

Lesson 9

As starting point for these exercises, you can continue working on your own project from the last lesson or you can download and use Lesson08_MbmStore_solution.zip from the [Documents > Examples and solutions](#) folder.

The first 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, Enable Code First Migrations

When you develop a new application, your data model changes frequently, and each time the model changes, it gets out of sync with the database. You have configured the EF to automatically drop and re-create the database each time you change the data model in the MbmStoreInitializer class.

```
public class MbmStoreInitializer :  
System.Data.Entity.DropCreateDatabaseIfModelChanges<MbmStoreContext> { ... }
```

When you add, remove, or change entity classes or change your MbmStoreContext class, the next time you run the application it automatically deletes your existing database, creates a new one that matches the model, and seeds it with test data.

This method of keeping the database in sync with the data model works well until you deploy the application to production. When the application is running in production it is usually storing data that you want to keep, and you don't want to lose everything each time you make a change such as adding a new column. The [Code First Migrations](#) feature solves this problem by enabling Code First to update the database schema instead of dropping and re-creating the database. In this exercise, you'll deploy the application, and to prepare for that you'll enable Migrations.

1. Disable the initializer that you set up earlier by commenting out or deleting the DatabaseInitializerForType key that you added to the application *Web.config* file.

```
<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>
```

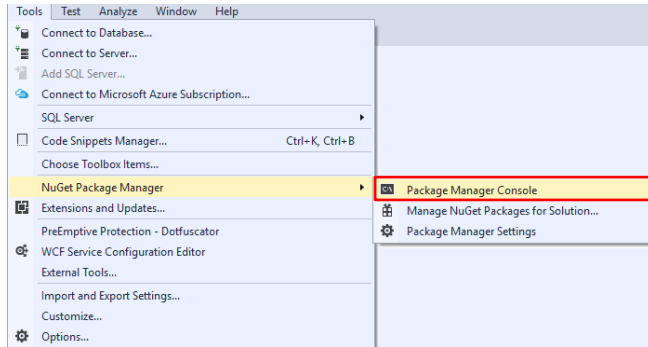
2. Also in the application *Web.config* file, change the name of the database in the connection string to *MbmStore2.mdf*.

```
<connectionStrings>  
  <add name="MbmStoreContext" connectionString="Data Source=(LocalDB)\MSSQLLocalDB;  
Integrated Security=True; MultipleActiveResultSets=True;  
AttachDbFilename=|DataDirectory|MbmStore2.mdf"  
providerName="System.Data.SqlClient" />
```

```
</connectionStrings>
```

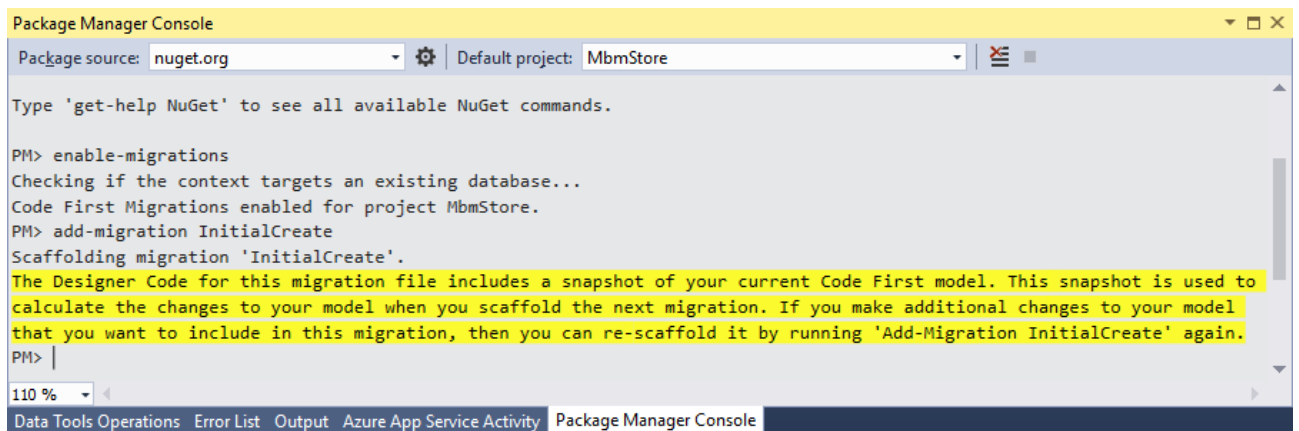
This change sets up the project so that the first migration will create a new database. This isn't required but you'll see later why it's a good idea.

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



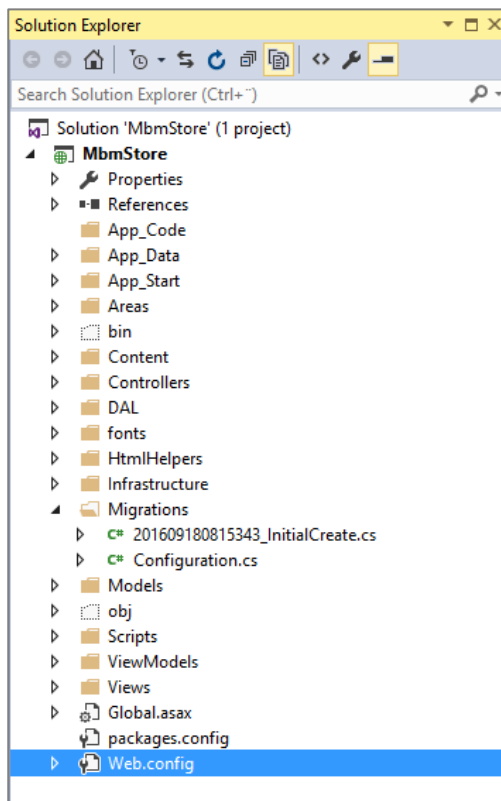
4. At the PM> prompt enter the following commands – one at a time:

```
enable-migrations
add-migration InitialCreate
```



The `enable-migrations` command creates a *Migrations* folder in the *MbmStore* project, and it puts in that folder a *Configuration.cs* file that you can edit to configure Migrations.

(If you missed the step above that directs you to change the database name, Migrations will find the existing database and automatically do the `add-migration` command. That's OK, it just means you won't run a test of the migrations code before you deploy the database. Later when you run the `update-database` command nothing will happen because the database will already exist.)



Like the initializer class that you saw earlier, the `Configuration` class includes a `Seed` method.

```
internal sealed class Configuration :
    DbMigrationsConfiguration<MbmStore.DAL.MbmStoreContext>
{
    public Configuration()
    {
        AutomaticMigrationsEnabled = false;
    }

    protected override void Seed(MbmStore.DAL.MbmStoreContext context)
    {
        // This method will be called after migrating to the latest version.

        // You can use the DbSet<T>.AddOrUpdate() helper extension method
        // to avoid creating duplicate seed data. E.g.
        //
        // context.People.AddOrUpdate(
        //     p => p.FullName,
        //     new Person { FullName = "Andrew Peters" },
        //     new Person { FullName = "Brice Lambson" },
        //     new Person { FullName = "Rowan Miller" }
        // );
        //
    }
}
```

The purpose of the `Seed` method is to enable you to insert or update test data after Code First creates or updates the database. The method is called when the database is created and every time the database

schema is updated after a data model change.

Set up the Seed Method

When you are dropping and re-creating the database for every data model change, you use the initializer class's `Seed` method to insert test data, because after every model change the database is dropped and all the test data is lost. With Code First Migrations, test data is retained after database changes, so including test data in the `Seed` method is typically not necessary. In fact, you don't want the `Seed` method to insert test data if you'll be using Migrations to deploy the database to production, because the `Seed` method will run in production. In that case you want the `Seed` method to insert into the database only the data that you need in production.

For this exercise, you'll be using Migrations for production environment, but your `Seed` method will insert test data anyway in order to make it easier to see how application functionality works without having to manually insert a lot of data.

1. Replace the contents of the *Configuration.cs* file with the following code, which will load test data into the new database.

```
namespace MbmStore.Migrations
{
    using Models;
    using System;
    using System.Collections.Generic;
    using System.Data.Entity.Migrations;

    internal sealed class Configuration :
        DbMigrationsConfiguration<MbmStore.DAL.MbmStoreContext>
    {
        public Configuration()
        {
            AutomaticMigrationsEnabled = false;
        }

        protected override void Seed(MbmStore.DAL.MbmStoreContext 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="Pettersen", 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. The code is the exact same code we used before in the `MbmStoreInitializer` class. 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.

Execute the First Migration

When you executed the `add-migration` command, Migrations generated the code that would create the database from scratch. This code is also in the `Migrations` folder, in the file named `<timestamp>_InitialCreate.cs`. The `Up` method of the `InitialCreate` class creates the database tables that correspond to the data model entity sets, and the `Down` method deletes them.

```
public partial class InitialCreate : DbMigration
{
    public override void Up()
    {
        CreateTable(
            "dbo.Product",
            c => new
            {
                ProductId = c.Int(nullable: false, identity: true),
                Title = c.String(),
                Price = c.Decimal(nullable: false, precision: 18, scale: 2),
                ImageUrl = c.String(),
                Category = c.String(),
            })
            .PrimaryKey(t => t.ProductId);

        CreateTable(
            "dbo.Customer",
            c => new
            {
                CustomerId = c.Int(nullable: false, identity: true),
                Firstname = c.String(),
                Lastname = c.String(),
                Address = c.String(),
                Zip = c.String(),
                City = c.String(),
                Email = c.String(),
                Birthdate = c.DateTime(nullable: false, precision: 7, storeType:
"datetime2"),
            })
            .PrimaryKey(t => t.CustomerId);

        CreateTable(
            "dbo.Invoice",
            c => new
            {
                InvoiceId = c.Int(nullable: false, identity: true),
```

```

        OrderDate = c.DateTime(nullable: false, precision: 7, storeType:
"datetime2"),
        CustomerId = c.Int(nullable: false),
    })
    .PrimaryKey(t => t.InvoiceId)
    .ForeignKey("dbo.Customer", t => t.CustomerId, cascadeDelete: true)
    .Index(t => t.CustomerId);

CreateTable(
    "dbo.OrderItem",
    c => new
    {
        OrderItemId = c.Int(nullable: false, identity: true),
        ProductId = c.Int(nullable: false),
        InvoiceId = c.Int(nullable: false),
        Quantity = c.Int(nullable: false),
    })
    .PrimaryKey(t => t.OrderItemId)
    .ForeignKey("dbo.Product", t => t.ProductId, cascadeDelete: true)
    .ForeignKey("dbo.Invoice", t => t.InvoiceId, cascadeDelete: true)
    .Index(t => t.ProductId)
    .Index(t => t.InvoiceId);

CreateTable(
    "dbo.Track",
    c => new
    {
        TrackId = c.Int(nullable: false, identity: true),
        Title = c.String(),
        Composer = c.String(),
        Length = c.Time(nullable: false, precision: 7),
        MusicCD_ProductId = c.Int(),
    })
    .PrimaryKey(t => t.TrackId)
    .ForeignKey("dbo.MusicCD", t => t.MusicCD_ProductId)
    .Index(t => t.MusicCD_ProductId);

CreateTable(
    "dbo.Phone",
    c => new
    {
        PhoneId = c.Int(nullable: false, identity: true),
        Number = c.String(),
        CustomerId = c.Int(nullable: false),
    })
    .PrimaryKey(t => t.PhoneId)
    .ForeignKey("dbo.Customer", t => t.CustomerId, cascadeDelete: true)
    .Index(t => t.CustomerId);

CreateTable(
    "dbo.Book",
    c => new
    {
        ProductId = c.Int(nullable: false),
        Author = c.String(),
        Publisher = c.String(),
        Published = c.Short(nullable: false),
        ISBN = c.String(),
        tmp = c.String(),
    })

```



```

        })
        .PrimaryKey(t => t.ProductId)
        .ForeignKey("dbo.Product", t => t.ProductId)
        .Index(t => t.ProductId);

CreateTable(
    "dbo.Movie",
    c => new
    {
        ProductId = c.Int(nullable: false),
        Director = c.String(),
    })
    .PrimaryKey(t => t.ProductId)
    .ForeignKey("dbo.Product", t => t.ProductId)
    .Index(t => t.ProductId);

CreateTable(
    "dbo.MusicCD",
    c => new
    {
        ProductId = c.Int(nullable: false),
        Artist = c.String(),
        Label = c.String(),
        Released = c.Short(nullable: false),
    })
    .PrimaryKey(t => t.ProductId)
    .ForeignKey("dbo.Product", t => t.ProductId)
    .Index(t => t.ProductId);
}

public override void Down()
{
    DropForeignKey("dbo.MusicCD", "ProductId", "dbo.Product");
    DropForeignKey("dbo.Movie", "ProductId", "dbo.Product");
    DropForeignKey("dbo.Book", "ProductId", "dbo.Product");
    DropForeignKey("dbo.Phone", "CustomerId", "dbo.Customer");
    DropForeignKey("dbo.OrderItem", "InvoiceId", "dbo.Invoice");
    DropForeignKey("dbo.OrderItem", "ProductId", "dbo.Product");
    DropForeignKey("dbo.Track", "MusicCD_ProductId", "dbo.MusicCD");
    DropForeignKey("dbo.Invoice", "CustomerId", "dbo.Customer");
    DropIndex("dbo.MusicCD", new[] { "ProductId" });
    DropIndex("dbo.Movie", new[] { "ProductId" });
    DropIndex("dbo.Book", new[] { "ProductId" });
    DropIndex("dbo.Phone", new[] { "CustomerId" });
    DropIndex("dbo.Track", new[] { "MusicCD_ProductId" });
    DropIndex("dbo.OrderItem", new[] { "InvoiceId" });
    DropIndex("dbo.OrderItem", new[] { "ProductId" });
    DropIndex("dbo.Invoice", new[] { "CustomerId" });
    DropTable("dbo.MusicCD");
    DropTable("dbo.Movie");
    DropTable("dbo.Book");
    DropTable("dbo.Phone");
    DropTable("dbo.Track");
    DropTable("dbo.OrderItem");
    DropTable("dbo.Invoice");
    DropTable("dbo.Customer");
    DropTable("dbo.Product");
}

```

```
}
```

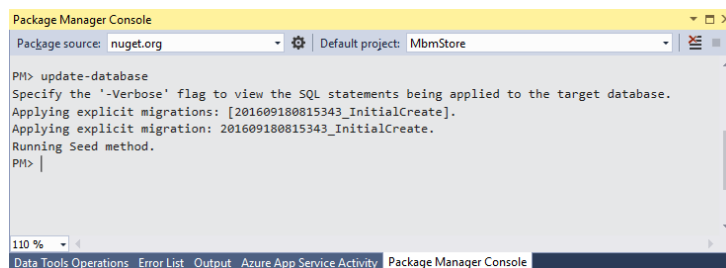
Migrations calls the Up method to implement the data model changes for a migration. When you enter a command to roll back the update, Migrations calls the Down method.

This is the initial migration that was created when you entered the `add-migration InitialCreate` command. The parameter (`InitialCreate` in the example) is used for the file name and can be whatever you want; you typically choose a word or phrase that summarizes what is being done in the migration. For example, you might name a later migration "AddCountryTable" for storing a full list of countries.

If you created the initial migration when the database already exists, the database creation code is generated but it doesn't have to run because the database already matches the data model. When you deploy the app to another environment where the database doesn't exist yet, this code will run to create your database, so it's a good idea to test it first. That's why you changed the name of the database in the connection string earlier -- so that migrations can create a new one from scratch.

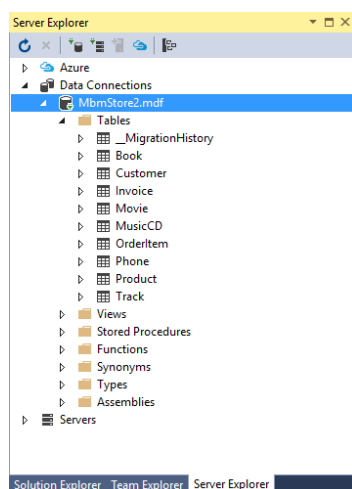
1. In the **Package Manager Console** window, enter the following command:

```
update-database
```



The `update-database` command runs the Up method to create the database and then it runs the Seed method to populate the database.

2. Use **Server Explorer** to create a new connection and inspect the database as you did in the last exercise, and run the application to verify that everything still works the same as before.



Change the MbmStoreInitializer class inheritance from:

```
public class MbmStoreInitializer :
System.Data.Entity.DropCreateDatabaseIfModelChanges<MbmStoreContext> { ... }
```

to:

```
public class MbmStoreInitializer :
System.Data.Entity.CreateDatabaseIfNotExists<MbmStoreContext> { ... }
```

This prevents the database from being re-created every time the model changes. Also, with example data in the database, you *should comment out the content of the seed method* in Migrations/Configuration.cs as there is apparently no way of preventing the seed method from running you run the update-database command.

Exercise 2, Changing the model by adding a CreatedDate property to Product

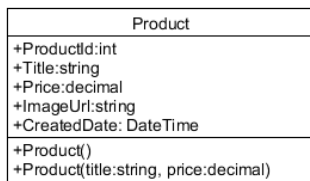
We'll now add a new DateTime property CreatedDate to the Product class. We'll need that if we later want to display the newest products in each categories.

To do that, follow these steps:

1. Open the Product class (Models\Product.cs) and add a new CreatedDate property:

```
public DateTime CreatedDate { get; set; }
```

and update the UML diagram accordingly:



2. Open the Package manager and execute this command:

```
PM> add-migration "ProductCreatedDateAttribute"
```

The command returns this message:

The Designer Code for this migration file includes a snapshot of your current Code First model. This snapshot is used to calculate the changes to your model when you scaffold the next migration. If you make additional changes to your model that you want to include in this migration, then you can re-scaffold it by running 'Add-Migration ProductCreatedDateAttribute' again.

The message tells that the migration file has the code that must be sent to the database in order to change the database in accordance with the altered model. If you want to include more model changes in this migration, you can run add-migration "ProductCreatedDateAttribute" once more. To make things

clear and simple though, you normally want one migration for each model change.

Like the `InitialCreate` the migration creates a `Up` method for executing the changes and a `Down` method which can be used for roll-back:

```
public partial class ProductCreateDateAttribute : DbMigration
{
    public override void Up()
    {
        AddColumn("dbo.Product", "CreateDate", c => c.DateTime(nullable:
false));
    }

    public override void Down()
    {
        DropColumn("dbo.Product", "CreateDate");
    }
}
```

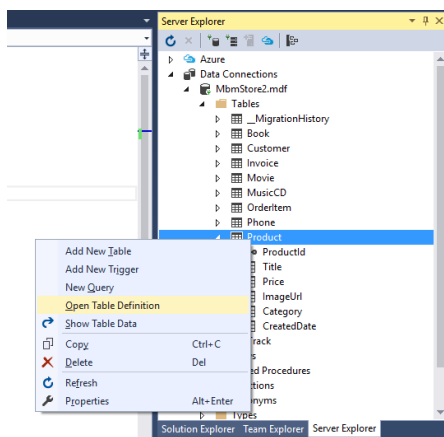
When we insert new product into the database, we want the database to insert the actual time as default value. In order to accomplish that you must modify the `Up` method to set a default value:

```
public override void Up()
{
    AddColumn("dbo.Product", "CreateDate", c => c.DateTime(nullable: true,
defaultValueSql: "GETUTCDATE()"));
}
```

3. Open the Package manager and execute this command:

```
PM> update-database
```

4. Open the server explorer panel in Visual Studio. Connect to the database and inspect the **Table Definition** window of the **Product** table by right clicking the **Product** table and click at **Open Table Definition**:



Check to see that the `CreateDate` column is added, and Coordinated Universal Time is inserted as default value:

dbo.Product [Design] ✕

Update Script File: dbo.Product.sql

Name	Data Type	Allow Nulls	Default
ProductId	int	<input type="checkbox"/>	
Title	nvarchar(MAX)	<input checked="" type="checkbox"/>	
Price	decimal(18,2)	<input type="checkbox"/>	
ImageUrl	nvarchar(MAX)	<input checked="" type="checkbox"/>	
Category	nvarchar(MAX)	<input checked="" type="checkbox"/>	
CreatedDate	datetime	<input checked="" type="checkbox"/>	(getutcdate())

5. On the **Show Data Table** window for the Product table and insert data in the CreatedDate column:

dbo.MusicCD [Data] **dbo.Product [Data]** ✕ dbo.Product [Design]

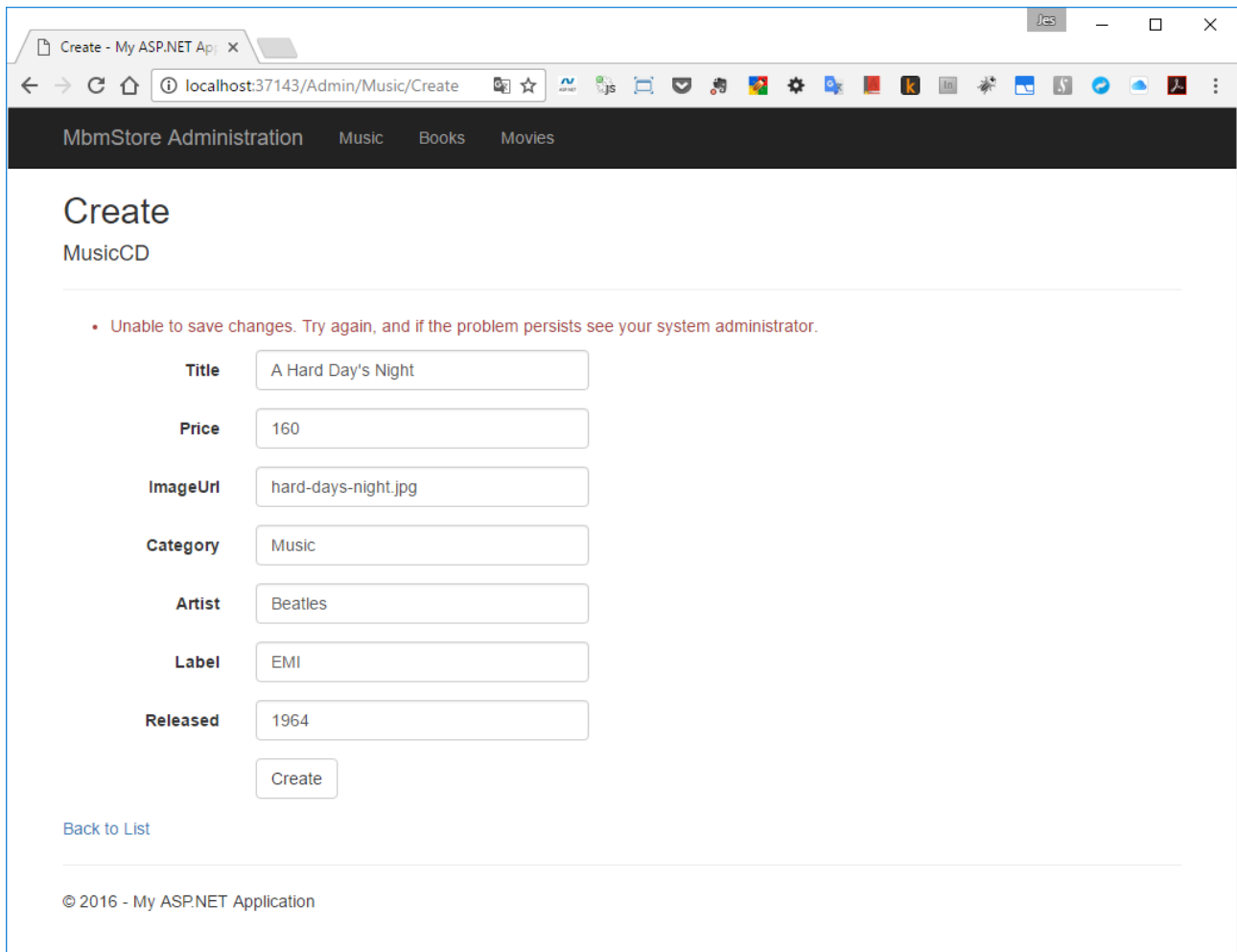
Max Rows: 1000

	ProductId	Title	Price	ImageUrl	Category	CreatedDate
1		A Hard Day's W...	150,00	A_Hard_Days_...	Book	10-10-2016 00:00:00
2		With a Little He...	180,00	The Making of ...	Book	10-10-2016 00:00:00
3		Abbey Road (R...	128,00	abbey_road.jpg	Music	10-10-2016 00:00:00
4		Revolver (Rema...	128,00	revolver.jpg	Music	10-10-2016 00:00:00
5		Jungle Book	160,50	junglebook.jpg	Movie	10-10-2016 00:00:00
6		Gladiator	49,95	gladiator.jpg	Movie	10-10-2016 00:00:00
7		Forrest Gump	160,50	forrest-gump.jpg	Movie	10-10-2016 00:00:00
➤	NULL	NULL	NULL	NULL	NULL	NULL

The is needed because CreatedDate property in the Product class cannot be null.

Exercise 3, correct errors by inspecting SQL

Open the **Music** administration page and create a new MusicCD. This will give you this error message:



The screenshot shows a web browser window with the address bar at `localhost:37143/Admin/Music/Create`. The page has a navigation bar with links: **MbmStore Administration**, **Music**, **Books**, and **Movies**. The main heading is **Create MusicCD**. Below the heading, a red error message states: "Unable to save changes. Try again, and if the problem persists see your system administrator." Below the error message is a form with the following fields: **Title** (A Hard Day's Night), **Price** (160), **ImageUrl** (hard-days-night.jpg), **Category** (Music), **Artist** (Beatles), **Label** (EMI), and **Released** (1964). There is a **Create** button at the bottom of the form. A [Back to List](#) link is located below the form. At the bottom of the page, the footer reads "© 2016 - My ASP.NET Application".

The error message comes from the catch block in Create method of the Music controller:

```
[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);
}

```

Because we wrote this try-catch block the error is caught and we do not see the runtime error. That's a good thing for the end user – because it is user friendly – but it is not very informative for you as a programmer because you do not see the error that is returned from the database and without that it is very hard to discover the error.

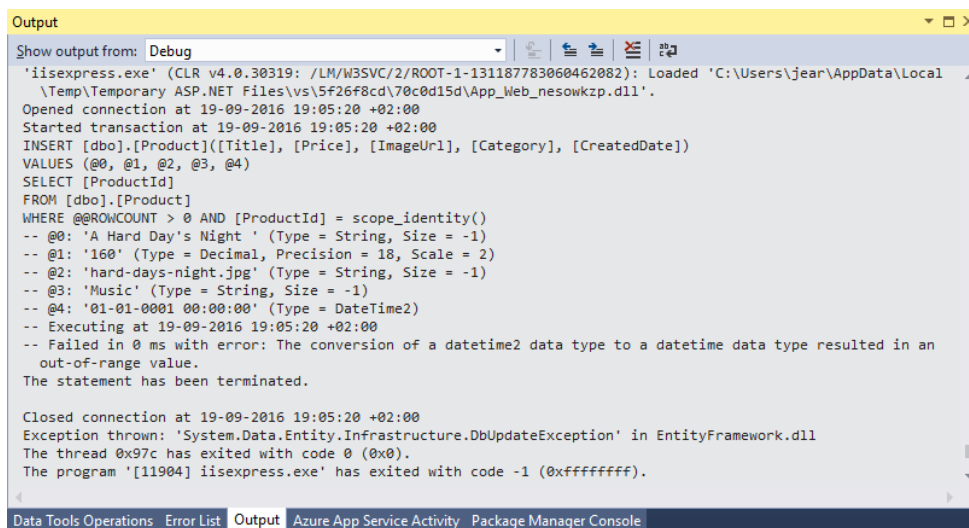
A solution is to log all SQL-statements in debug-mode inside the output window. Open the MbmStoreContext class (DAL\ MbmStoreContext.cs) and add this line to the constructor:

```

public MbmStoreContext() : base("MbmStoreContext")
{
    Database.Log = sql => Debug.WriteLine(sql);
}

```

Load the administration panel in debug mode and try once more to create a new MusicCD. It'll give you the same error message as before. Now, go to Visual Studio and open the **Output window**:



```

Output
Show output from: Debug
'iisexpress.exe' (CLR v4.0.30319: /LM/W3SVC/2/ROOT-1-131187783060462082): Loaded 'C:\Users\jear\AppData\Local
\Temp\Temporary ASP.NET Files\vs\5f26f8cd\70c0d15d\App_Web_nesowkzp.dll'.
Opened connection at 19-09-2016 19:05:20 +02:00
Started transaction at 19-09-2016 19:05:20 +02:00
INSERT [dbo].[Product]([Title], [Price], [ImageUrl], [Category], [CreatedDate])
VALUES (@0, @1, @2, @3, @4)
SELECT [ProductId]
FROM [dbo].[Product]
WHERE @@ROWCOUNT > 0 AND [ProductId] = scope_identity()
-- @0: 'A Hard Day's Night ' (Type = String, Size = -1)
-- @1: '160' (Type = Decimal, Precision = 18, Scale = 2)
-- @2: 'hard-days-night.jpg' (Type = String, Size = -1)
-- @3: 'Music' (Type = String, Size = -1)
-- @4: '01-01-0001 00:00:00' (Type = DateTime2)
-- Executing at 19-09-2016 19:05:20 +02:00
-- Failed in 0 ms with error: The conversion of a datetime2 data type to a datetime data type resulted in an
out-of-range value.
The statement has been terminated.

Closed connection at 19-09-2016 19:05:20 +02:00
Exception thrown: 'System.Data.Entity.Infrastructure.DbUpdateException' in EntityFramework.dll
The thread 0x97c has exited with code 0 (0x0).
The program '[11904] iisexpress.exe' has exited with code -1 (0xffffffff).

```

In the output window, you see the SQL insert statement sent to the server from the EF and you see the error message returned from the database server:

Failed in 0 ms with error: The conversion of a datetime2 data type to a datetime data type resulted in an out-of-range value.
The statement has been terminated.

This is much more informative and the error happens because EF will add a default date for the field with the value {01/01/0001 00:00:00}, which is outside the range of SQL Date, which has the range: January 1, 1753, through December 31, 9999.

To fix that, you can assign the current date to the `CreatedDate` property of the `MusicCD` object before you add the `musicCD` object to the `MbmStoreContext` class inside the `Music Controller`:

```
try
{
    if (ModelState.IsValid)
    {
        musicCD.CreatedDate = DateTime.Now;
        db.MusicCDs.Add(musicCD);
        db.SaveChanges();
        return RedirectToAction("Index");
    }
}
```

This solves the problem, and can now store new `MusicCD`'s in the database.

To enable updates, you must prevent the `CreatedDate` property from being updated. You can do that by setting the `IsModified` property of the `CreatedDate` entity object to `false` inside the `HttpPost` version of the `Edit` action method:

```
try
{
    if (ModelState.IsValid)
    {
        db.Entry(musicCD).State = EntityState.Modified;
        db.Entry(musicCD).Property(c => c.CreatedDate).IsModified = false;
        db.SaveChanges();
        return RedirectToAction("Index");
    }
}
```

Make the same two corrections to the `Book` and `Movie` controller.

Exercise 4, Process order

To complete orders we need to code order processing logic into the `Checkout` action method of the `Cart` controller:

```
[HttpPost]
public ViewResult Checkout(Cart cart, ShippingDetails shippingDetails)
{
    if (cart.Lines.Count() == 0)
    {
        ModelState.AddModelError("", "Sorry, your cart is empty!");
    }

    if (ModelState.IsValid)
    {
        // order processing logic
        cart.Clear();
        return View("Completed");
    }
}
```



```

    }
    else
    {
        return View(shippingDetails);
    }
}

```

The first thing we need to do is inserting new customers

When customers submit a new reservation, it results in creation of a new Customer object based on data from the shippingDetails parameter:

```

[HttpPost]
public ViewResult Checkout(Cart cart, ShippingDetails shippingDetails)
{
    if (cart.Lines.Count() == 0)
    {
        ModelState.AddModelError("", "Sorry, your cart is empty!");
    }
    if (ModelState.IsValid)
    {
        Customer customer = new Customer
        {
            Firstname = shippingDetails.Firstname,
            Lastname = shippingDetails.Lastname,
            Address = shippingDetails.Address,
            Zip = shippingDetails.Zip,
            Email = shippingDetails.Email
        };

        Invoice invoice = new Invoice(DateTime.Now, customer);
        ...
        return View("Completed");
    }
    else
    {
        return View(shippingDetails);
    }
}

```

There is a problem though. For each order a new Customer object is created and data is inserted as a new row in the Customer table. That means that returning customers will be created as new customers for each new order. To avoid duplicates, we must check and see if the customer already exists in the database. We'll use the LINQ method Any that returns true or false depending on the outcome of the query. To ensure that each customer is a unique customer, we'll set up the rule, that customers must have a unique combination of firstname + lastname + email.

To see if a customer exists, we write a query for that customer with the desired filtering. If the customer exists, we can select that customer, update the Address and Zip properties, and change its EF state to Modified. By doing that, the EF will do an update instead of an insert.

You'll see a simplified filtering example here:

```

[HttpPost]

```

```

public ActionResult Checkout(Cart cart, ShippingDetails shippingDetails)
{
    if (cart.Lines.Count() == 0)
    {
        ModelState.AddModelError("", "Sorry, your cart is empty!");
    }
    if (ModelState.IsValid)
    {
        Customer customer = new Customer
        {
            Firstname = shippingDetails.Firstname,
            Lastname = shippingDetails.Lastname,
            Address = shippingDetails.Address,
            Zip = shippingDetails.Zip,
            Email = shippingDetails.Email
        };

        if (db.Customers.Any(c => c.Firstname == customer.Firstname && c.Lastname ==
customer.Lastname && c.Email == customer.Email))
        {
            customer = db.Customers.Where(c => c.Firstname == customer.Firstname &&
c.Lastname == customer.Lastname && c.Email == customer.Email).First();
            customer.Address = shippingDetails.Address;
            customer.Zip = shippingDetails.Zip;
            // ensure update instead of insert
            db.Entry(customer).State = EntityState.Modified;
        }

        Invoice invoice = new Invoice(DateTime.Now, customer);
        ...

        cart.Clear();
        return View("Completed");
    }
    else
    {
        return View(shippingDetails);
    }
}

```

Before you continue coding the Checkout action method, you must edit the OrderItem class to tell EF that the Product property is a navigation property:

```

public class OrderItem
{
    public int OrderItemId { get; set; }
    public int ProductId { get; set; }
    public int InvoiceId { get; set; }
    public int Quantity { get; set; }
    public decimal TotalPrice { get { return Quantity * Product.Price; } }

    // navigation property
    public virtual Product Product { get; set; }

    public OrderItem() { }
}

```

```

public OrderItem(Product product, int quantity)
{
    Product = product;
    Quantity = quantity;
}

public OrderItem(int orderItemId, Product product, int quantity)
{
    OrderItemId = orderItemId;
    Product = product;
    Quantity = quantity;
}
}

```

The last step is to insert order items into the OrderItems collection of the Invoice object:

```

...
Invoice invoice = new Invoice(DateTime.Now, customer);

foreach (CartLine cartline in cart.Lines) {
    OrderItem orderItem = new OrderItem(cartline.Product, cartline.Quantity);
    orderItem.ProductId = cartline.Product.ProductId;
    orderItem.Product = null;
    invoice.OrderItems.Add(orderItem);
}

db.Invoices.Add(invoice);
db.SaveChanges();

cart.Clear();

return View("Completed");
}
else
{
    return View(shippingDetails);
}
}

```

As we loop through each CartLine item in the Cart, we create a new OrderItem, and we'll add each OrderItem to the OrderItems collection in Invoice. But we also assign value to the foreign key property ProductId in OrderItem, and we set the value of navigation property Product to null. That is to tell EF not to insert each Product in Cart as a new row in the products tables. This behavior might be contrary to what you would expect, but EF's entity state behavior doesn't always align with your ideas of how it should work. You can read more about that in Julie Lerman's *MSDN Magazine* article [Data Points - Why Does Entity Framework Reinsert Existing Objects into My Database?](#)

Exercise 5, create a Book repository for handling database access

If you take a look at the controllers we have created so far, there is a good deal EF related code inside most of the controllers. This is not the best way to organize code because it is contrary to the principle of separation of concerns. Inside the controllers, you should call methods for retrieving, inserting, updating and deleting data, and you definitely

shouldn't worry about how it is done or the inner workings and peculiarities of EF. Besides, if you want to optimize some of your EF related code, you have to look through all the controllers, and you have to rewrite the code in most controllers if you want to switch another ORM (*Object Relational Mapper*).

The solution is to create repositories for each model class with methods that handle the CRUD functions we need. For each model, we need to get a collection, an item, and we need to save and delete item. That means - if we look at the Book model class – we must implement these methods:

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

namespace MbmStore.DAL
{
    public interface IBookRepository
    {
        IEnumerable<Book> GetBookList();
        Book GetBookById(int id);
        void SaveBook(Book book);
        Book DeleteBook(int bookId);
    }
}
```

Create the above interface inside the DAL folder.

The next step is to create a EFBookRepository class in the same DAL folder that implements the interface and handles all EF interaction:

```
using MbmStore.Models;
using System;
using System.Collections.Generic;
using System.Data.Entity;
using System.Linq;

namespace MbmStore.DAL
{
    public class EFBookRepository : IBookRepository
    {
        private MbmStoreContext db = new MbmStoreContext();

        public IEnumerable<Book> GetBookList()
        {
            return db.Books.ToList();
        }

        public Book GetBookById(int id)
        {
            return db.Books.Find(id);
        }

        public void SaveBook(Book book)
        {
            if (book.ProductId == 0)
            {
                book.CreatedDate = DateTime.Now;
                db.Books.Add(book);
                db.SaveChanges();
            }
        }
    }
}
```

```

        else
        {
            db.Entry(book).State = EntityState.Modified;
            db.Entry(book).Property(c => c.CreatedDate).IsModified = false;
            db.SaveChanges();
        }
    }

    public Book DeleteBook(int bookId)
    {
        Book book = db.Books.Find(bookId);
        db.Books.Remove(book);
        db.SaveChanges();
        return book;
    }
}

```

Notice that we only need one method for insert and update, because we'll insert or update based on the status of the ProductId property. If it doesn't exist we'll create a new book otherwise update an existing.

We are now ready to use our new EFBookRepository class inside the BookController:

```

public class BookController : Controller
{
    private IBookRepository repo = new EFBookRepository();

    // GET: Admin/Book
    public ActionResult Index()
    {
        return View(repo.GetBookList());
    }

    // GET: Admin/Book/Details/5
    public ActionResult Details(int? id)
    {
        if (id == null)
        {
            return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
        }
        int bookId = (int)id;
        Book book = repo.GetBookById(bookId);
        if (book == null)
        {
            return HttpNotFound();
        }
        return View(book);
    }

    // GET: Admin/Book/Create
    public ActionResult Create()
    {
        return View();
    }

    // POST: Admin/Book/Create
    // To protect from overposting attacks, please enable the specific properties you want
    to bind to, for

```

```

// more details see http://go.microsoft.com/fwlink/?LinkId=317598.
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Create([Bind(Include =
"Title,Price,ImageUrl,Category,Author,Publisher,Published,ISBN,tmp")] Book book)
{
    try
    {
        if (ModelState.IsValid)
        {
            repo.SaveBook(book);
            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(book);
}

// GET: Admin/Book/Edit/5
public ActionResult Edit(int? id)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }
    int bookId = (int)id;
    Book book = repo.GetBookById(bookId);
    if (book == null)
    {
        return HttpNotFound();
    }
    return View(book);
}

// POST: Admin/Book/Edit/5
// To protect from overposting attacks, please enable the specific properties you want
to bind to, for
// more details see http://go.microsoft.com/fwlink/?LinkId=317598.
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Edit([Bind(Include =
"ProductId,Title,Price,ImageUrl,Category,Author,Publisher,Published,ISBN,tmp")] Book book)
{
    try
    {
        if (ModelState.IsValid)
        {
            repo.SaveBook(book);
            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(book);
}

// GET: Admin/Book/Delete/5
public ActionResult Delete(int? id, bool? saveChangesError = false)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }
    int bookId = (int)id;

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

    Book book = repo.GetBookById(bookId);
    if (book == null)
    {
        return HttpNotFound();
    }
    return View(book);
}

// POST: Admin/Book/Delete/5
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Delete(int id)
{
    try
    {
        Book book = repo.DeleteBook(id);
    }
    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");
}
}

```

The code inside the controller is much cleaner now, because we are dealing with objects and collection of objects and doesn't have to worry about the syntax and inner workings of EF.

Exercise 6, using Fluent API

We have followed the Entity Framework (EF) conventions for primary and foreign keys and therefore we haven't used DataAnnotation attributes to override default Code-First conventions except for annotating the OrderDate attribute with the SQL Server datetime2 type. This ensures the correct converting of the datetime data type between .NET and SQL Server.

```

public class Invoice
{
    private List<OrderItem> orderItems = new List<OrderItem>();

    public int InvoiceId { get; set; }

    [Column(TypeName = "datetime2")]
    public DateTime OrderDate { get; set; }
    public decimal TotalPrice { get { return orderItems.Sum(e => e.Product.Price *
e.Quantity); } }
    public List<OrderItem> OrderItems { get { return orderItems; } set { orderItems = value;
} }
    public int CustomerId { get; set; }

    // navigation property
    public virtual Customer Customer { get; set; }

    ...
}

```

In order to clean up our code a bit, we'll remove this database specific logic from the model class and use the Fluent API instead.

Open the `mbmStoreContext.cs` file in the DAL folder, and insert these few lines that specifies the same mapping as the `DataAnnotation` attribute:

```

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

    //Configure Invoice Column
    modelBuilder.Entity<Invoice>()
        .Property(p => p.OrderDate)
        .HasColumnType("datetime2");
}

```

You can now delete the `DataAnnotation` attribute in the `Invoice` class.

The Fluent API gives you an effective way to setup the mapping between your model and the database. See [Fluent API in Code-First](#) for further information.

Exercise 7, repositories for Movie and MusicCD, optional

Create similar repository classes for `Movie` and `MusicCD`.

Exercise 8, create an Invoice repository for handling database access, optional

1. Create a new `IInvoiceRepository` interface like this:

```

public interface IInvoiceRepository
{
    Product GetProductById(int productId);
    void SaveInvoice(Cart cart, ShippingDetails shippingDetails);
}

```



```
}
```

2. Create a `EFInvoiceRepository` class that implements these two methods.
3. Use the class in the `CartController` instead of the Entity Framework `DbContext` class `MbmStoreContext`.

Exercise 9, fix invoice prices, optional

We do have a problem with the shopping cart and ordering process, because there can be a change in the prices from the time users place items in the shopping cart and until they complete their orders. It is important that the price that is stated when users click **Add to cart** is the price they are billed for.

To fix that have to add a new `Price` property to the `CartLine` class:

```
public class CartLine
{
    public Product Product { get; set; }
    public int Quantity { get; set; }
    public decimal Price { get; set; }
}
```

That price must also be reflected in the `OrderItem` class and stored to the database when the order is executed. We'll add a `Price` property to the `OrderItem` class to handle that:

```
public class OrderItem
{
    public int OrderItemId { get; set; }
    public int ProductId { get; set; }
    public int InvoiceId { get; set; }
    public decimal Price { get; set; }
    public int Quantity { get; set; }
    public decimal TotalPrice { get { return Quantity * Price; } }

    // navigation property
    public virtual Product Product { get; set; }
}
```

We have now changed the model and that should be reflected in the database. To make the change to the database, you must add a migration:

```
PM> add-migration Product-CreatedDate-Attribute
```

Inspect migration and update the database:

```
PM> update-database
```

When new items are added to the Cart the CartLine property Price must be assigned to the value of the Price property of Product.

```
public class Cart
{
    private List<CartLine> lines = new List<CartLine>();

    public decimal TotalPrice
    {
        // Linq syntax
        get { return lines.Sum(e => e.Product.Price * e.Quantity); }
    }

    public List<CartLine> Lines { get { return lines; } }

    public Cart() { }

    public void AddItem(Product product, int quantity)
    {
        CartLine item = lines.Where(p => p.Product.ProductId ==
product.ProductId).FirstOrDefault();

        if (item == null)
        {
            lines.Add(new CartLine { Product = product, Quantity = quantity, Price =
product.Price });
        }
        else
        {
            item.Quantity += quantity;
        }
    }

    public void RemoveItem(Product product)
    {
        lines.RemoveAll(i => i.Product.ProductId == product.ProductId);
    }

    public void Clear()
    {
        lines.Clear();
    }
}
```

In Invoice the calculation of TotalPrice must use the OrderItem.Price instead of the Product.Price:

```
public class Invoice
{
    private List<OrderItem> orderItems = new List<OrderItem>();

    public int InvoiceId { get; set; }
    public DateTime OrderDate { get; set; }
    public decimal TotalPrice { get { return orderItems.Sum(e => e.Price * e.Quantity); } }
    public List<OrderItem> OrderItems {
        get { return orderItems; }
        set { orderItems = value; }
    }
}
```

```

    }
    public int CustomerId { get; set; }

    // navigation property
    public virtual Customer Customer { get; set; }

    ...
}

```

When the invoice is saved to the database, the price from the shopping cart value – which is the `CartLine.Price` – must be assigned to the price on the invoice – which is the `OrderItem.Price`:

```

public void SaveInvoice(Cart cart, ShippingDetails shippingDetails) {
    Customer customer = new Customer
    {
        Firstname = shippingDetails.Firstname,
        Lastname = shippingDetails.Lastname,
        Address = shippingDetails.Address,
        Zip = shippingDetails.Zip,
        Email = shippingDetails.Email
    };

    if (db.Customers.Any(c => c.Firstname == customer.Firstname && customer.Lastname ==
customer.Lastname && customer.Email == customer.Email))
    {
        customer = db.Customers.Where(c => c.Firstname == customer.Firstname && c.Lastname
== customer.Lastname && c.Email == customer.Email).First();
        customer.Address = shippingDetails.Address;
        customer.Zip = shippingDetails.Zip;
        // ensure update instead of insert
        db.Entry(customer).State = EntityState.Modified;
    }

    Invoice invoice = new Invoice(DateTime.Now, customer);

    foreach (CartLine cartLine in cart.Lines)
    {
        OrderItem orderItem = new OrderItem(cartLine.Product, cartLine.Quantity);
        // price when item went into the basket
        orderItem.Price = cartLine.Price;
        orderItem.ProductId = cartLine.Product.ProductId;
        orderItem.Product = null;
        invoice.OrderItems.Add(orderItem);
    }
    db.Invoices.Add(invoice);
    db.SaveChanges();
}

```

That's it. We now have a shopping cart system, which is protected against ongoing price changes.