DAL.MbmStoreContext, MbmStore"
value="MbmStore.DAL.MbmStoreInitializer, MbmStore" />
  </appSettings>
```

The key starts with `DatabaseInitializerForType` and includes the fully qualified assembly name for your Context object. The value is the fully qualified assembly name for the `MbmStoreInitializer` database initializer class.

As an alternative to setting the initializer in the *Web.config* file is to do it in code by adding a `Database.SetInitializer` statement to the `Application_Start` method in in the *Global.asax.cs* file. For more information, see Understanding Database Initializers in Entity Framework Code First.

The application is now set up so that when you access the database for the first time in a given run of the application, the EF compares the database to the model (your `MbmMbmStoreContext` and entity classes). If there's a difference, the application drops and re-creates the database.

Note: When you deploy an application to a production web server, you must remove or disable code that drops and re-creates the database. You'll do that in the next lesson.

Build the project (Ctrl+Shift+B) to ensure that there are no compile errors in the code.

Exercise 5, Setup EF to use a SQL Server Express LocalDB database

LocalDB is a lightweight version of the SQL Server Express Database Engine. It's easy to install and configure, starts on demand, and runs in user mode. LocalDB runs in a special execution mode of SQL Server Express that enables you to work with databases as *.mdf* files. You can put LocalDB database files in the *App_Data* folder of a web project if you want to be able to copy the database with the project. The user instance feature in SQL Server Express also enables you to work with *.mdf* files, but the user instance feature is deprecated; therefore, LocalDB is recommended for working with *.mdf* files. In Visual Studio 2012 and later versions, LocalDB is installed by default with Visual Studio.

Typically SQL Server Express is not used for production web applications. LocalDB in particular is not recommended for production use with a web application because it is not designed to work with IIS.

In this and upcoming exercises you'll work with LocalDB. Open the application *Web.config* file and add a `connectionStrings` element preceding the `appSettings` element, as shown in the following example. (Make sure you update the *Web.config* file in the root project folder. There's also a *Web.config* file is in the *Views* subfolder that you don't need to update.)

```
<configuration>
  <connectionStrings>
    <add name="MbmStoreContext" connectionString="Data Source=(LocalDB)\MSSQLLocalDB;;
Integrated Security=True; MultipleActiveResultSets=True;
AttachDbFilename=|DataDirectory|MbmStore1.mdf" providerName="System.Data.SqlClient" />
  </connectionStrings>

   ...
</configuration>
```

The connection string you've added specifies that EF will use a LocalDB database named *MbmStore1.mdf*. The database doesn't exist yet; EF will create it. The database will be created in your *App_Data* folder. For more information about connection strings, see SQL Server Connection Strings.

You don't actually have to have a connection string in the *Web.config* file. If you don't supply a connection string, EF will use a default one based on your context class. For more information, see Code First to a New Database.

Build (Ctrl+Shift+B) and run (Ctrl+F5) the project to ensure that everything still works as expected.

Exercise 6, Retrieve data from the database

When you open the `CatalogueController` class (*CatalogueController.cs*) , you'll see we're still using the old hard coded `Repository` class in the *Infrastructure* folder:

```
using MbmStore.Infrastructure;
using MbmStore.ViewModels;
using System.Linq;
using System.Web.Mvc;

namespace MbmStore.Controllers
{
    public class CatalogueController : Controller
    {

        public int PageSize = 4;

        // GET: Catalogue
        public ActionResult Index(string category, int page = 1)
        {
            Repository repository = new Repository();
            ProductsListViewModel model = new ProductsListViewModel
            {
                Products = repository.Products
                .Where(p => category == null || p.Category == category)
               .OrderBy(p => p.ProductId)
               .Skip((page - 1) * PageSize)
               .Take(PageSize),

                PagingInfo = new PagingInfo
                {
                    CurrentPage = page,
                    ItemsPerPage = PageSize,
                    TotalItems = category == null ?
                    repository.Products.Count() :
                    repository.Products.Where(e => e.Category == category).Count()
                },
                CurrentCategory = category
            };
            return View(model);
        }
    }

}
```

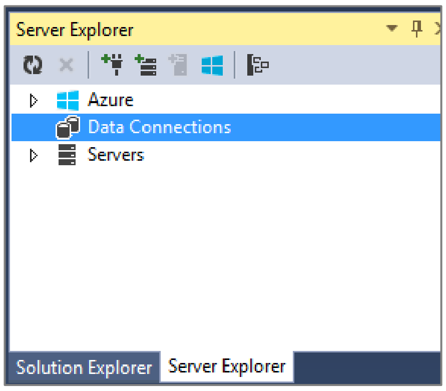
To change that and connect to the database we'll create a new object and use that for connecting to the database:

```
using MbmStore.DAL;
using MbmStore.ViewModels;
using System.Linq;
using System.Web.Mvc;

namespace MbmStore.Controllers
{
    public class CatalogueController : Controller
    {

        private MbmStoreContext db;

        public int PageSize = 4;

        // GET: Catalogue
        public ActionResult Index(string category, int page = 1)
        {
            db = new MbmStoreContext();
            ProductsListViewModel model = new ProductsListViewModel
            {
                Products = db.Products
                .Where(p => category == null || p.Category == category)
               .OrderBy(p => p.ProductId)
               .Skip((page - 1) * PageSize)
               .Take(PageSize).ToList(),

                PagingInfo = new PagingInfo
                {
                    CurrentPage = page,
                    ItemsPerPage = PageSize,
                    TotalItems = category == null ?
                    db.Products.Count() :
                    db.Products.Where(e => e.Category == category).Count()
                },
                CurrentCategory = category
            };
            return View(model);
        }
    }
}
```

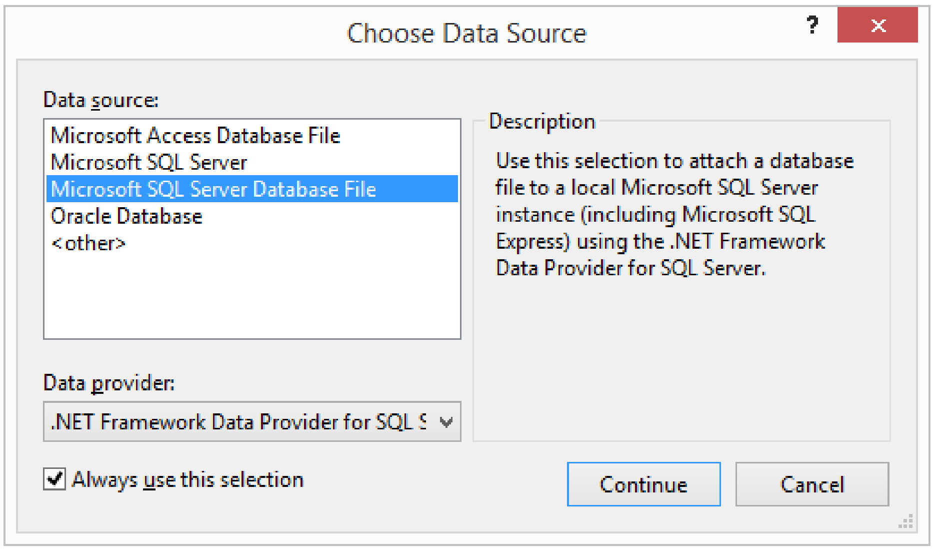
We have now changed the code to get the data from the `Products` DbSet instead of getting the data from the repository. When the `ToList()` method is called a database connection and SQL statement is created . This instantiates the `MbmStoreContext` class which generates DDL sql-statements and a MbmStoreInitializer object, and the `Seed` method is run which generates DML sql-statements. The result is a new database inside the *App_Data* folder with the name specified in the *web.config* file (*MbmStore1.mdf*).

Exercise 7, Inspect the database

Open the **Server Explorer** panel inside Visual Studio:



If there is not already a Data Connection, create a new one to the database. As data source, you must choose **Microsoft SQL Server Database File**:


Choose Data Source

Data source:

Microsoft Access Database File
Microsoft SQL Server
Microsoft SQL Server Database File
Oracle Database
<other>

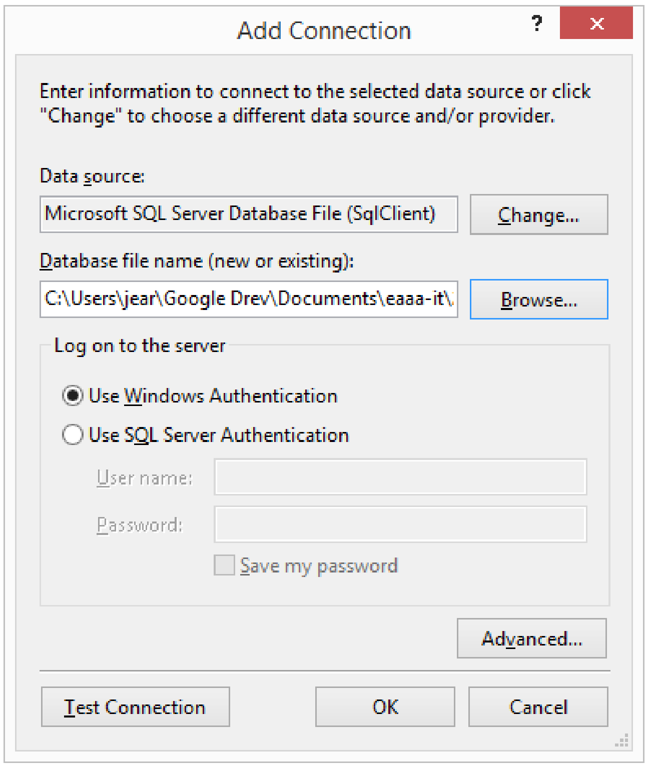
Description

Use this selection to attach a database file to a local Microsoft SQL Server instance (including Microsoft SQL Express) using the .NET Framework Data Provider for SQL Server.

Data provider: .NET Framework Data Provider for SQL S

[✓] Always use this selection

Pick the file and use Windows Authentication as login method:


Add Connection

Enter information to connect to the selected data source or click "Change" to choose a different data source and/or provider.

Data source: Microsoft SQL Server Database File (SqlClient)

Database file name (new or existing): C:\Users\jear\Google Drev\Documents\eaaa-it\

Log on to the server

● Use Windows Authentication

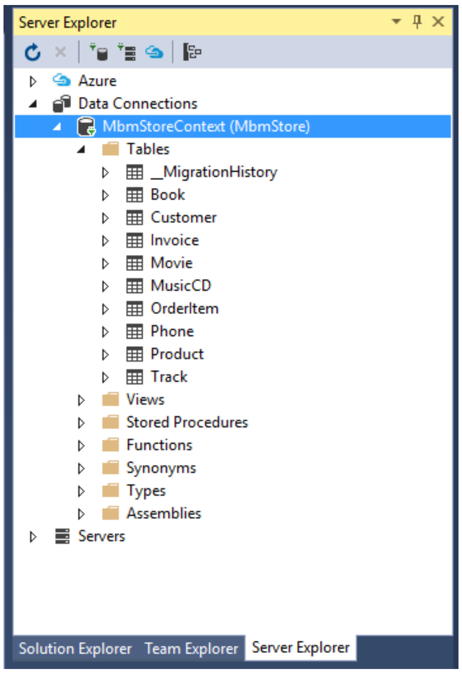
○ Use SQL Server Authentication

User name:

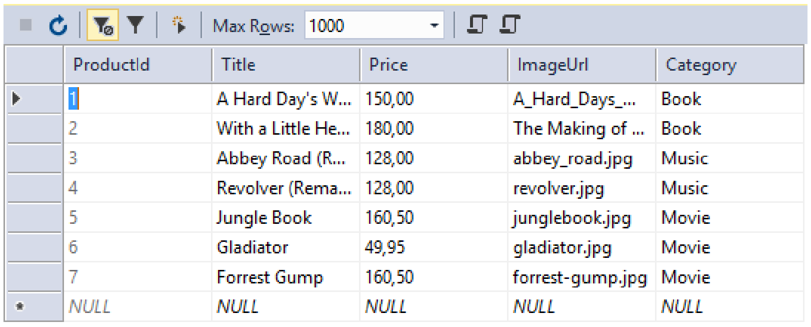
Password:

☐ Save my password

Use the Server Explorer to investigate the database (View -> Server Explorer or Ctrl+Alt+S):


```
Server Explorer
▷ Azure
◢ Data Connections
  ◢ MbmStoreContext (MbmStore)
    ◢ Tables
      ▷ _MigrationHistory
      ▷ Book
      ▷ Customer
      ▷ Invoice
      ▷ Movie
      ▷ MusicCD
      ▷ OrderItem
      ▷ Phone
      ▷ Product
      ▷ Track
    ▷ Views
    ▷ Stored Procedures
    ▷ Functions
    ▷ Synonyms
    ▷ Types
    ▷ Assemblies
▷ Servers
```

Right click at the product table and choose **Show Table Data**:


Max Rows: 1000

	ProductId	Title	Price	ImageUrl	Category
▶	1	A Hard Day's W...	150,00	A_Hard_Days_...	Book
	2	With a Little He...	180,00	The Making of ...	Book
	3	Abbey Road (R...	128,00	abbey_road.jpg	Music
	4	Revolver (Rema...	128,00	revolver.jpg	Music
	5	Jungle Book	160,50	junglebook.jpg	Movie
	6	Gladiator	49,95	gladiator.jpg	Movie
	7	Forrest Gump	160,50	forrest-gump.jpg	Movie
*	NULL	NULL	NULL	NULL	NULL

Can you add a new Customer to the database? Try it, if possible.

Exercise 8, Fetch all Data from the Database and Cleaning up

Open the `NavController` class file and update the class the retrieve data from the database

```
using MbmStore.DAL;
using System.Collections.Generic;
using System.Linq;
using System.Web.Mvc;

namespace MbmStore.Controllers
{
    public class NavController : Controller
    {
        private MbmStoreContext db;

        // constructor
        // instantiale a new db object
        public NavController()
        {
            db = new MbmStoreContext();
        }

        public PartialViewResult Menu(string category = null)
        {
            ViewBag.SelectedCategory = category;

            IEnumerable<string> categories = db.Products
            .Select(x => x.Category)
            .Distinct()
            .OrderBy(x => x);
            return PartialView(categories);
        }
    }
}
```

Do the same for `CartController` and `InvoiceController` classes.

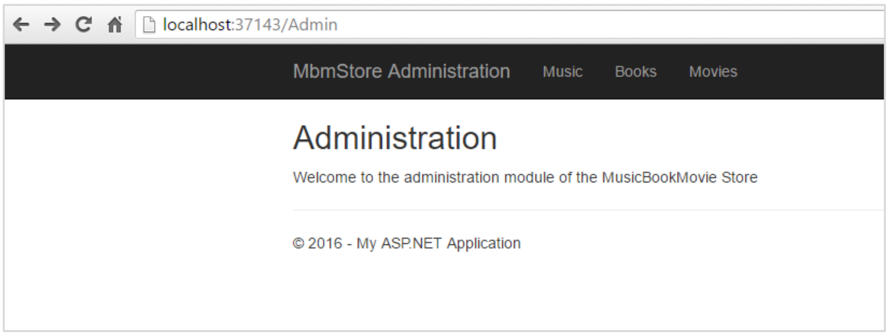
Run and the website (Ctrl-F5).

Delete the *Repository.cs* file in the *Infrastructure* folder.

Run and the website (Ctrl-F5). All dynamic data is now coming from the database.

Exercise 3

In this and the following exercises you'll start building an administration module for MusicBookMovie Store. Start by adding a new **Admin** area to the project and let the administration home page be similar to this screen dump:


localhost:37143/Admin

MbmStore Administration Music Books Movies

Administration

Welcome to the administration module of the MusicBookMovie Store

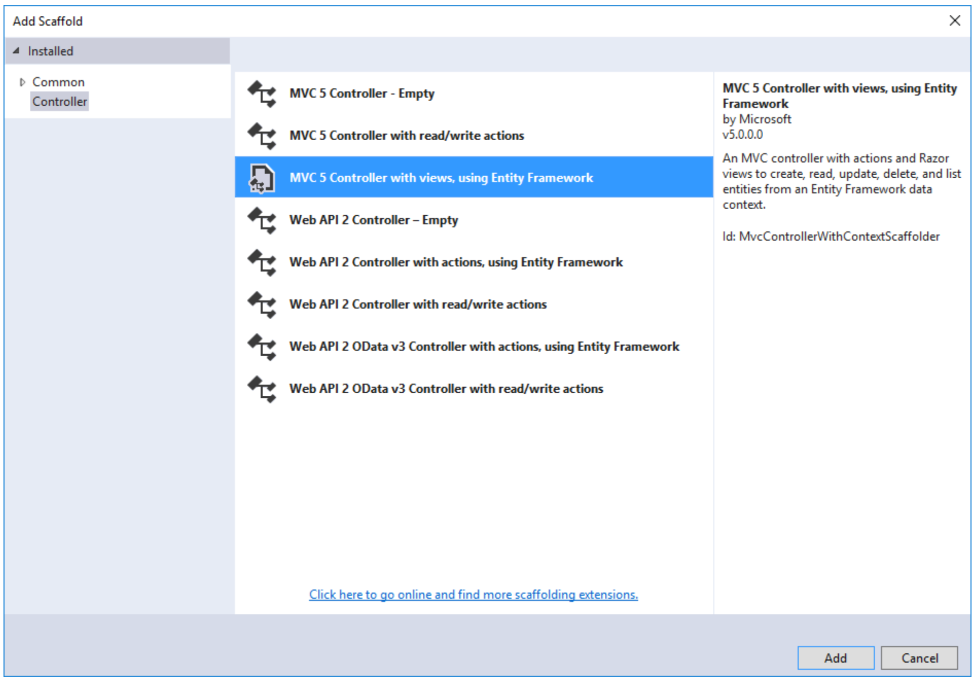
© 2016 - My ASP.NET Application

Music, **Books** and **Movies** are links pointing to the index method of a `Music`, Book and `Movie` Controller. The links of course will result in a *The resource cannot be found* error because we haven't yet created these controllers.

Creating a Music Controller and Views

To create the administration pages for the music, you'll begin by creating a new controller:

1. Right-click the **Areas\Admin\Controllers** folder in **Solution Explorer**, select **Add**, and then click **New Scaffolded Item**.
2. In the **Add Scaffold** dialog box, select **MVC 5 Controller with views, using Entity Framework**.


Add Scaffold

Installed
Common
Controller

MVC 5 Controller - Empty
MVC 5 Controller with read/write actions
MVC 5 Controller with views, using Entity Framework
Web API 2 Controller – Empty
Web API 2 Controller with actions, using Entity Framework
Web API 2 Controller with read/write actions
Web API 2 OData v3 Controller with actions, using Entity Framework
Web API 2 OData v3 Controller with read/write actions

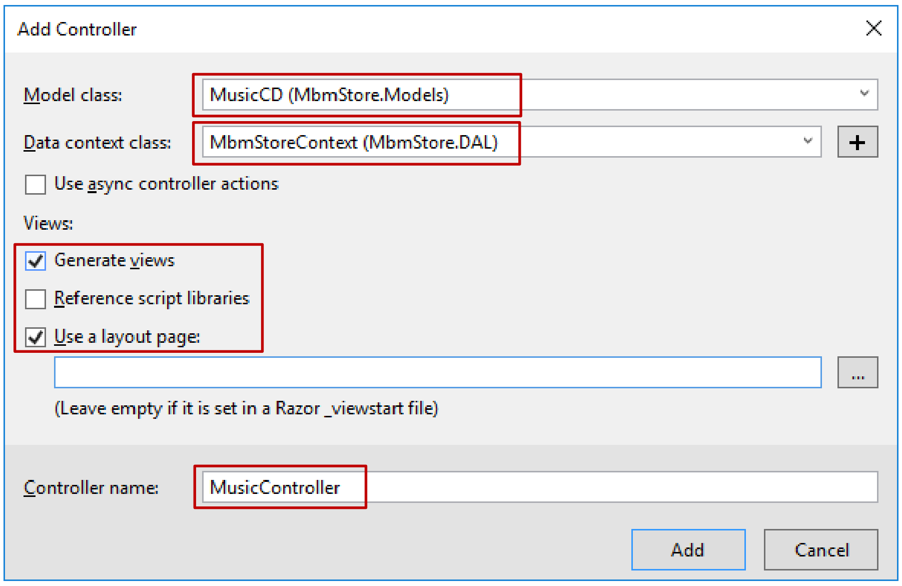
MVC 5 Controller with views, using Entity Framework
by Microsoft
v5.0.0.0
An MVC controller with actions and Razor views to create, read, update, delete, and list entities from an Entity Framework data context.
Id: MvcControllerWithContextScaffolder

Click here to go online and find more scaffolding extensions.

Add Cancel

3. In the Add Controller dialog box, make the following selections and then click **Add**:
   - Model class: **MusicCD (MbmStore.Models)**. (If you don't see this option in the drop-down list, build the project and try again.)

- Data context class: **MbmStoreContext (MbmStore.DAL)**.
- Use a layout page
- Controller name: **MusicController**.
- Leave the default values for the other fields.


Add Controller

Model class: MusicCD (MbmStore.Models)

Data context class: MbmStoreContext (MbmStore.DAL)

☐ Use async controller actions

Views:

☑ Generate views

☐ Reference script libraries

☑ Use a layout page:

(Leave empty if it is set in a Razor _viewstart file)

Controller name: MusicController

When you click **Add**, the scaffolder creates a *MusicController.cs* file and a set of views (.cshtml files) that work with the controller. In the future when you create projects that use EF you can also take advantage of some additional functionality of the scaffolder: just create your first model class, don't create a connection string, and then in the **Add Controller** box specify new context class. The scaffolder will create your `DbContext` class and your connection string as well as the controller and views.

4. Visual Studio opens the *Areas\Admin\Controllers\MusicController.cs* file. Before you continue replace all

   `db.Products` by `db.MusicCDs`.

   The scaffolder code is not optimized for class hierarchies and selects from the base class instead of the sub class.

5. You see a class variable has been created that instantiates a database context object:

```
private MbmStoreContext db = new MbmStoreContext();
```

The `Index` action method gets a list of music CDs from the `MusicCD` entity set by reading the `Students` property of the database context instance:

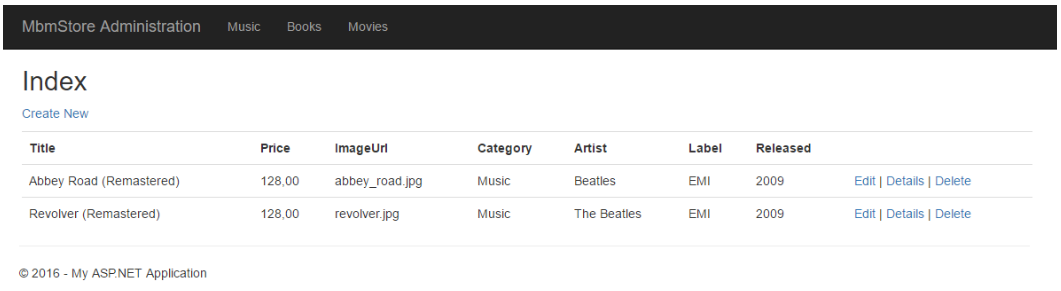
```
public ActionResult Index()
{
    return View(db.MusicCDs.ToList());
}
```

The *Areas\Admin\Music\Index.cshtml* view displays this list in a table:

```
<table class="table">
    <tr>
        <th>
            @Html.DisplayNameFor(model => model.Title)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.Price)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.ImageUrl)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.Category)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.Artist)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.Label)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.Released)
        </th>
        <th></th>
    </tr>

@foreach (var item in Model) {
    <tr>
        <td>
            @Html.DisplayFor(modelItem => item.Title)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Price)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.ImageUrl)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Category)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Artist)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Label)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Released)
```

```
                </td>
                <td>
                    @Html.ActionLink("Edit", "Edit", new { id=item.ProductId }) |
                    @Html.ActionLink("Details", "Details", new { id=item.ProductId
        }) |
                    @Html.ActionLink("Delete", "Delete", new { id=item.ProductId
        })
                </td>
            </tr>
        }

        </table>
```

6. Press CTRL+F5 to run the project. Click the Music tab to see the test data that the Seed method inserted.



Index

Create New

Title	Price	ImageUrl	Category	Artist	Label	Released	
Abbey Road (Remastered)	128,00	abbey_road.jpg	Music	Beatles	EMI	2009	Edit | Details | Delete
Revolver (Remastered)	128,00	revolver.jpg	Music	The Beatles	EMI	2009	Edit | Details | Delete

© 2016 - My ASP.NET Application

Conventions

The amount of code you had to write in order for the EF to be able to create a complete database for you is minimal because of the use of *conventions*, or assumptions that the EF makes. Some of them have already been noted or were used without your being aware of them:

- The pluralized forms of entity class names are used as table names.
- Entity property names are used for column names.
- Entity properties that are named `ID` or *classname*`ID` are recognized as primary key properties.
- A property is interpreted as a foreign key property if it's named *<navigation property name><primary key property name>*. Foreign key properties can also be named the same simply <primary key property name> (for example, `CustomerId` in `Invoice` since the `Customer` entity's primary key is `CustomerId`).

You've seen that conventions can be overridden. For example, you specified that table names shouldn't be pluralized. You'll learn more about conventions and how to override them in the next lesson. For more information about conventions, see Code First Conventions.

Exercise 9, Implementing Basic CRUD Functionality with the Entity Framework in ASP.NET MVC Application

In the previous exercise you created an MVC application that stores and displays data using the EF and SQL Server LocalDB. In this tutorial you'll review and customize the CRUD (create, read, update, delete) code that the MVC scaffolding automatically creates for you in controllers and views.

Note It's a common practice to implement the repository pattern in order to create an abstraction layer between your controller and the data access layer. To keep these tutorials simple and focused on teaching how to use the EF itself, they don't use repositories. For information about how to implement repositories, see the Using the Repository Pattern with ASP.NET MVC and Entity Framework.

Update the Create Page

In *Controllers\MusicController.cs*, replace the `HttpPost Create` action method with the following code to add a `try-catch` block and remove `ProductId` from the Bind attribute for the scaffolded method:

```
 [HttpPost]
        [ValidateAntiForgeryToken]
        public ActionResult Create([Bind(Include = "Title, Price, ImageUrl, Category,
Artist, Label, Released")] MusicCD musicCD)
        {
            try
            {
                if (ModelState.IsValid)
                {
                    db.MusicCDs.Add(musicCD);
                    db.SaveChanges();
                    return RedirectToAction("Index");
                }
            }
            catch (DataException /* dex */)
            {
                //Log the error (uncomment dex variable name and add a line here to write a
log.
                ModelState.AddModelError("", "Unable to save changes. Try again, and if the
problem persists see your system administrator.");
            }

           return View(musicCD);
       }
```

This code adds the `MusicCD` entity created by the ASP.NET MVC model binder to the `MusicCDs` entity set and then saves the changes to the database. (Model binder refers to the ASP.NET MVC functionality that makes it easier for you to work with data submitted by a form; a model binder converts posted form values to CLR types and passes them to the action method in parameters. In this case, the model binder instantiates a `MusicCD` entity for you using property values from the Form collection.)

You removed `ProductId` from the Bind attribute because `ProductId` is the primary key value which SQL Server will set automatically when the row is inserted. Input from the user does not set the `ProductId` value.

Security Note: The `ValidateAntiForgeryToken` attribute helps prevent cross-site request forgery attacks. It requires a corresponding `Html.AntiForgeryToken()` statement in the view.

The `Bind` attribute protects against over-posting. For example, suppose the `MusicCD` entity includes a Secret property that you don't want this web page to update.

It's a security best practice to use the `Include` parameter with the `Bind` attribute to *whitelist* fields. It's also possible to use the Exclude parameter to *blacklist* fields you want to exclude. The reason `Include` is more secure is that when you add a new property to the entity, the new field is not automatically protected by an `Exclude` list.

Another alternative approach, and one preferred by many, is to use only view models with model binding. The view model contains only the properties you want to bind. Once the MVC model binder has finished, you copy the view model properties to the entity instance.

Other than the `Bind` attribute, the `try-catch` block is the only change you've made to the scaffolded code. If an exception that derives from DataException is caught while the changes are being saved, a generic error message is displayed. DataException exceptions are sometimes caused by something external to the application rather than a programming error, so the user is advised to try again. Although not implemented in this sample, a production quality application would log the exception.

Update the Edit HttpPost Page

In *Controllers\MusicController.cs*, the `HttpGet Edit` method (the one without the `HttpPost` attribute) uses the `Find` method to retrieve the selected `MusicCD` entity. You don't need to change this method.

However, replace the `HttpPost Edit` action method with the following code to add a `try-catch` block:

```
[HttpPost]
        [ValidateAntiForgeryToken]
        public ActionResult Edit([Bind(Include =
"ProductId,Title,Price,ImageUrl,Category,Artist,Label,Released")] MusicCD musicCD)
        {
            try
            {
                if (ModelState.IsValid)
                {
                    db.Entry(musicCD).State = EntityState.Modified;
                    db.SaveChanges();
                    return RedirectToAction("Index");
                }
            }
            catch (DataException /* dex */)
            {
                //Log the error (uncomment dex variable name and add a line here to write a
log.
                ModelState.AddModelError("", "Unable to save changes. Try again, and if the
problem persists see your system administrator.");
            }
            return View(musicCD);
        }
```

This code is similar to what you saw in the `HttpPost Create` method. However, instead of adding the entity created by the model binder to the entity set, this code sets a flag on the entity indicating it has been changed. When the `SaveChanges` method is called, the `Modified` flag causes the EF to create SQL statements to update the database row. All columns of the database row will be updated, including those that the user didn't change, and concurrency conflicts are ignored.

Updating the Delete Page

In *Controllers\MusicController.cs*, the template code for the `HttpGet Delete` method uses the `Find` method to retrieve the selected `MusicCD` entity, as you saw in the `Details` and `Edit` methods. However, to implement a custom error message when the call to `SaveChanges` fails, you'll add some functionality to this method and its corresponding view.

As you saw for update and create operations, delete operations require two action methods. The method that is called in response to a GET request displays a view that gives the user a chance to approve or cancel the delete operation. If the user approves it, a POST request is created. When that happens, the `HttpPost Delete` method is called and then that method actually performs the delete operation.

You'll add a `try-catch` block to the `HttpPost Delete` method to handle any errors that might occur when the database is updated. If an error occurs, the `HttpPost Delete` method calls the `HttpGet Delete` method, passing it a parameter that indicates that an error has occurred. The `HttpGet Delete` method then redisplays the confirmation page along with the error message, giving the user an opportunity to cancel or try again.

1. Replace the `HttpGet Delete` action method with the following code, which manages error reporting:

```
public ActionResult Delete(int? id, bool? saveChangesError = false)
{
     if (id == null)
     {
         return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
     }

     if (saveChangesError.GetValueOrDefault())
     {
         ViewBag.ErrorMessage = "Delete failed. Try again, and if the problem
persists see your system administrator.";
     }

     MusicCD musicCD = db.MusicCDs.Find(id);
     if (musicCD == null)
     {
         return HttpNotFound();
     }
         return View(musicCD);
      }
```

   This code accepts an optional parameter that indicates whether the method was called after a failure to save changes. This parameter is `false` when the `HttpGet Delete` method is called without a previous failure. When it is called by the `HttpPost Delete` method in response to a database update error, the parameter is `true` and an error message is passed to the view.

2. Replace the HttpPost Delete action method (named DeleteConfirmed) with the following code, which performs the actual delete operation and catches any database update errors.

```
[HttpPost, ActionName("Delete")]
```

```
 [ValidateAntiForgeryToken]
 public ActionResult Delete(int id)
 {
     try
     {
         MusicCD musicCD = db.MusicCDs.Find(id);
     db.MusicCDs.Remove(musicCD);
     db.SaveChanges();
     }
     catch (DataException/* dex */)
     {
         //Log the error (uncomment dex variable name and add a line here to write a
log.
         return RedirectToAction("Delete", new { id = id, saveChangesError = true });
     }
     return RedirectToAction("Index");
}
```

This code retrieves the selected entity, then calls the Remove method to set the entity's status to `Deleted`. When `SaveChanges` is called, a SQL `DELETE` command is generated. You have also changed the action method name from `DeleteConfirmed` to `Delete`. The scaffolded code named the `HttpPost Delete` method `DeleteConfirmed` to give the `HttpPost` method a unique signature. (The CLR requires overloaded methods to have different method parameters). Now that the signatures are unique, you can stick with the MVC convention and use the same name for the `HttpPost` and `HttpGet` delete methods.

If improving performance in a high-volume application is a priority, you could avoid an unnecessary SQL query to retrieve the row by replacing the lines of code that call the `Find` and `Remove` methods with the following code:

```
MusicCD MusicCDToDelete = new MusicCD() { ProductId = id };
db.Entry(MusicCDToDelete).State = EntityState.Deleted;
```

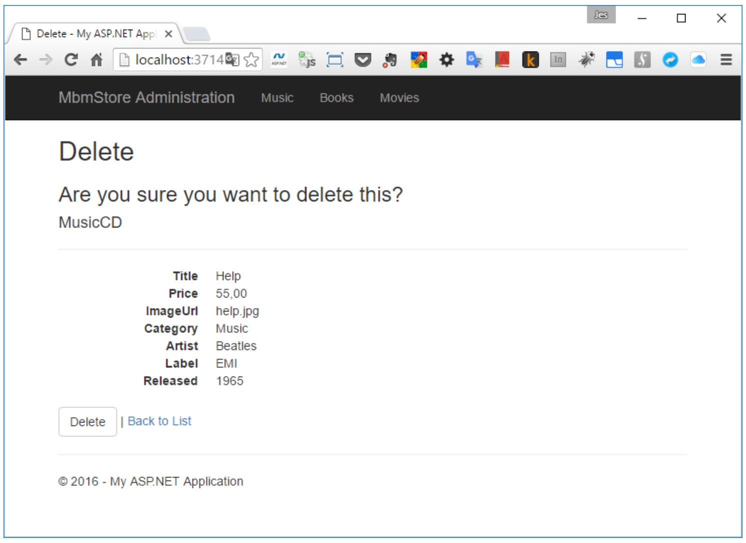
This code instantiates a `Student` entity using only the primary key value and then sets the entity state to `Deleted`. That's all that the EF needs in order to delete the entity.

As noted, the `HttpGet Delete` method doesn't delete the data. Performing a delete operation in response to a GET request (or for that matter, performing any edit operation, create operation, or any other operation that changes data) creates a security risk. For more information, see ASP.NET MVC Tip #46 — Don't use Delete Links because they create Security Holes on Stephen Walther's blog.

3. In *Views\Music\Delete.cshtml*, add an error message between the `h2` heading and the `h3` heading, as shown in the following example:

```
<h2>Delete</h2>
<p class="error">@ViewBag.ErrorMessage</p>
<h3>Are you sure you want to delete this?</h3>
```

Run the page by selecting the **Music** tab and clicking a **Delete** hyperlink:



Delete

Are you sure you want to delete this?

MusicCD

Title	Help
Price	55,00
ImageUrl	help.jpg
Category	Music
Artist	Beatles
Label	EMI
Released	1965

Delete | Back to List

© 2016 - My ASP.NET Application

4. Click **Delete**. The Index page is displayed without the deleted student.

Ensuring that Database Connections Are Not Left Open

To make sure that database connections are properly closed and the resources they hold freed up, you have to dispose the context instance when you are done with it. That is why the scaffolded code provides a Dispose method at the end of the `MusicController` class in *MusicController.cs*, as shown in the following example

```
protected override void Dispose(bool disposing)
{
    if (disposing)
    {
        db.Dispose();
    }
    base.Dispose(disposing);
}
```

The base `Controller` class already implements the `IDisposable` interface, so this code simply adds an override to the `Dispose(bool)` method to explicitly dispose the context instance.

Exercise 10, More CRUD

Implement the same CRUD functionality for books and movies.