

# **MVC and Databases**




## **Part 1 – Main functionality**

# Before you begin

- You should have installed
  - **Visual Studio Community 2016/2017**
  - **SQL Server Express 2016** (at the moment, SQL Server 2017 databases can't be deployed to our hosting service) and **SQL Server Management Studio** (2016 or 2017)
- You should have worked through the C# material and be comfortable with writing console applications in C#, including object-oriented programs
- You should have some knowledge of HTML and CSS
- You should have worked through the introductory MVC material, including the first MVC tutorial without a database (this is in Chapter 2 of the MVC book by Adam Freeman)
- You should have a core understanding of relational databases

# The Project

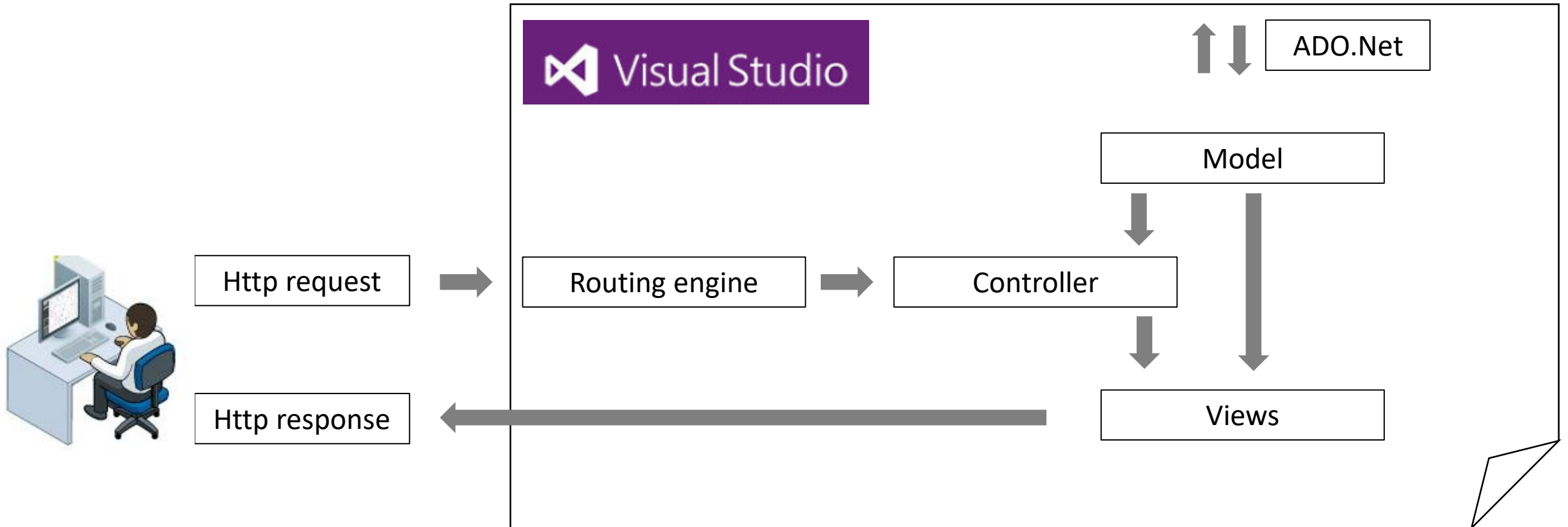
# Movie Application

Movies DB   Home   About   Contact					
Movies					
<a href="#">Create new movie</a>					
Genre: <input type="text" value="All"/> Title: <input type="text"/> <input type="button" value="Filter"/>					
Title	Release Date	Genre	Price		
Grosse Pointe Blank	11-Apr-1997	Comedy	£4.99		<a href="#">Edit Record</a> <a href="#">Details of Record</a> <a href="#">Delete Record</a>
The Force Awakens	16-Dec-2015	Science Fiction	£9.99		<a href="#">Edit Record</a> <a href="#">Details of Record</a> <a href="#">Delete Record</a>
Hidden Figures	25-Dec-2016	Biographical	£12.99		<a href="#">Edit Record</a> <a href="#">Details of Record</a> <a href="#">Delete Record</a>

## Requirements

- This is an MVC5 application with a **single-table SQL database**
- It has **Create, Read, Update and Delete (CRUD) functionality**
- It has filtering and searching
- A database-first approach is used to create the database, which is then attached to the project by creating an ADO.NET Entity Framework model

# **The *Architecture***



# The Workflow

## Requirements

## Design and Create the Database

Connect to SQL Server using SSMS  
Create DB

## SDLC

Review Requirements  
Design Solution  
Implement Solution  
Project Management

## Create Project

In Visual Studio create project  
ADO.Net Entity Data Model

## Setup Environment

SQL Server 2016 Express  
SSMS  
Visual Studio Community 2016  
Test Connections

## Build Application

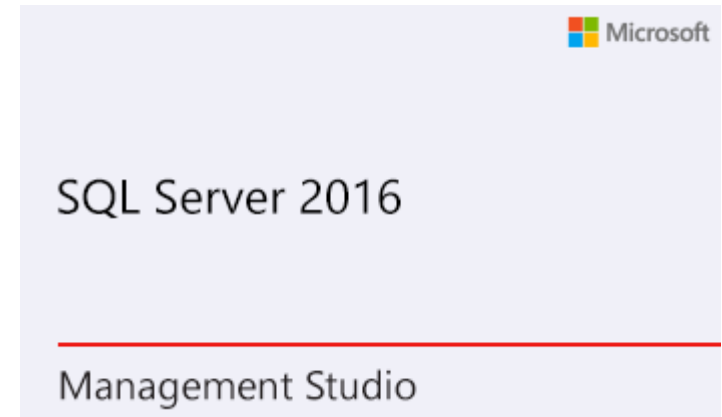
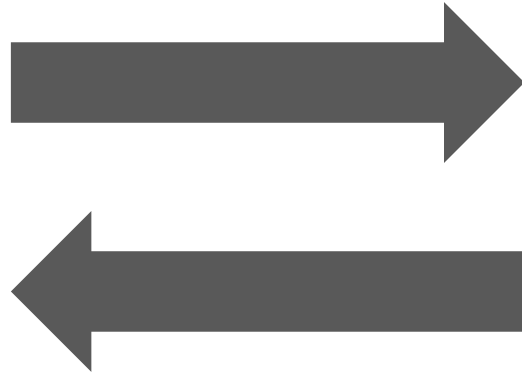
Create Model  
Create Controllers  
Create Views  
Test

## Deploy Application

Connect to Smarter ASP.Net  
Add Database  
Create Folder Container  
Publish Application from Visual Studio

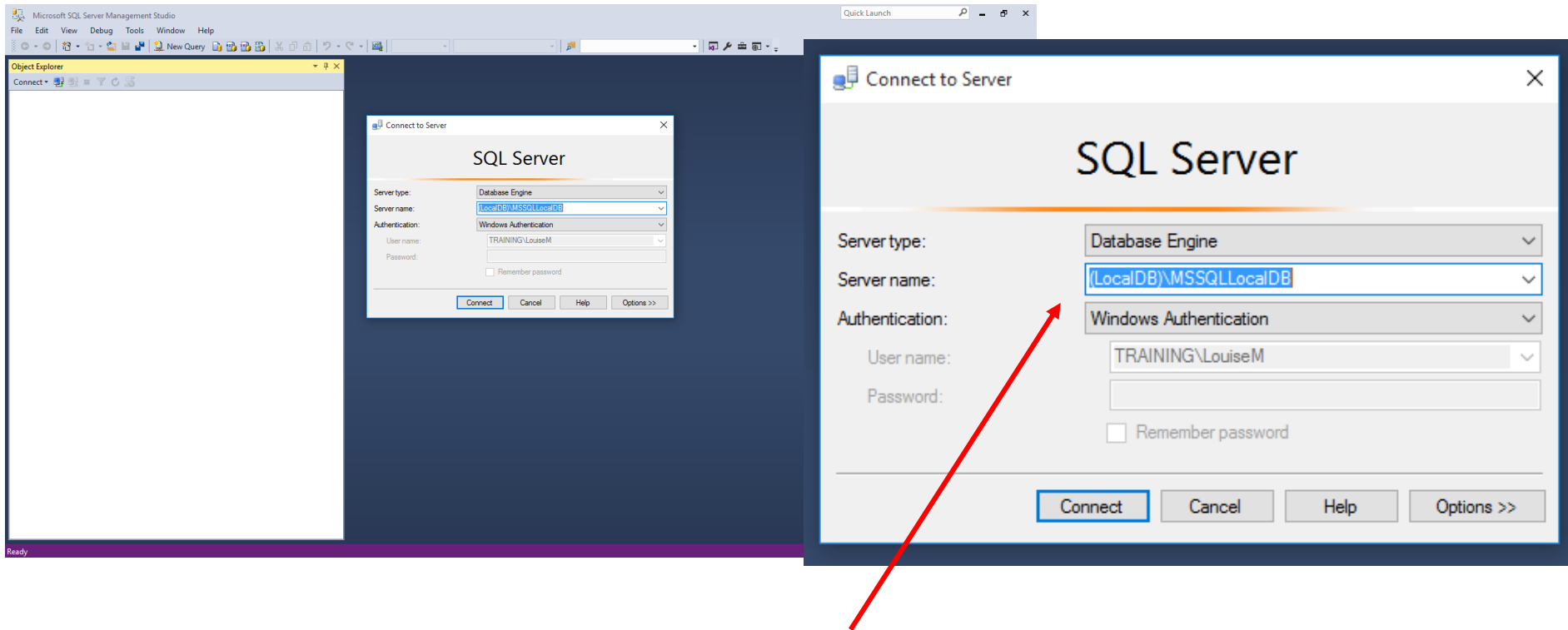


# Create the SQL Server Database



<https://www.microsoft.com/en-gb/sql-server/sql-server-editions-express>

# Open SQL Server Management Studio



- Start SQL Server Management Studio and connect with the following credentials:
  - Server type: **Database Engine**
  - Server name: **(LocalDB)\MSSQLLocalDB** (type this in if it isn't in the list of options)
  - Authentication: **Windows Authentication**
- If you get an error here, see the instructions at the end of the database slides

## Using SQL Server Management Studio

### To Attach a Database

1. In SQL Server Management Studio Object Explorer, connect to an instance of the SQL Server Database Engine, and then click to expand that instance view in SSMS.

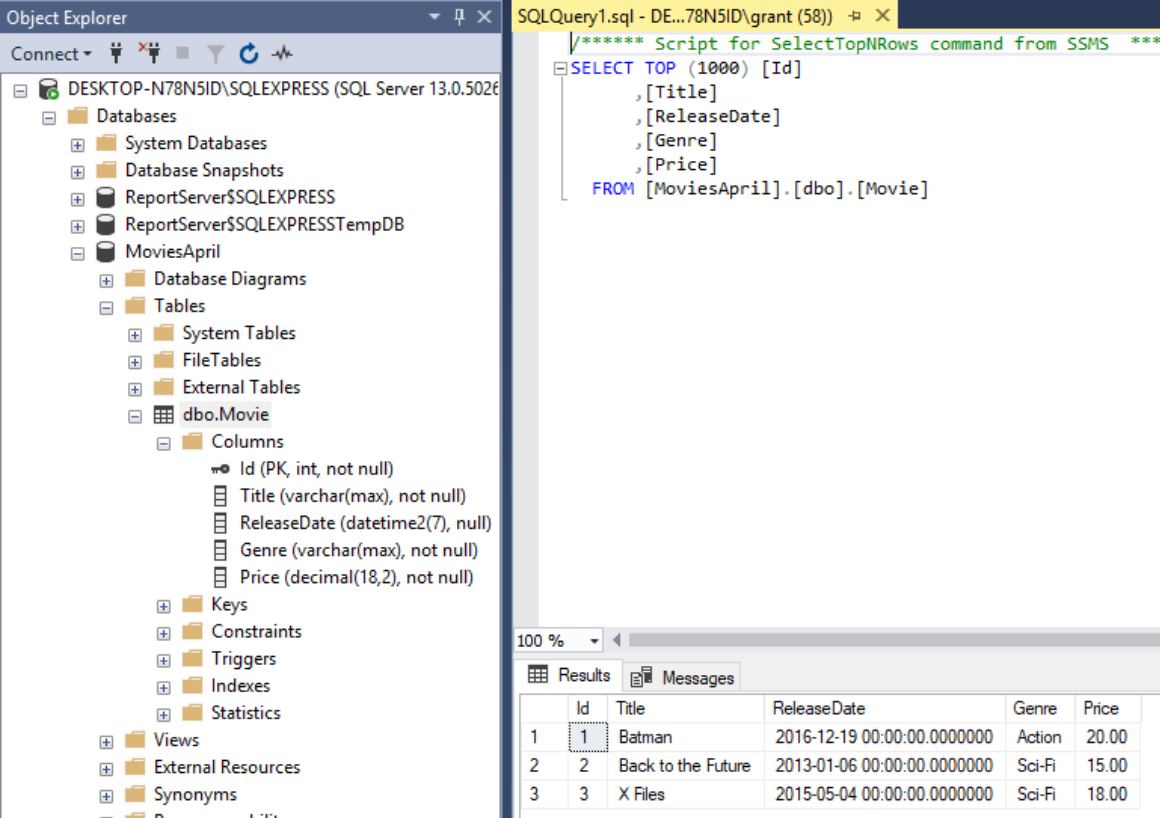
2. Right-click **Databases** and click **Attach**.

3. In the **Attach Databases** dialog box, to specify the database to be attached, click **Add**; and in the **Locate Database Files** dialog box, select the disk drive where the database resides and expand the directory tree to find and select the .mdf file of the database; for example:

C:\Program Files\Microsoft SQL  
Server\MSSQL13.MSSQLSERVER\MSSQL\DATA\AdventureWorks\_Data.mdf

### Important

Trying to select a database that is already attached generates an error.



The screenshot displays the SQL Server Management Studio interface. On the left, the Object Explorer shows the 'Databases' folder expanded, with 'MoviesApril' selected. The 'dbo.Movie' table is also visible. The main window shows a SQL query in the 'SQLQuery1.sql' editor:

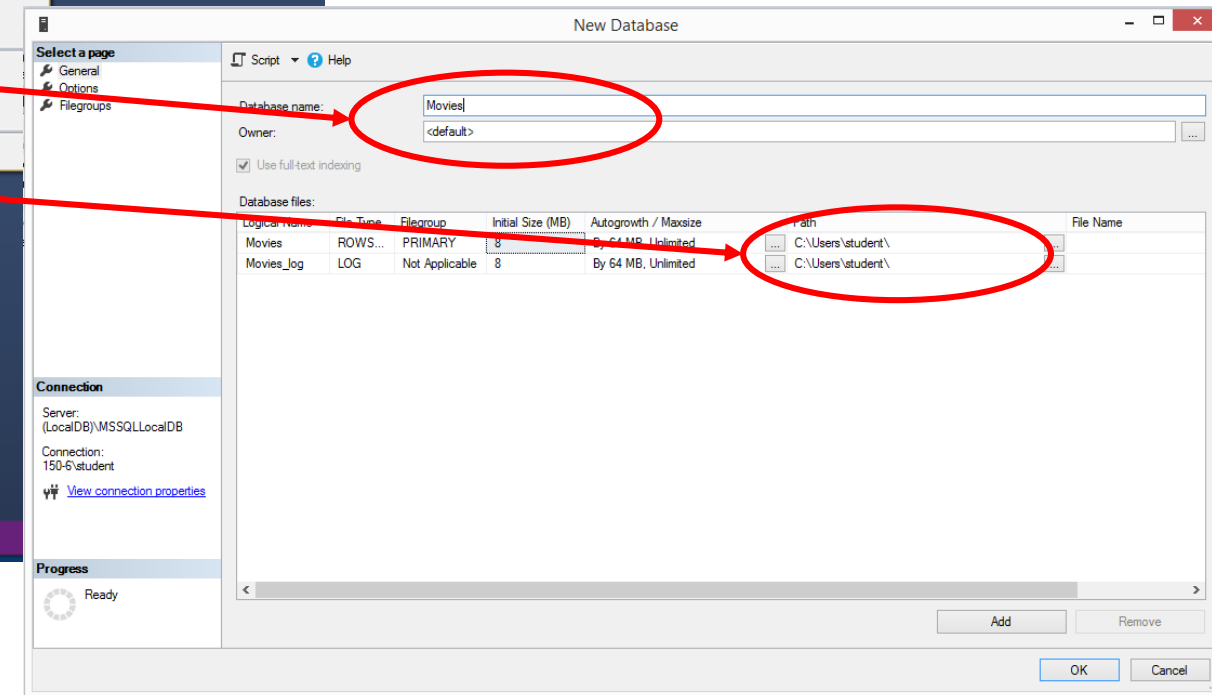
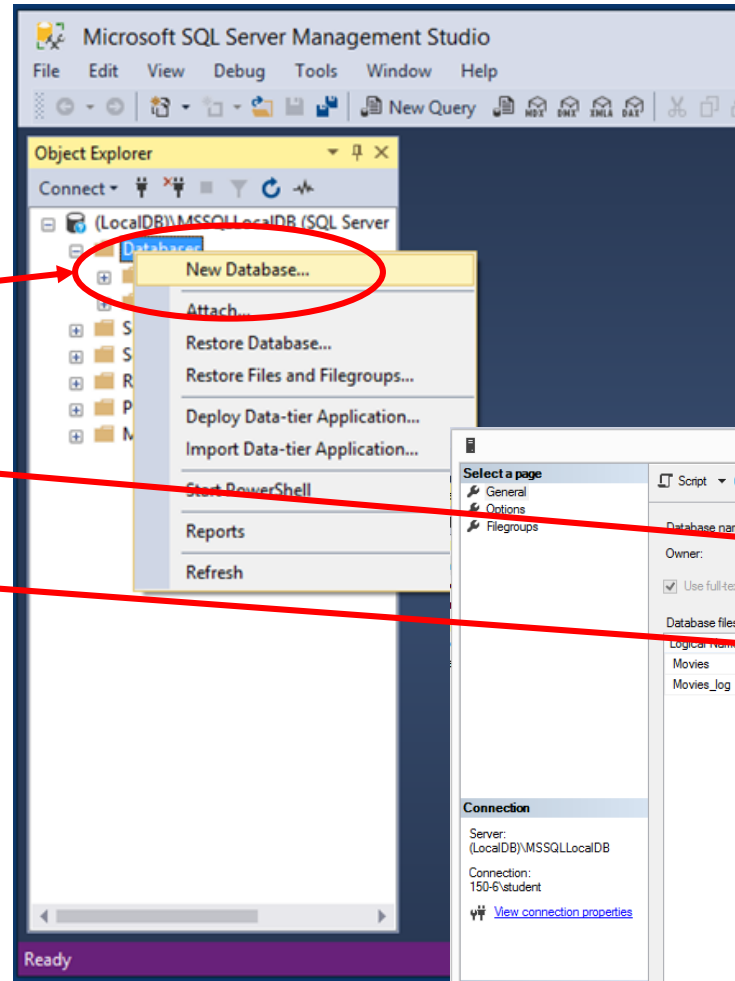
```
/****** Script for SelectTopNRows command from SSMS *****/  
SELECT TOP (1000) [Id]  
      ,[Title]  
      ,[ReleaseDate]  
      ,[Genre]  
      ,[Price]  
FROM [MoviesApril].[dbo].[Movie]
```

Below the query editor, the 'Results' tab is active, showing a table with 3 rows and 5 columns: Id, Title, ReleaseDate, Genre, and Price.

	Id	Title	ReleaseDate	Genre	Price
1	1	Batman	2016-12-19 00:00:00.0000000	Action	20.00
2	2	Back to the Future	2013-01-06 00:00:00.0000000	Sci-Fi	15.00
3	3	X Files	2015-05-04 00:00:00.0000000	Sci-Fi	18.00

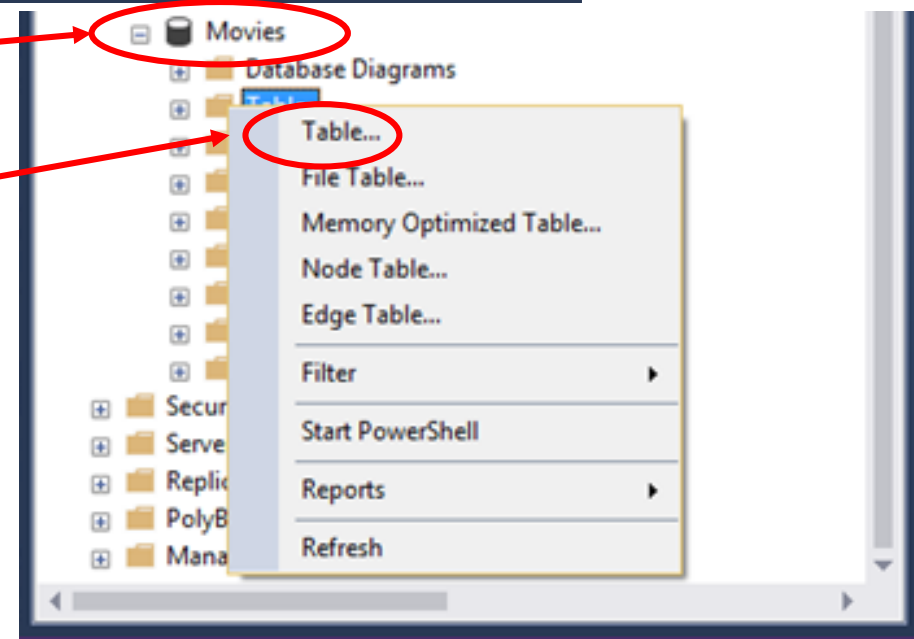
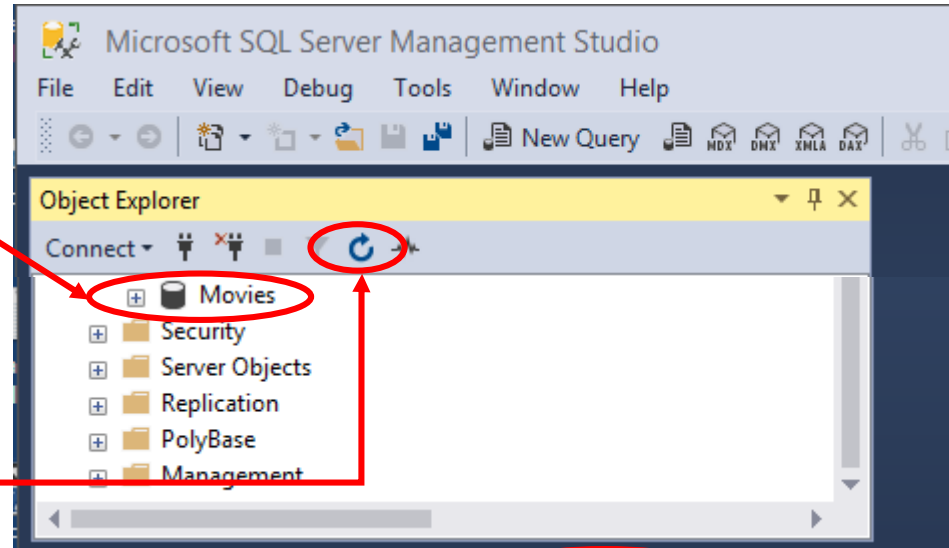
# SSMS – creating a database

- After logging in, right-click on the Databases folder in the Object Explorer
- Select New Database ...
- In the New Database dialog
  - Enter Movies as the name of your database
  - Expand the dialog to the right to check where the database file will be saved (it's usually in the folder C:\Users\[your username]) – you will need to know this later
  - If you change the location, make sure that both files are being saved in the same place (one is the data file and the other is the log file)



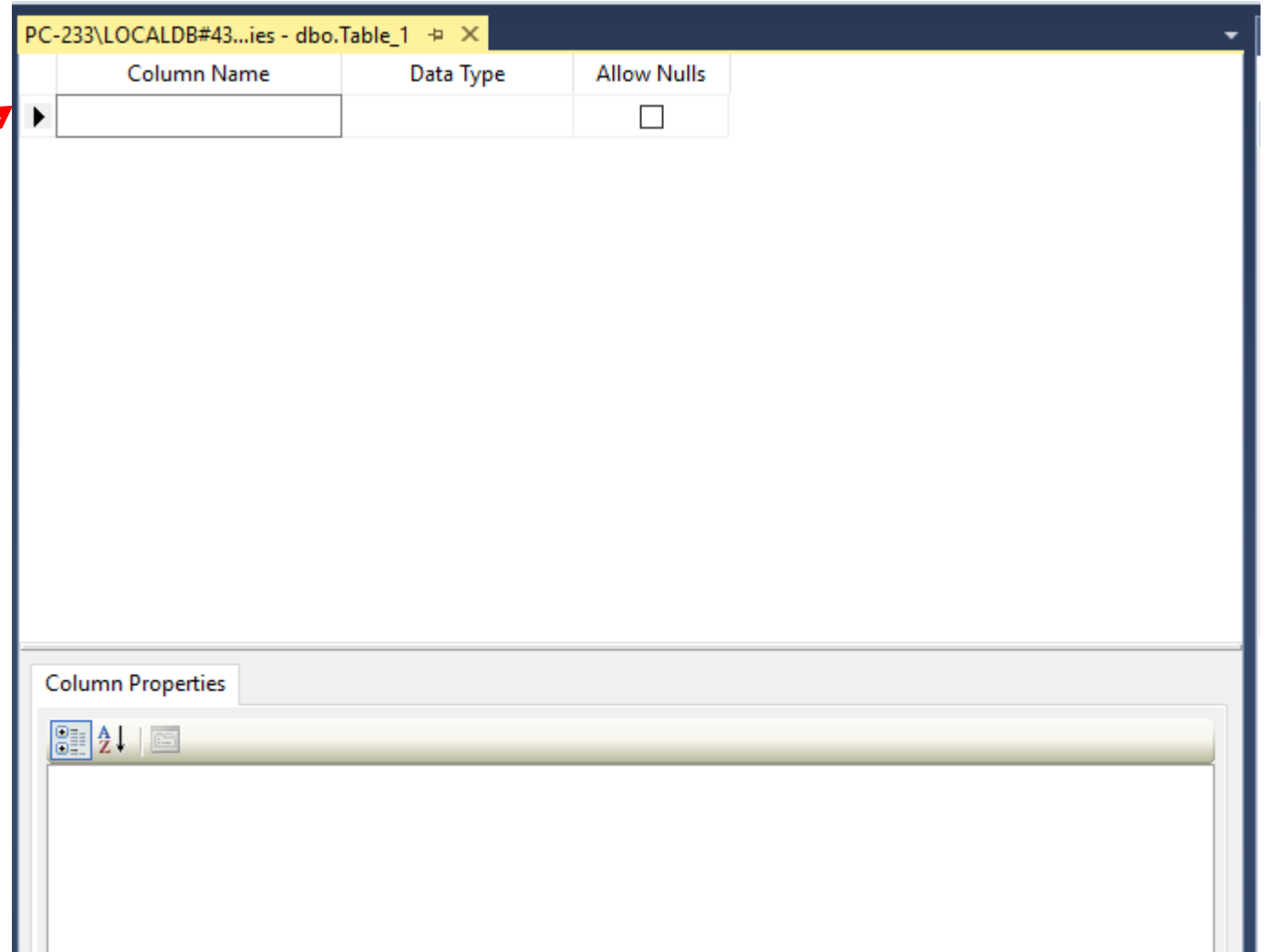
# Creating the database table

- You should now be able to see your database in the Object Explorer under the Databases folder
  - If it doesn't appear, click the Refresh button until it does
- Expand the node with your database name on it
- Right-click on the Tables folder and select Table...
  - This takes you to the design window



# Database designer

- You will now see the database design window
- Here you will enter details of the columns you want to have in your Movies table
- Each column will have a name and a data type and you will choose to allow or not allow nulls
  - Some columns, such as the primary key, need some extra information

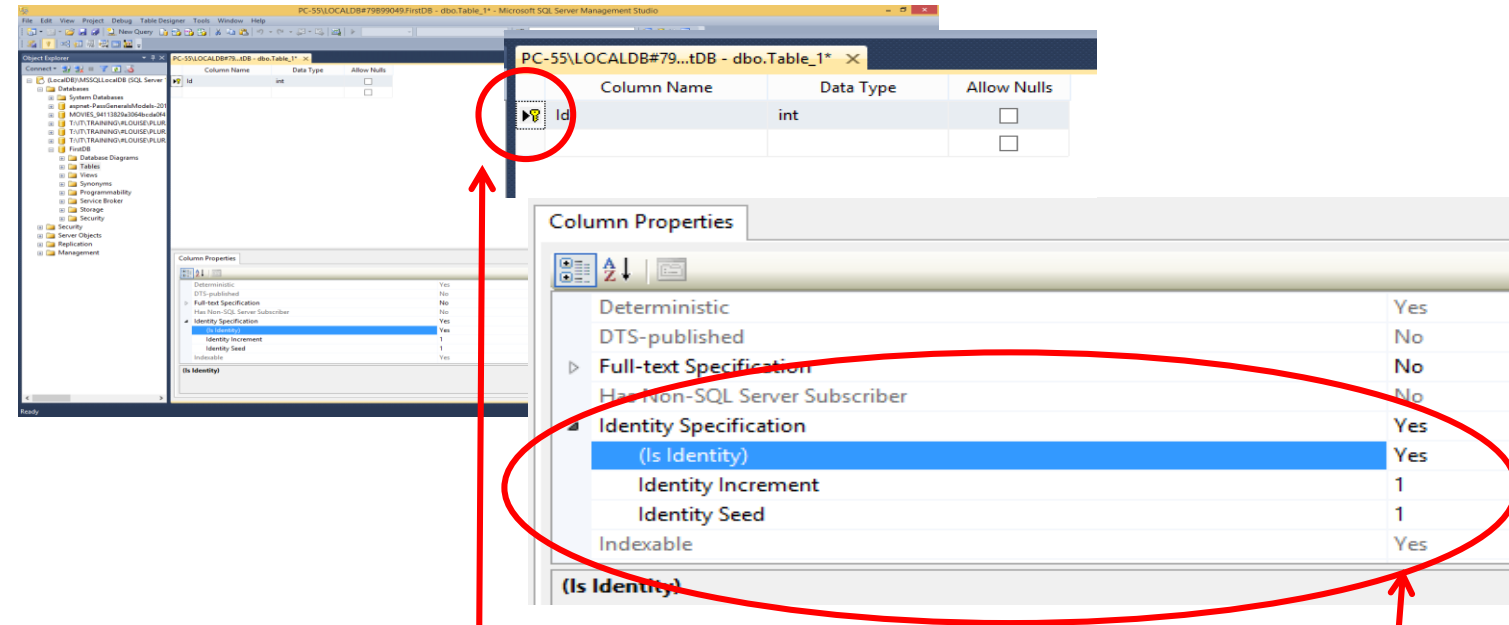


The screenshot shows a database designer window titled "PC-233\LOCALDB#43...ies - dbo.Table\_1". It features a table design grid with three columns: "Column Name", "Data Type", and "Allow Nulls". A red arrow points to the first row of the grid, which is currently empty. Below the grid is a "Column Properties" pane with a toolbar containing icons for setting primary key, index, and other properties.

Column Name	Data Type	Allow Nulls
		<input type="checkbox"/>

# Setting up your primary key

- Every database table must have a primary key
  - This identifies each record in your table so that the database software can find it
  - **A correctly set up primary key is essential for using the ADO.NET Entity Data Model to connect your database to your code**
- The primary key should be:
  - **Called Id (or [TableName]Id), with no spaces in the name**
  - Int data type
  - Nulls not allowed



- To complete the setup of your primary key:
  - Right-click on the black triangle next to it and select 'Set primary key' from the menu. A yellow key will appear next to the triangle
  - Go into the Column Properties tab (below the table's design details), find Identity Specification and change its values to Yes, Yes, 1, 1



# Finishing your table design

- Please enter the following (if you stick to this exactly, it will make the rest of the tutorial easier):
  - Id int no nulls
    - Set up as primary key as on previous slide
  - Title varchar(MAX) no nulls
  - ReleaseDate datetime2 nulls allowed
  - Genre varchar(MAX) no nulls
  - Price decimal(18,2) nulls allowed
  - ImageUrl varchar(MAX) nulls allowed

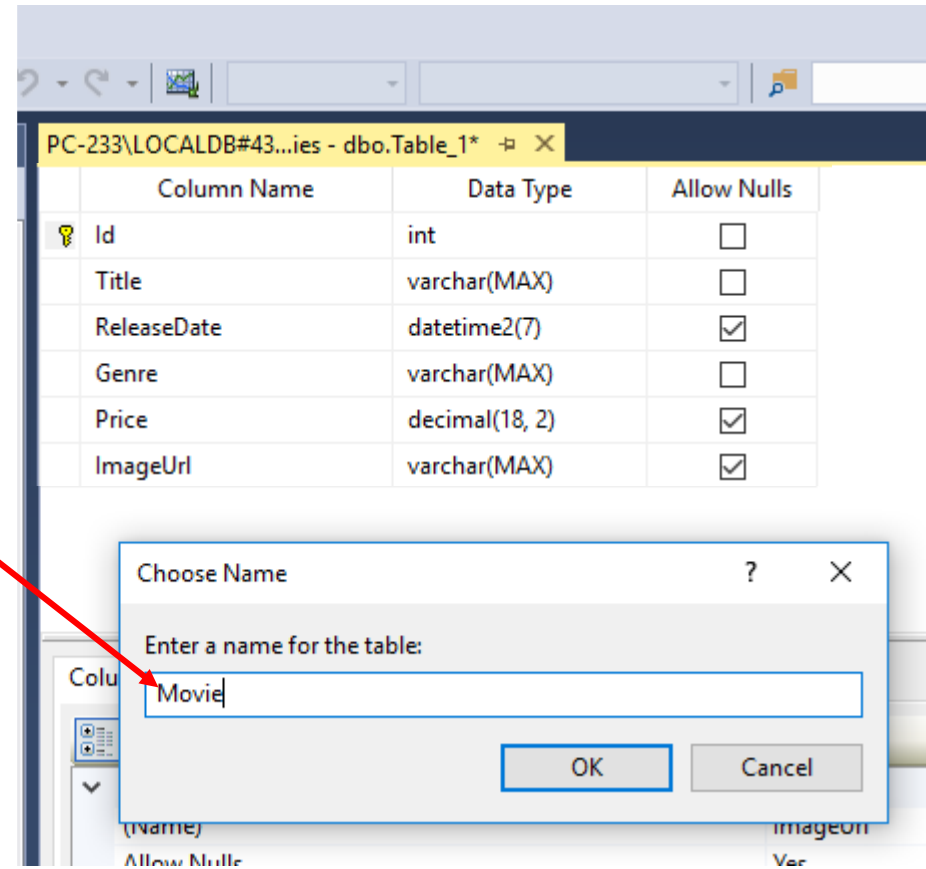
	Column Name	Data Type	Allow Nulls
🔑	Id	int	<input type="checkbox"/>
	Title	varchar(MAX)	<input type="checkbox"/>
	ReleaseDate	datetime2(7)	<input checked="" type="checkbox"/>
	Genre	varchar(MAX)	<input type="checkbox"/>
	Price	decimal(18, 2)	<input checked="" type="checkbox"/>
	ImageUrl	varchar(MAX)	<input checked="" type="checkbox"/>
▶			<input type="checkbox"/>

- Data types in SQL and C#:
  - int = int
  - varchar(MAX) = string
  - decimal(18,2) = decimal (with 2 decimal places – you can overwrite the numbers if you want more or fewer decimal places)
  - datetime2 = DateTime (recommended as it doesn't crash if a zero date is accidentally set)

- Tips for adding columns to your table
  - Avoid spaces in column names
  - Think about whether you want to allow null values
    - Allowing null values is recommended for dates, as there is a bug in Google Chrome that sometimes wipes dates. Allowing nulls will help prevent your code crashing
    - **Don't allow nulls for primary keys!**

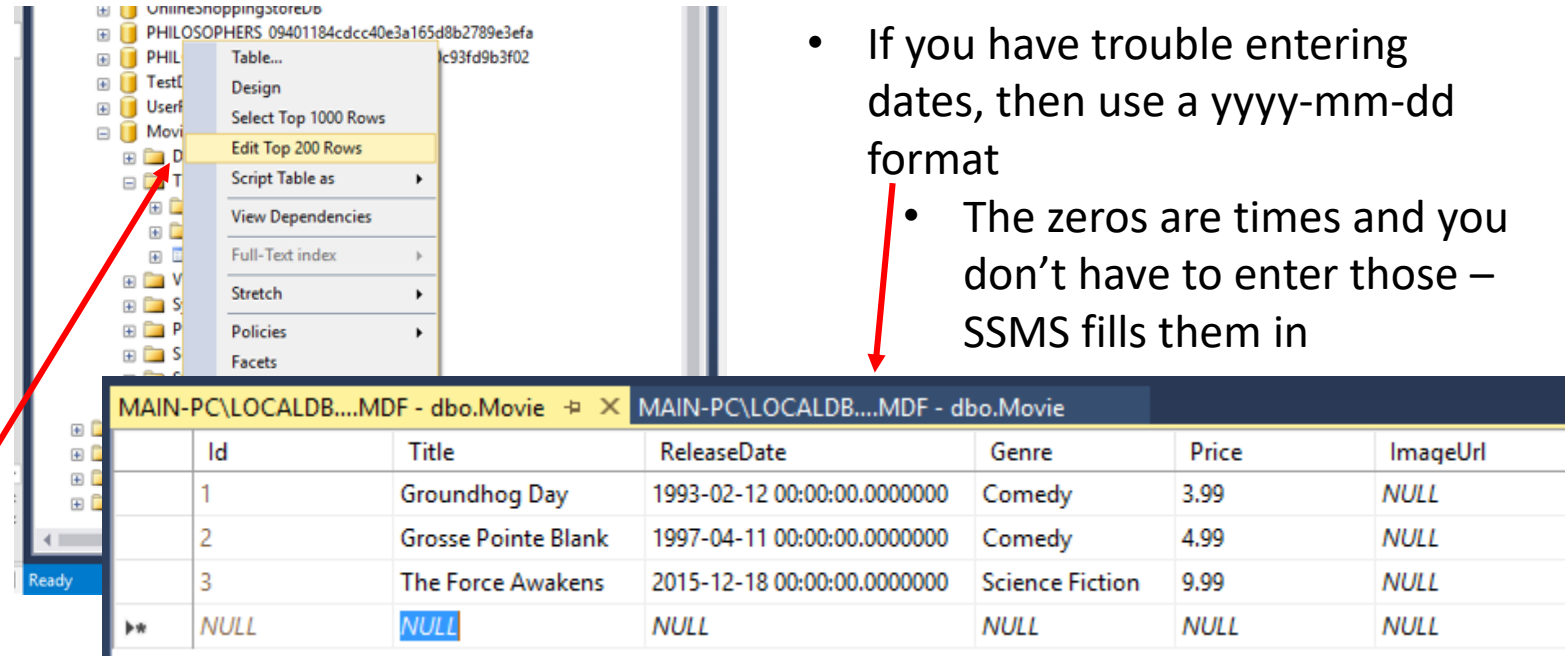
# Saving your table

- To save your table, click on the save button
- In the Choose Name dialog, name your table Movie and click OK
  - Table names should always be singular (not plural) with no spaces
- You may have to click Refresh in the Object Explorer to see the table under the Tables folder



# Entering data into your table

- It's best to add some data when you create a table, so that you have some data to display when you start working on your code
- Start this by right-clicking on the table in the Object Explorer
  - Select Edit Top 200 Rows from the menu

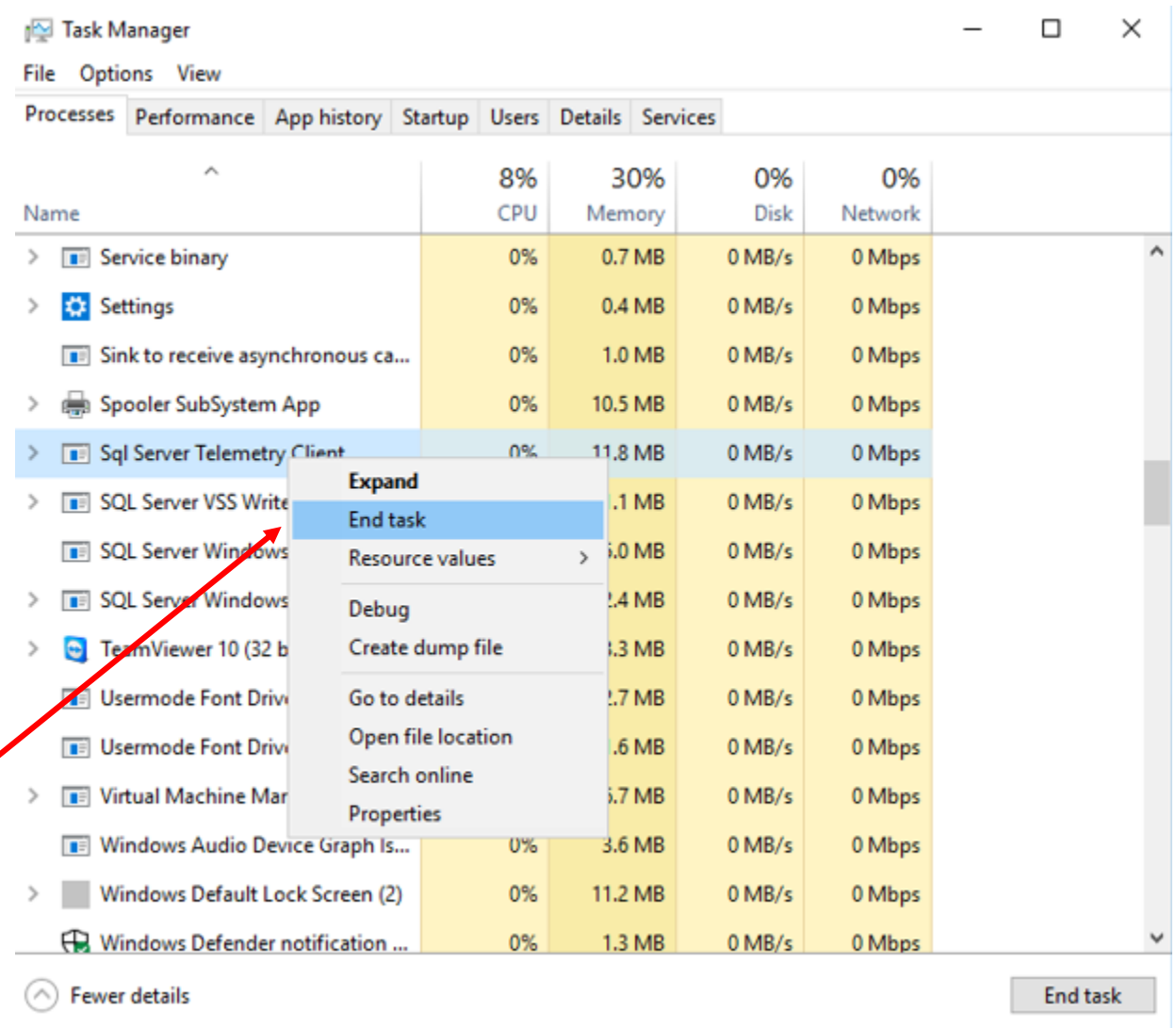


- If you have trouble entering dates, then use a yyyy-mm-dd format
  - The zeros are times and you don't have to enter those – SSMS fills them in

- You will then be able to add a few records to the table
  - Don't enter an id – SSMS will do this for you
  - Tab between fields and rows
  - Each record is saved when you finish a row
  - You can leave ImageUrl blank for now

# Preventing a 'file in use' error

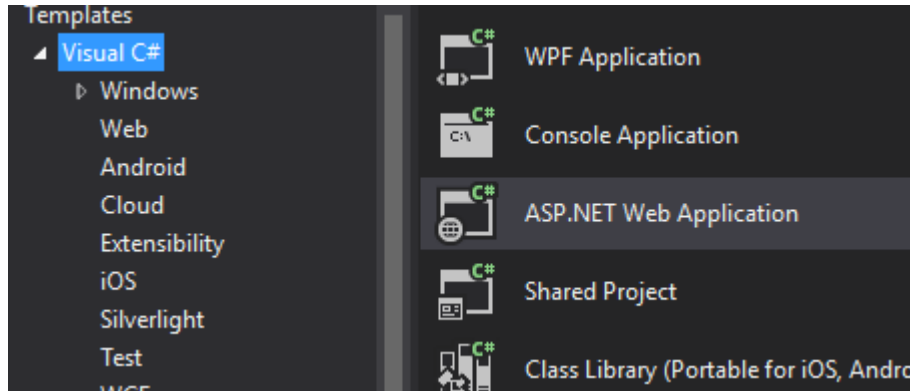
- Now close SQL Server Management Studio
- To prevent a 'file in use' error when you try to use the database file in Visual Studio:
  - Go to Task Manager
    - Right-click on the taskbar and select it from the menu
    - Or do Ctrl+Alt+Delete and select it from the menu
  - Close down all Background Processes that begin with SQL
    - Right-click on the process and select End Task from the menu



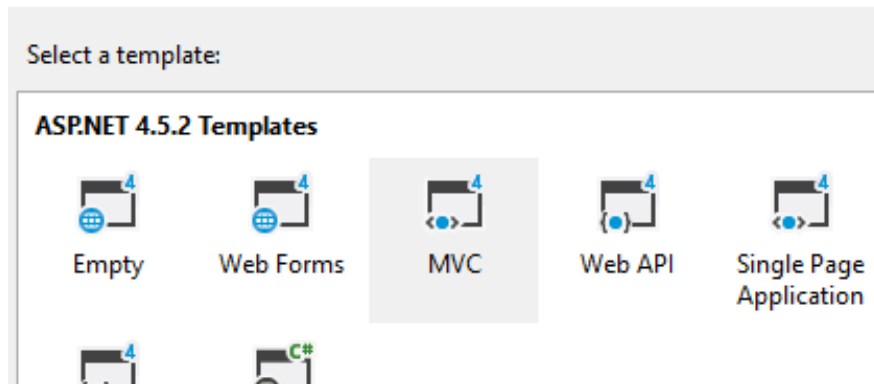
**Create the Project in Visual Studio**

# Create new MVC Project with SQL DB Authentication

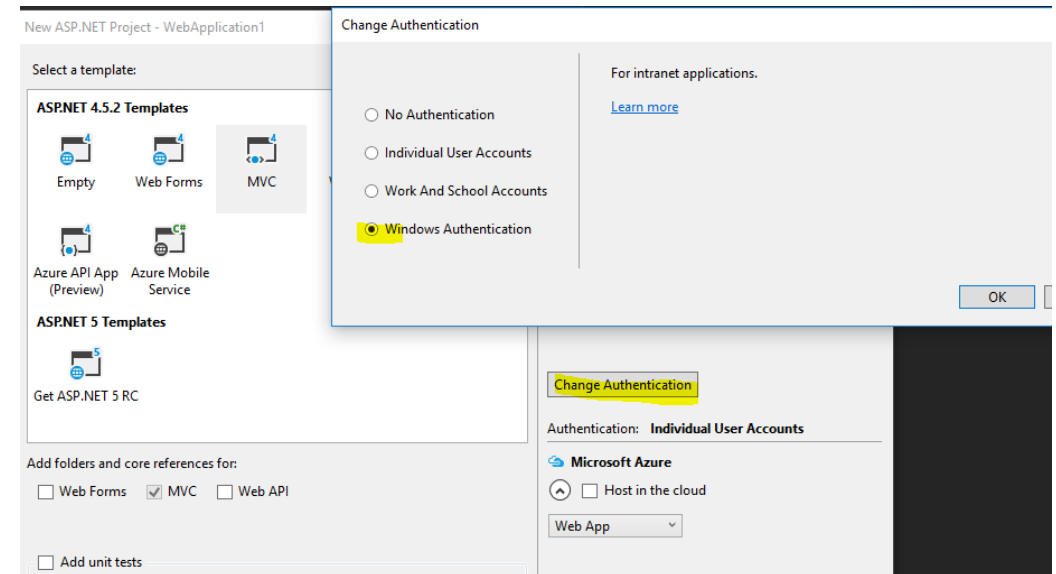
## New MVC Project



## New ASP.NET Project - WebApplication1

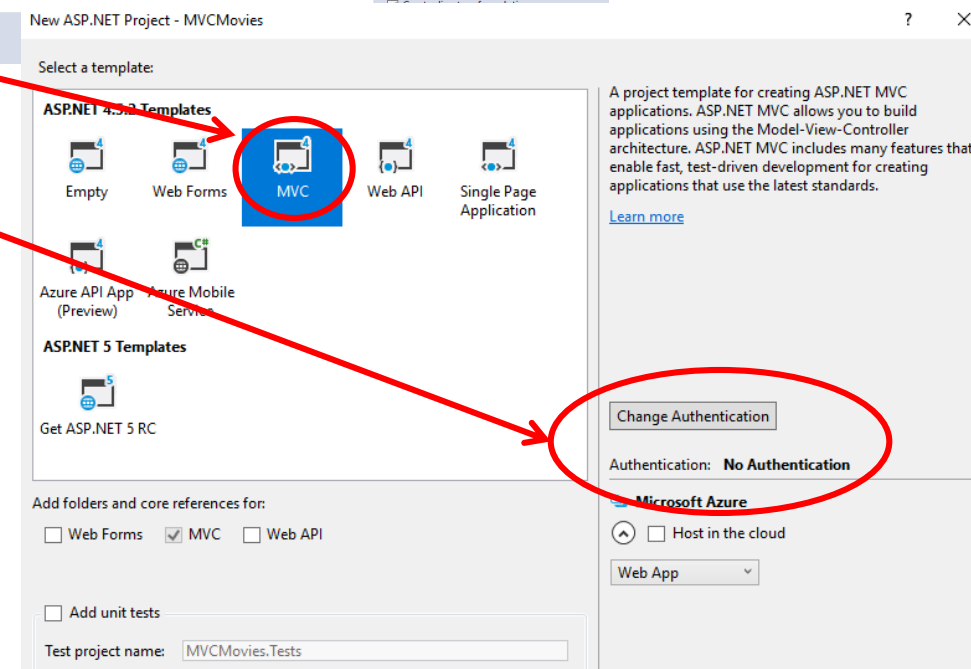
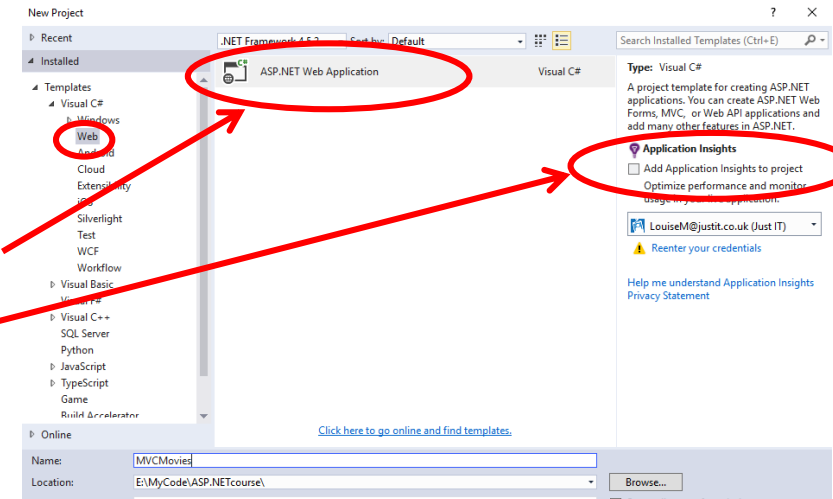


## Authentication set to Windows (SQL Database



# Starting a project in Visual Studio

- Open Visual Studio
- Start a new ASP.NET Web Application project
  - Leave the Application Insights box unchecked, as it's not relevant
- On the Select a Template dialog
  - Select the MVC icon with a small 4 in the top right corner
  - Click the **Change Authentication** button and select **No Authentication**
  - Leave Add Unit Tests and Host in the Cloud unchecked
  - Click OK and wait for Visual Studio to start the project (this may take a couple of minutes, as it is assembling a lot of files)



# **Connecting the Database to the project**



# ADO.NET Entity Framework

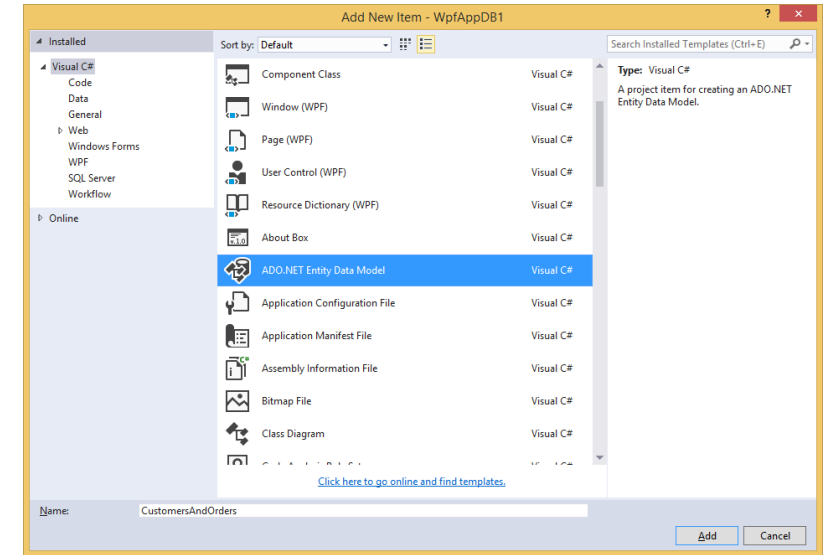
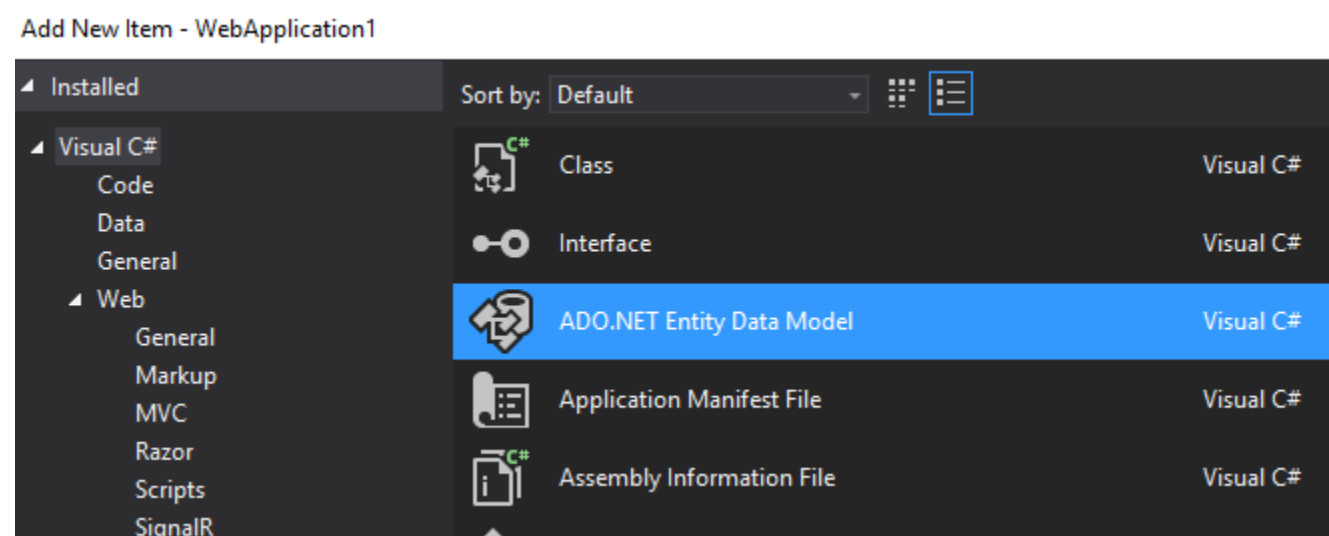
- Entity framework is an Object/Relational Mapping (O/RM) framework
- It is an enhancement to ADO.NET that gives developers an automated mechanism for accessing and storing data in the database
- Easy to use and saves a lot of work
- Can be set up in Visual Studio

# **ADO.Net**

## **Build an Entity Model**

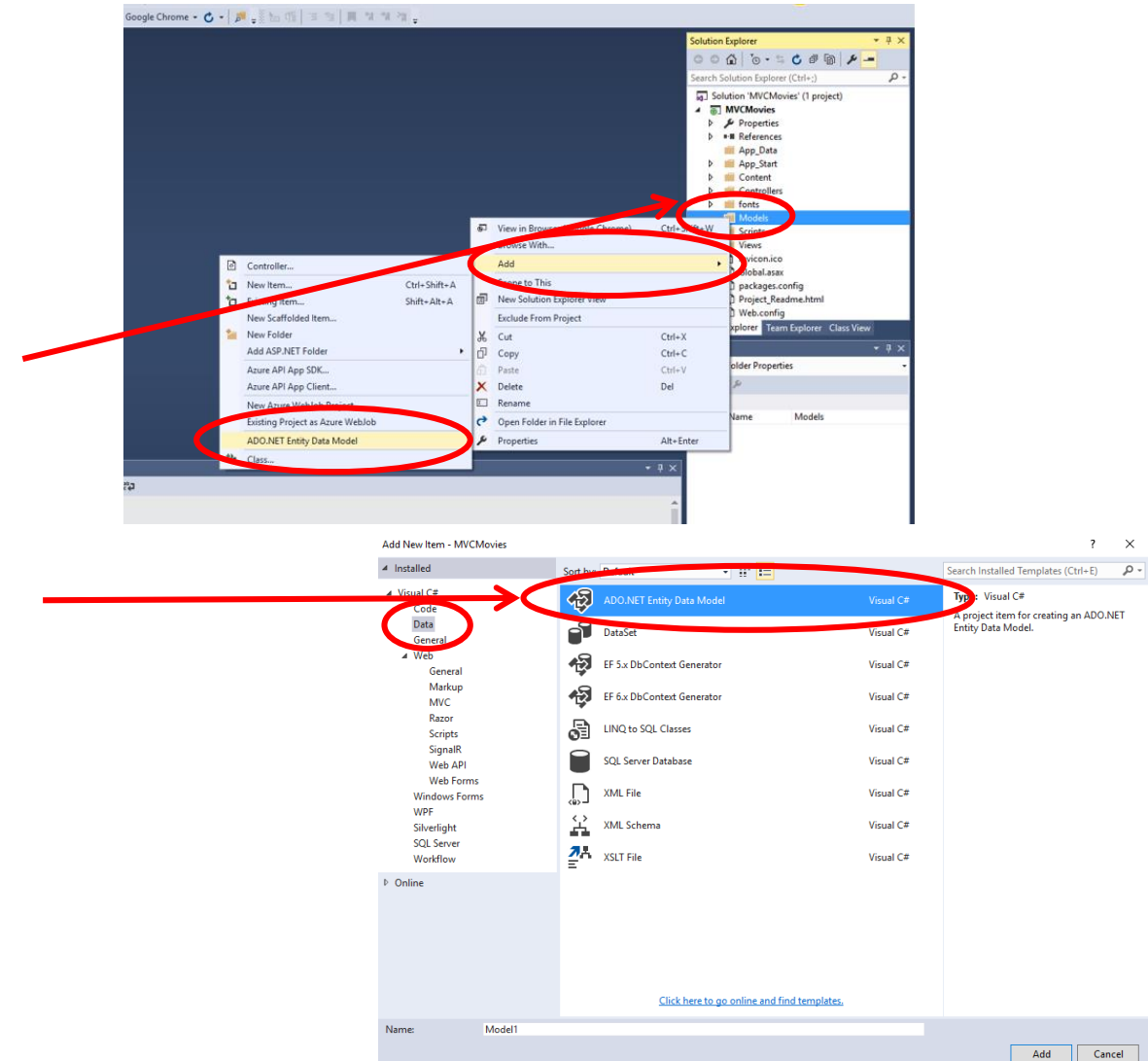
# Adding an ADO.NET Entity Data Model

- Right-click on the **Models** folder
- Select Add New Item
- Select ADO.NET Entity Data Model (either from that menu or from the dialog shown here)
- No need to change the model name



- You will need to add an ADO.NET Entity Data model to your project to link your database to your project

- In the Solution Explorer, right-click on the Models folder
- Select Add New Item
- Select ADO.NET Entity Data Model from the menu
  - If it's not on the menu, select Add New Item and then select ADO.NET Entity Data Model from the dialog shown here (you may have to select Data on the left-hand side of the dialog)
- Click OK to start creating the model. There's no need to change the model name



# Select EF Designer



Choose Model Contents

What should the model contain?



EF Designer  
from  
database



Empty EF  
Designer  
model



Empty Code  
First model




Code First  
from  
database

Creates a model in the EF Designer based on an existing database. You can choose the database connection, settings for the model, and database objects to include in the model. The classes your application will interact with are generated from the model.

# Add new Connection

Entity Data Model Wizard ✕

 **Choose Your Data Connection**

**Which data connection should your application use to connect to the database?**

New Connection...

This connection string appears to contain sensitive data (for example, a password) that is required to connect to the database. Storing sensitive data in the connection string can be a security risk. Do you want to include this sensitive data in the connection string?

☐ No, exclude sensitive data from the connection string. I will set it in my application code.


☐ Yes, include the sensitive data in the connection string.

**Connection string:**

☒ Save connection settings in Web.Config as:

# Connect to the SQL Database

Entity Data Model Wizard

 Choose Your Data Connection

Which data connection should your application use to connect to the database?

This connection string appears to contain sensitive data (for example, a password) that is required to connect to the database. Storing sensitive data in the connection string can be a security risk. Do you want to include this sensitive data in the connection string?

☐ No, exclude sensitive data from the connection string. I will set it in my application code.


☐ Yes, include the sensitive data in the connection string.

Connection string:

```
metadata=res://*/CustomerModel.csdl|res://*/CustomerModel.ssdl|
res://*/CustomerModel.msl;provider=System.Data.SqlClient;provider connection string="data source=
(LocalDB)\MSSQLLocalDB;attachdbfilename=C:\Users\Louise\FirstDB.mdf;integrated
security=True;connect timeout=30;MultipleActiveResultSets=True;App=EntityFramework"
```

☒ Save connection settings in App.Config as:

Entity Data Model Wizard

 Choose Your Data Connection

Which data connection should your application use to connect to the database?

This connection string appears to contain sensitive data (for example, a password) that is required to connect to the database. Storing sensitive data in the connection string can be a security risk. Do you want to include this sensitive data in the connection string?

☐ No, exclude sensitive data from the connection string. I will set it in my application code.


☐ Yes, include the sensitive data in the connection string.

Connection string:

```
metadata=res://*/CustomerModel.csdl|res://*/CustomerModel.ssdl|
res://*/CustomerModel.msl;provider=System.Data.SqlClient;provider connection string="data source=
(LocalDB)\MSSQLLocalDB;attachdbfilename=C:\Users\Louise\FirstDB.mdf;integrated
security=True;connect timeout=30;MultipleActiveResultSets=True;App=EntityFramework"
```

☒ Save connection settings in App.Config as:

Microsoft Visual Studio

 The connection you selected uses a local data file that is not in the current project. Would you like to copy the file to your project and modify the connection?

Connection Properties

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

Data source:

Server name:

Log on to the server

Authentication:

User name:

Password:

☐ Save my password

Connect to a database

☒ Select or enter a database name:

☐ Attach a database file:

Logical name:

Two options for connecting to a DB here – try the one below first

Connection Properties

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) Change...

Database file name (new or existing):  
C:\Users\louisem\TestDB.mdf Browse...

Log on to the server

☒ Use Windows Authentication  
☐ Use SQL Server Authentication

User name:   
Password:   
☐ Save my password

Advanced...

Test Connection OK Cancel

Then click Test Connection – if you get a green tick, then click OK

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 (SqlClient) Change...

Server name:  
(LocalDB)\MSSQLLocalDB Refresh

Log on to the server

☒ Use Windows Authentication  
☐ Use SQL Server Authentication

User name:   
Password:   
☐ Save my password

Connect to a database

☒ Select or enter a database name:

☐ Attach a database file:  
 Browse...

Logical name:

Advanced...

Test Connection OK Cancel


Same DB as we used to log in to SSMS

Put the DB filename here



# Entities Model created (DatabaseNameEntities)

Entity Data Model Wizard

 Choose Your Data Connection

Which data connection should your application use to connect to the database?

desktop-n78n5id\sqlexpress.MoviesApril.dbo New Connection...

This connection string appears to contain sensitive data (for example, a password) that is required to connect to the database. Storing sensitive data in the connection string can be a security risk. Do you want to include this sensitive data in the connection string?

☐ No, exclude sensitive data from the connection string. I will set it in my application code.

☐ Yes, include the sensitive data in the connection string.

Connection string:

```
metadata=res://*/Models.Model1.csdl|res://*/Models.Model1.ssdl|
res://*/Models.Model1.msl;provider=System.Data.SqlClient;provider connection string="data
source=DESKTOP-N78N5ID\SQLEXPRESS;initial catalog=MoviesApril;integrated
security=True;MultipleActiveResultSets=True;App=EntityFramework"
```

☒ Save connection settings in Web.Config as:

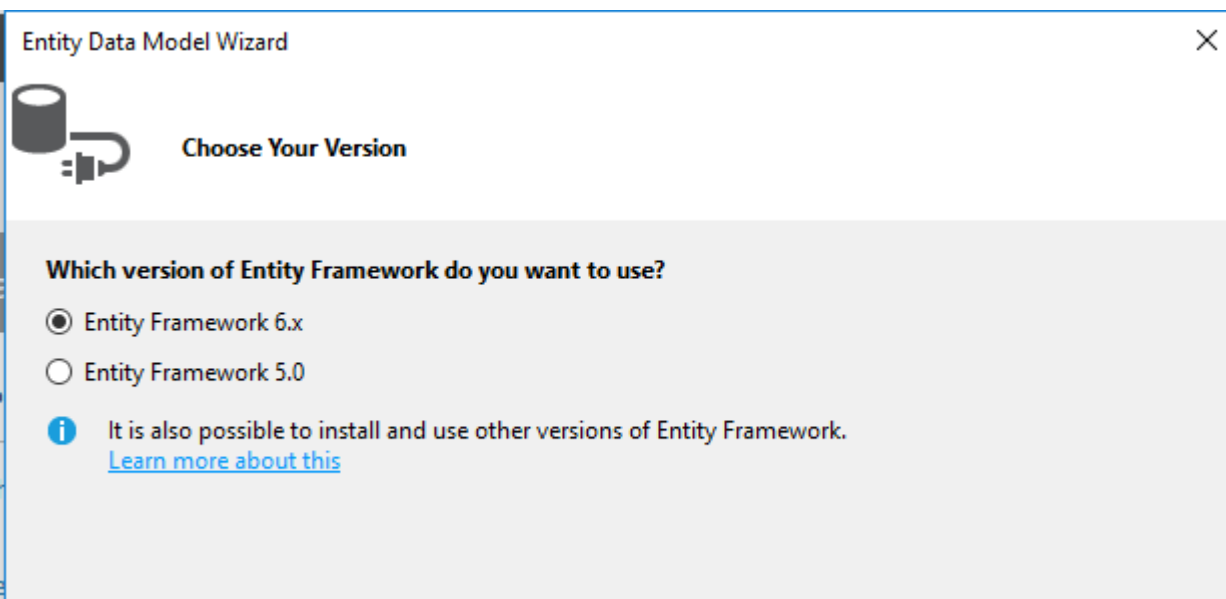
MoviesAprilEntities

< Previous Next > Finish Cancel

☒ Save connection settings in Web.Config as:


MoviesAprilEntities

# Select Framework



# Select the Database Objects to be part of the model

Entity Data Model Wizard

 Choose Your Database Objects and Settings

Which database objects do you want to include in your model?

- ☒ Tables
  - ☒ dbo
    - ☒ Movie
  - ☐ Views
  - ☐ Stored Procedures and Functions

☒ Pluralize or singularize generated object names

☒ Include foreign key columns in the model

☐ Import selected stored procedures and functions into the entity model

Model Namespace:

MoviesAprilModel

< Previous   Next >   **Finish**   Cancel

Model Namespace:

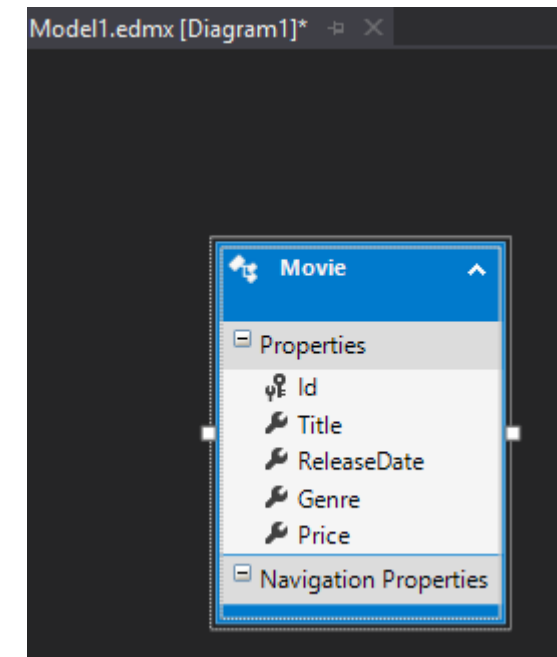
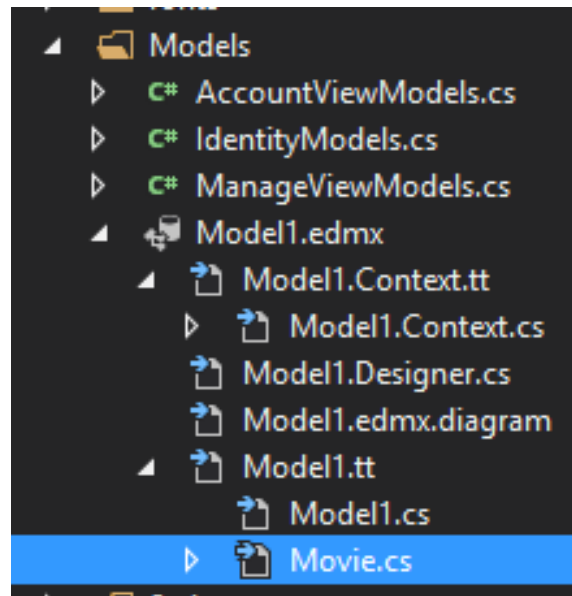
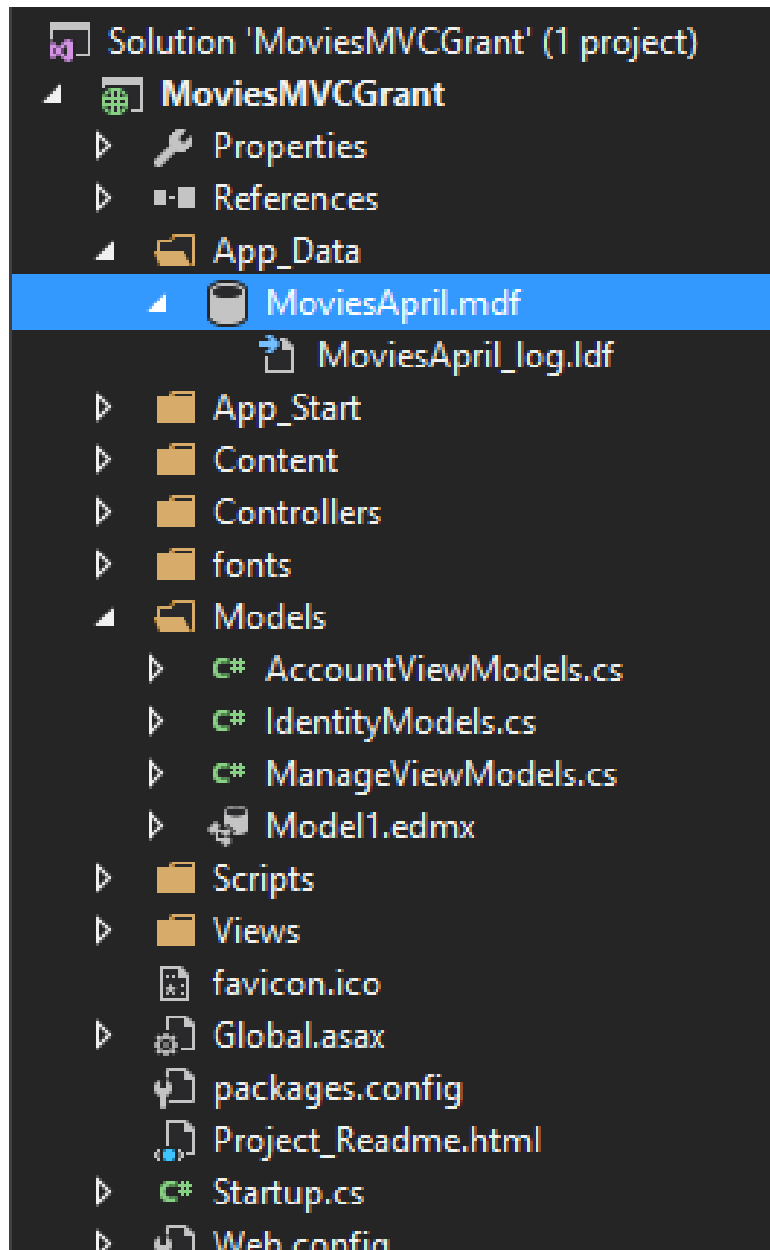
MoviesAprilModel

**ADO.Net Entity Model is then created**

**Checks**

**Model**

**Controller**



```
public partial class MoviesAprilEntitiesTwo : DbContext
{
    public MoviesAprilEntitiesTwo()
        : base("name=MoviesAprilEntitiesTwo")
    {
    }
}
```

```
namespace MoviesMVCGrant.Models
{
    using System;
    using System.Collections.Generic;

    public partial class Movie
    {
        public int Id { get; set; }
        public string Title { get; set; }
        public Nullable<System.DateTime> ReleaseDate { get; set; }
        public string Genre { get; set; }
        public decimal Price { get; set; }
    }
}
```

**Movie**

**Database and Model now connected into  
the project and can be access through  
the codebase**

**Enable the Home Controller to  
access the Data Model**

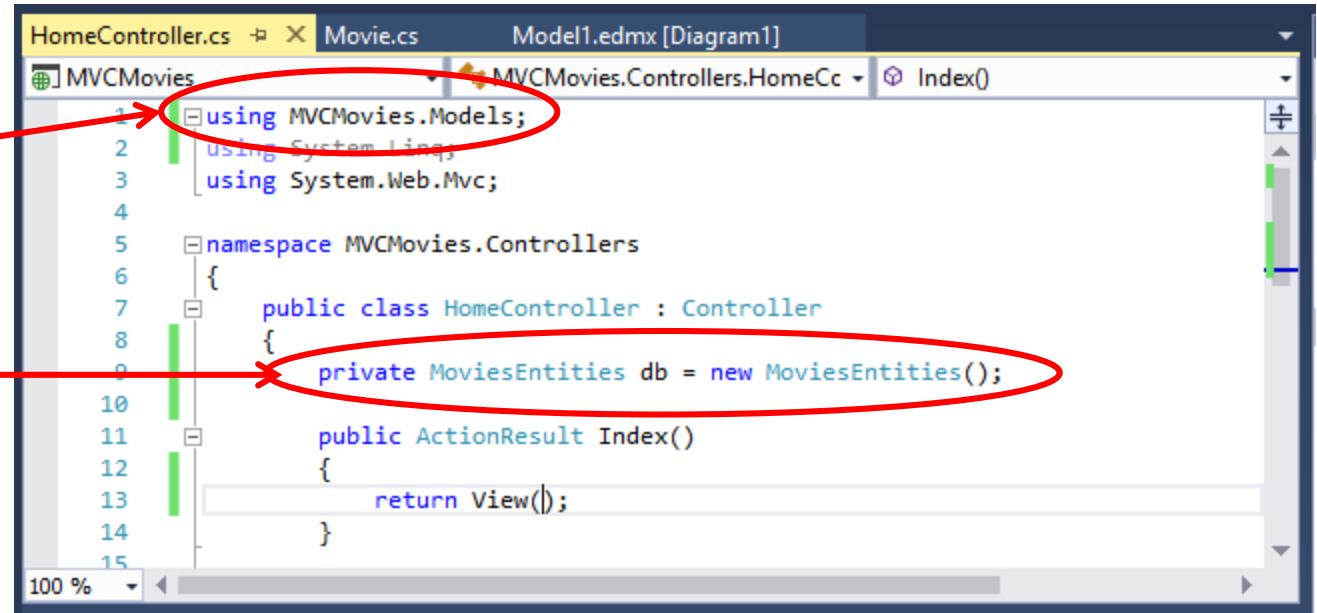


# Controller can now link to the model

```
public class HomeController : Controller
{
    //Set Access to the MoviesDB Entity and call it db
    private MoviesAprilEntitiesTwo db = new MoviesAprilEntitiesTwo();
}
```

# Adding a database connection object

- To get ready to use the db
  - Go into the Home Controller
  - Add a using statement for the Models folder
  - Declare a db connection object at the top of the Home Controller
    - `private MoviesEntities db = new MoviesEntities();`
- You can now write code to work with the database



The screenshot shows the Visual Studio IDE with the HomeController.cs file open. The file is part of the MVCMovies project, specifically in the MVCMovies.Controllers namespace. The code is as follows:

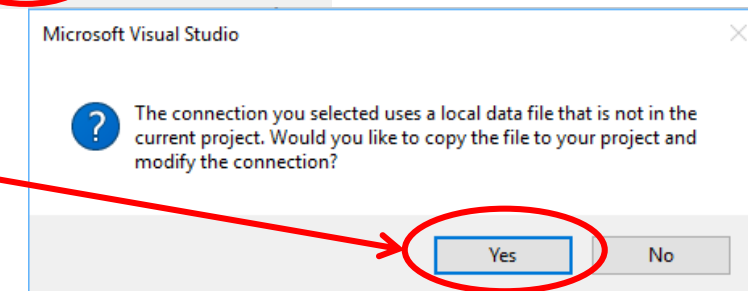
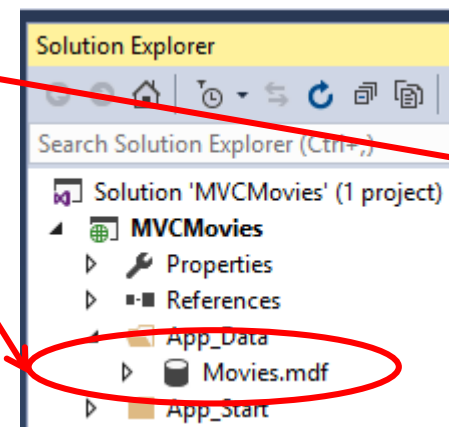
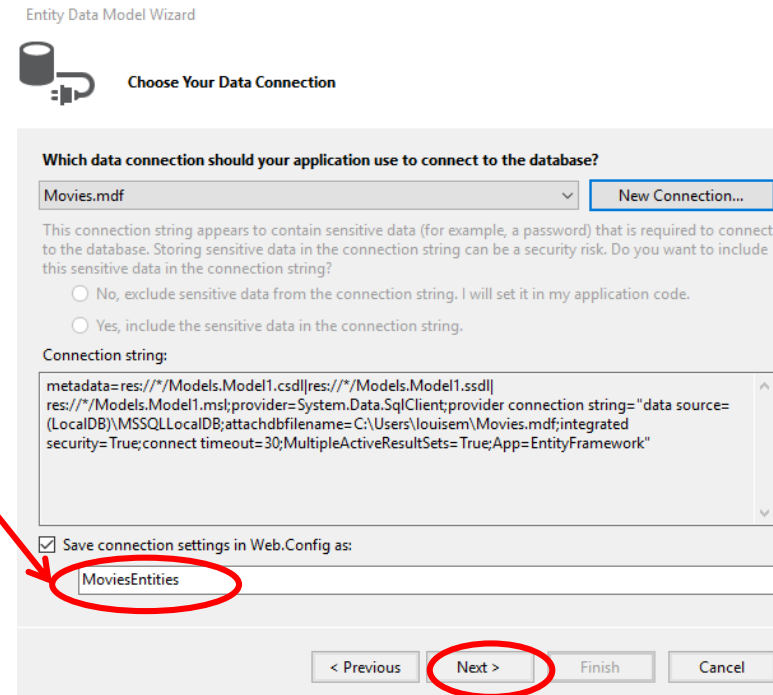
```
1 using MVCMovies.Models;
2 using System.Linq;
3 using System.Web.Mvc;
4
5 namespace MVCMovies.Controllers
6 {
7     public class HomeController : Controller
8     {
9         private MoviesEntities db = new MoviesEntities();
10
11         public ActionResult Index()
12         {
13             return View();
14         }
15 }
```

Two red annotations are present: a red oval around the using statement `using MVCMovies.Models;` on line 1, and a red oval around the declaration `private MoviesEntities db = new MoviesEntities();` on line 9. A red arrow points from the text 'Add a using statement for the Models folder' to the first oval, and another red arrow points from the text 'Declare a db connection object at the top of the Home Controller' to the second oval.

# If you need to change your database

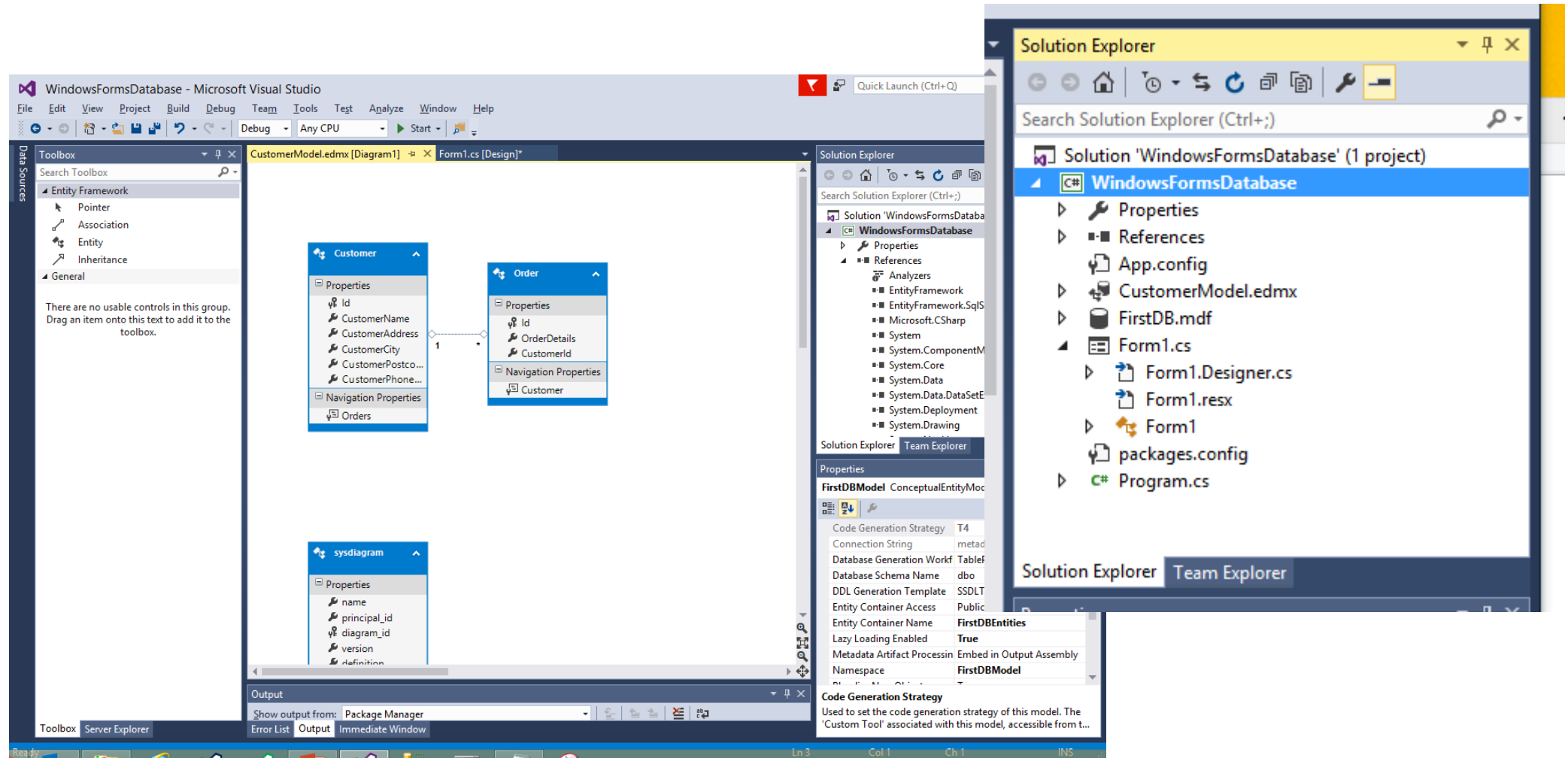
- It's really common that, while developing an application, you realise that you need to change the design of your database
- If you do want to change it, go back into SQL Server Management Studio (SSMS) and make the changes
  - Make sure that you are working on the copy of your database in the App\_Data folder of your project
  - Sometimes SSMS stops you from changing table designs
    - If that happens, go into Options, select Designers and uncheck the box that is labelled 'Prevent saving changes that require table re-creation. You should then be able to save the changes to your table.
- Close SSMS and go into Task Manager to shut down any SQL processes
- Open Visual Studio and your project
- Delete the existing model
  - I know this sounds scary, but you are only deleting the connection to your database, not the database itself
- Add a new model using your updated database file
- Update the declaration of your database connection object in your Home Controller, to match the class set up by the new ADO.NET model – you will probably add a 1 to it

- You will now go back to the Choose Your Data Connection screen
  - **PLEASE WAIT** until the text in the box at the bottom has changed to **MoviesEntities**
  - Make a note of this name, as you will use it in your code
  - Click Next
  - You will then see a dialog offering to copy your db file into the project
    - Say yes – it keeps your db safely with the project
    - Please remember that you now have 2 copies of your db and, if you go into SSMS to change it, you should use the copy that is now in the App\_Data folder of your project



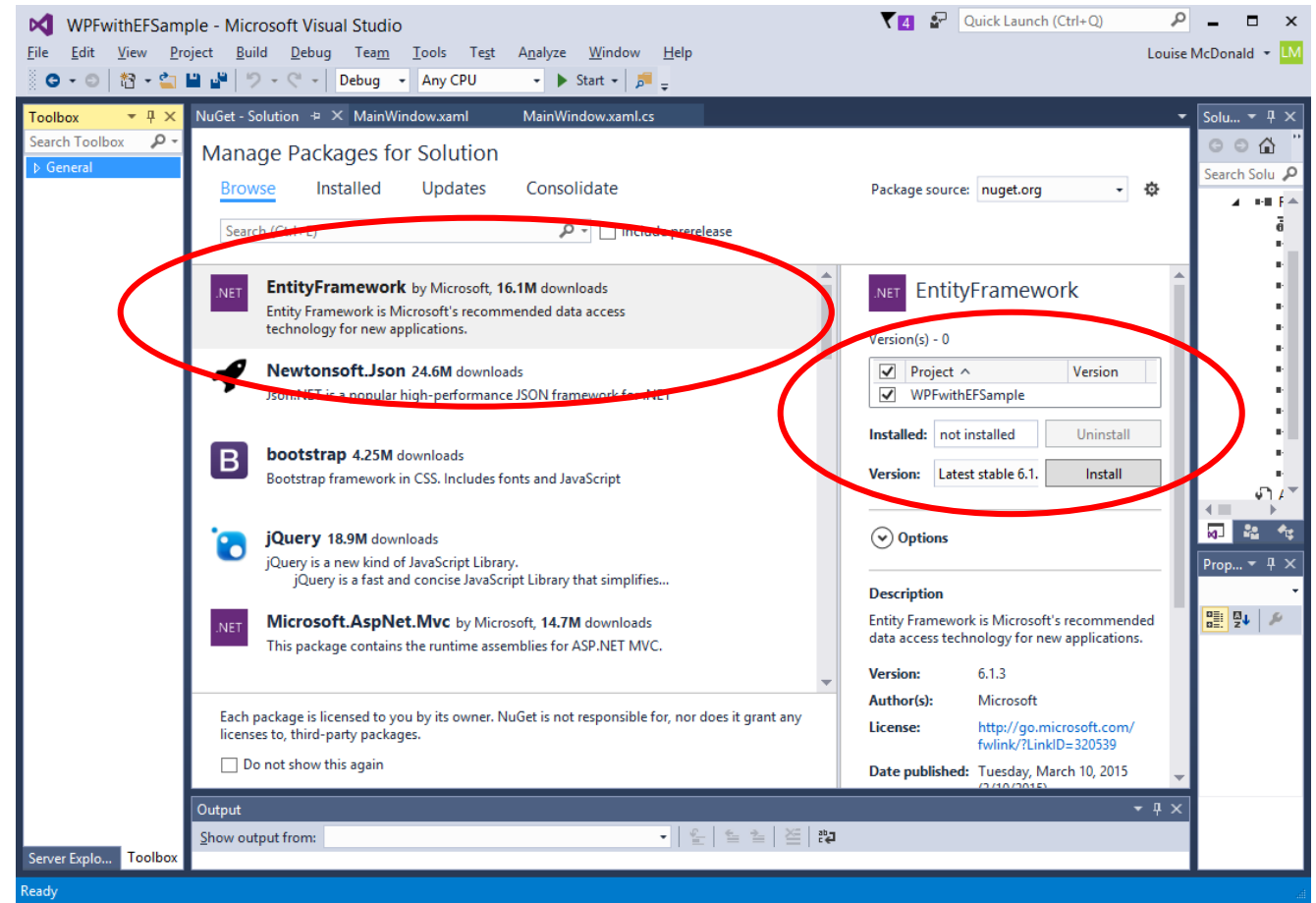
**EXTRAS**

# ADO.NET EF model in Visual Studio



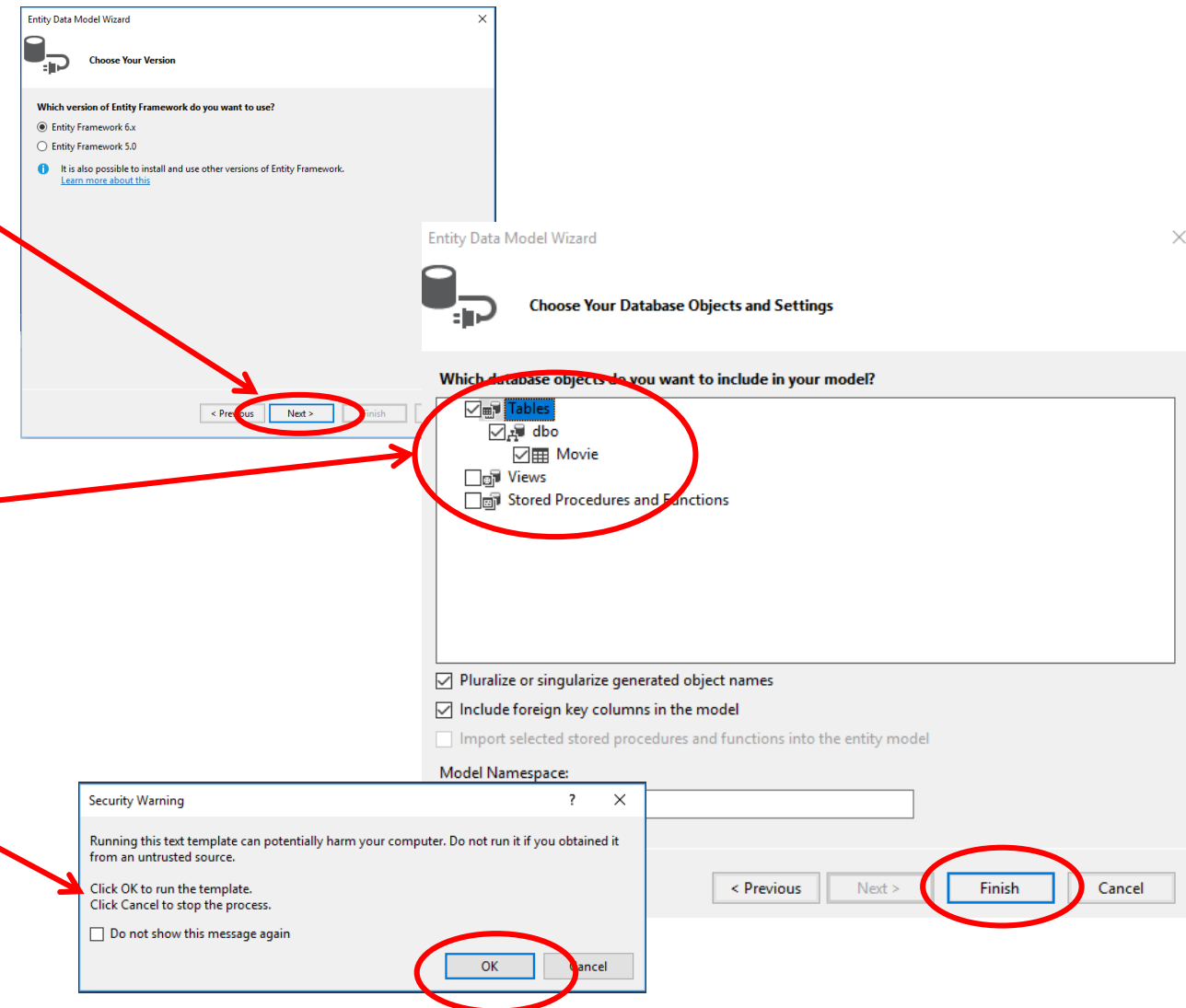
# Entity Framework

- ADO.NET should install this automatically, but it's worth checking the references to make sure
- If you need to install it, go to the Tools menu and select the NuGet Package Manager
  - Click Browse and type in Entity Framework
  - Select Entity Framework on the left of the screen
  - On the right of the screen, check the box next to your project
  - Click install and wait until you see a message in the Output window saying that the installation has been successful
- The NuGet Package Manager can also be used to install other useful packages, e.g. Bootstrap



# Adding an ADO.NET Entity Data Model

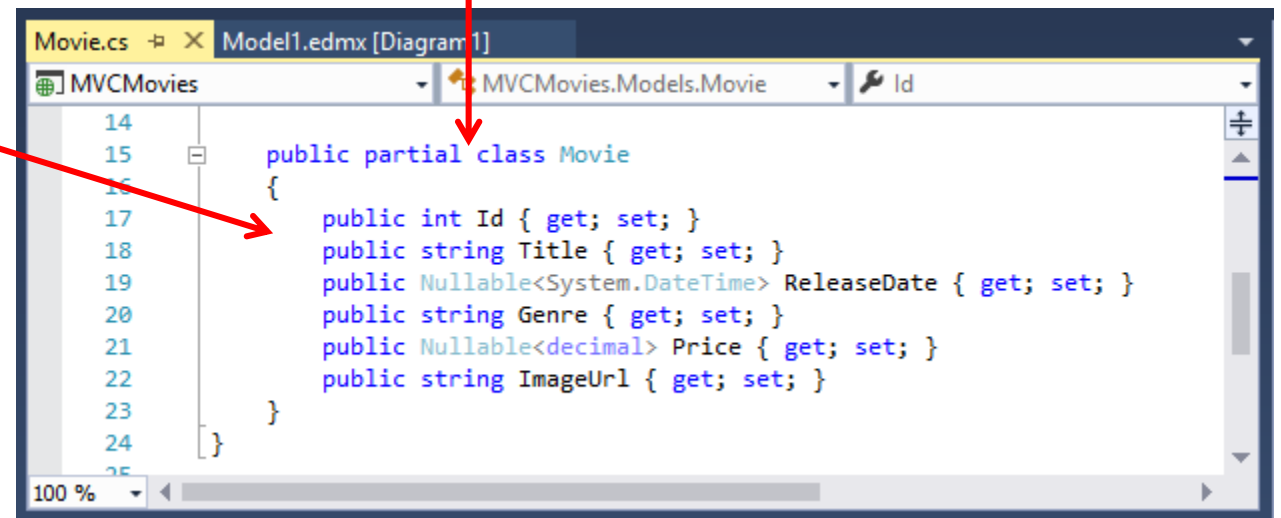
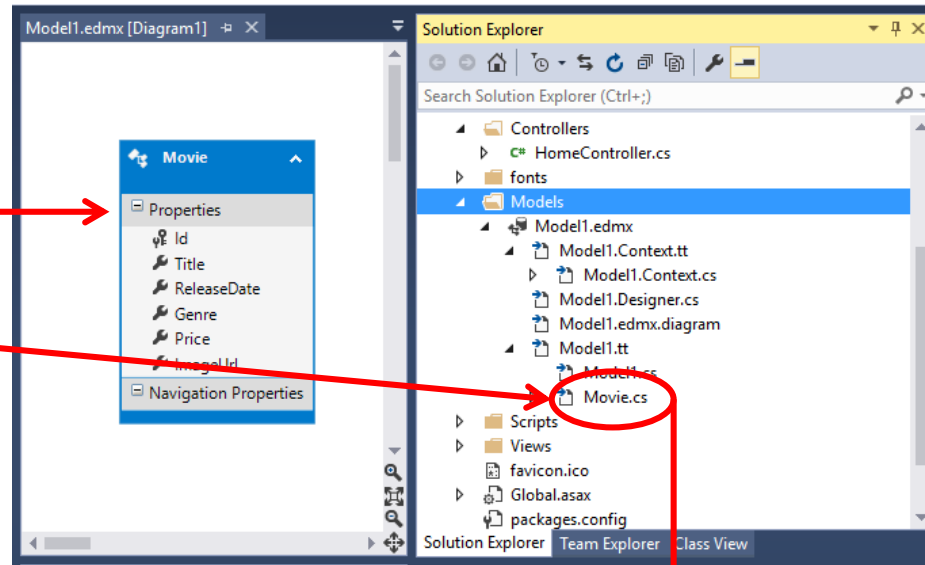
- The next screen is Choose Your Version
  - Leave this unchanged and click Next
- The next screen is Choose Your Database Objects and Settings
  - Please expand the Tables node until you see the names of the Movie table
  - Check the box next to the Movie table – if you don't do this, there will be no table in your db
  - Leave the other options unchanged and click Finish
- ADO.NET will now import your db
  - If you see any security warnings like this, just click OK





# Adding an ADO.NET Entity Data Model

- When the db has been added, you will see the db diagram showing the Movie table
- You will also see a class for the Movie table if you expand the nodes in the Models folder
  - The column names you set up in the database designer are now C# properties with the correct data types

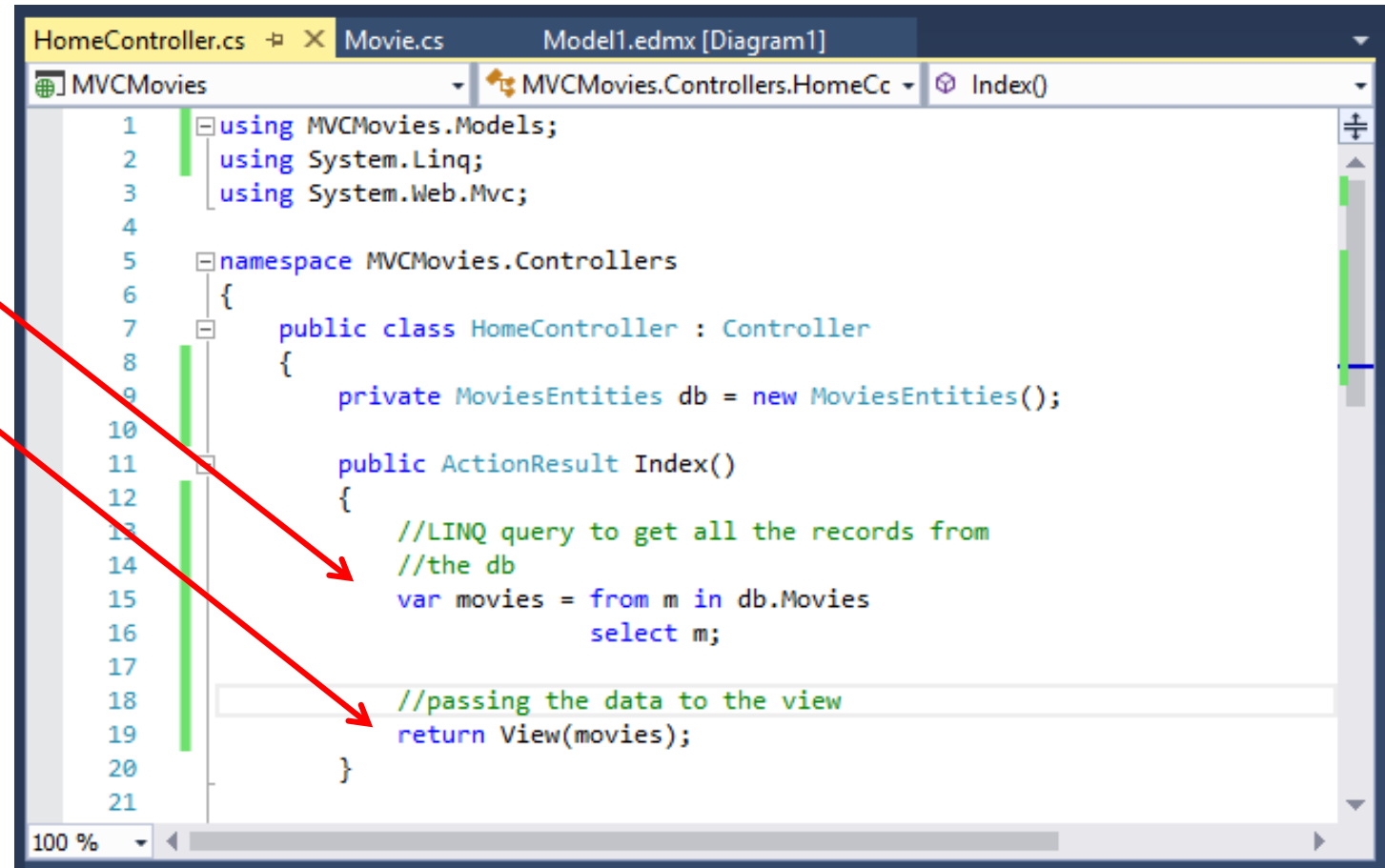


# Selecting the records from the database

- In the Index action method you can now write a LINQ query to get all the records from the database
- The results of the query should be passed to the View
- The application now has the db data, but we need a View to see it

The code in the Index action method is:

```
public ActionResult Index()
{
    var movies = from m in db.Movies
                  select m;
    return View(movies);
}
```

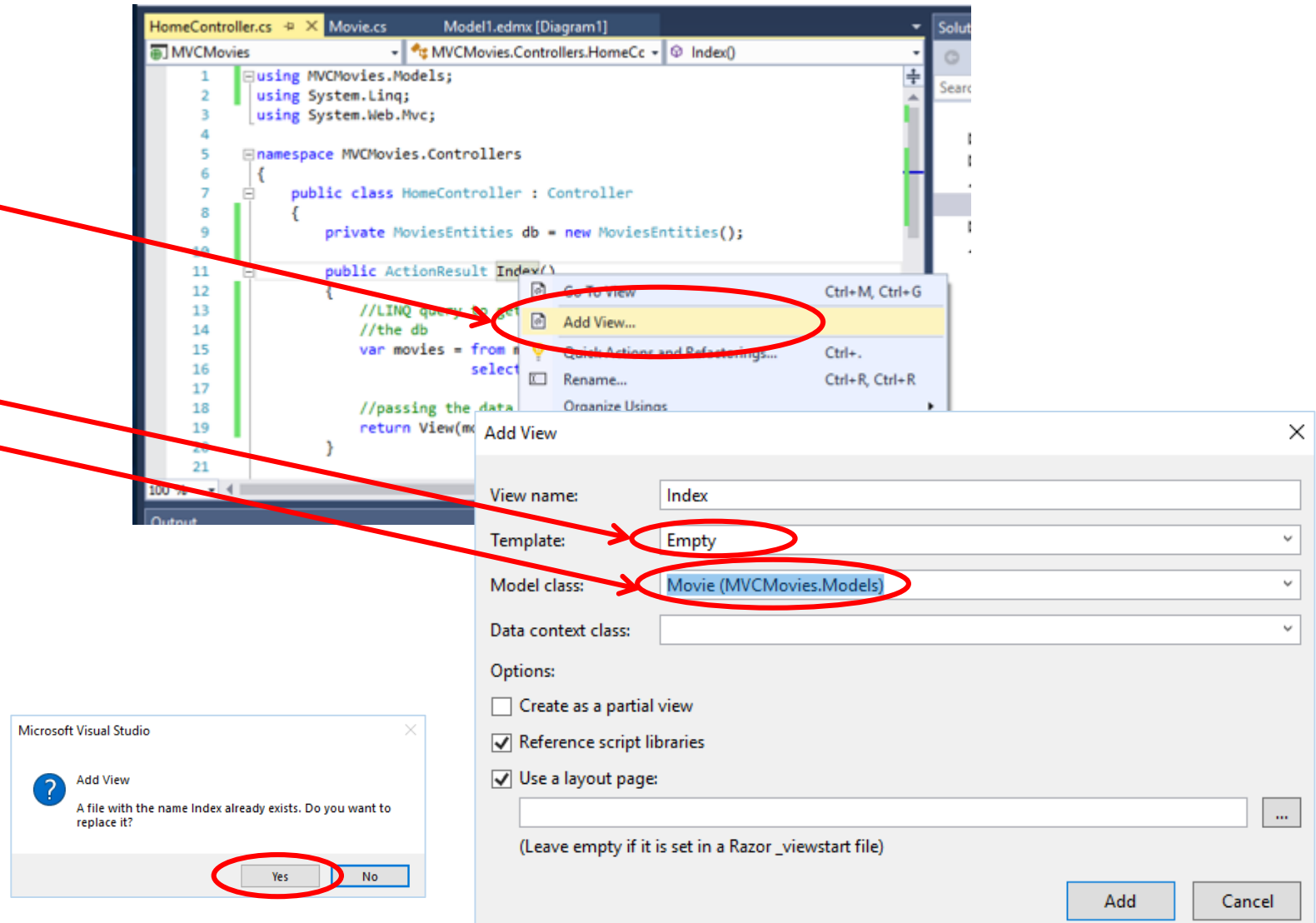


```
HomeController.cs Movie.cs Model1.edmx [Diagram1]
MVCMovies MVCMovies.Controllers.HomeCc Index()
1 using MVCMovies.Models;
2 using System.Linq;
3 using System.Web.Mvc;
4
5 namespace MVCMovies.Controllers
6 {
7     public class HomeController : Controller
8     {
9         private MoviesEntities db = new MoviesEntities();
10
11         public ActionResult Index()
12         {
13             //LINQ query to get all the records from
14             //the db
15             var movies = from m in db.Movies
16                           select m;
17
18             //passing the data to the view
19             return View(movies);
20         }
21     }
```

# The Views

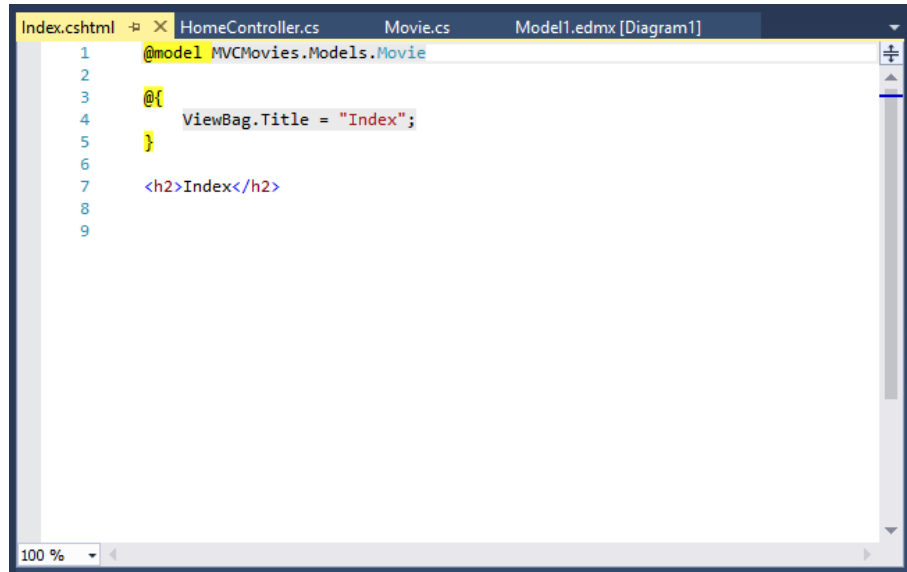
# Adding the Index View

- To add a View, right-click on the method signature of the Index action method and choose Add View from the menu
- In the Add View dialog, select:
  - Empty template
  - Movie as the model
  - Leave everything else unchanged and click Add
- If you get a message, asking if you want to replace the existing Index View, click Yes
  - This is the default Index View created by Visual Studio, which we don't need



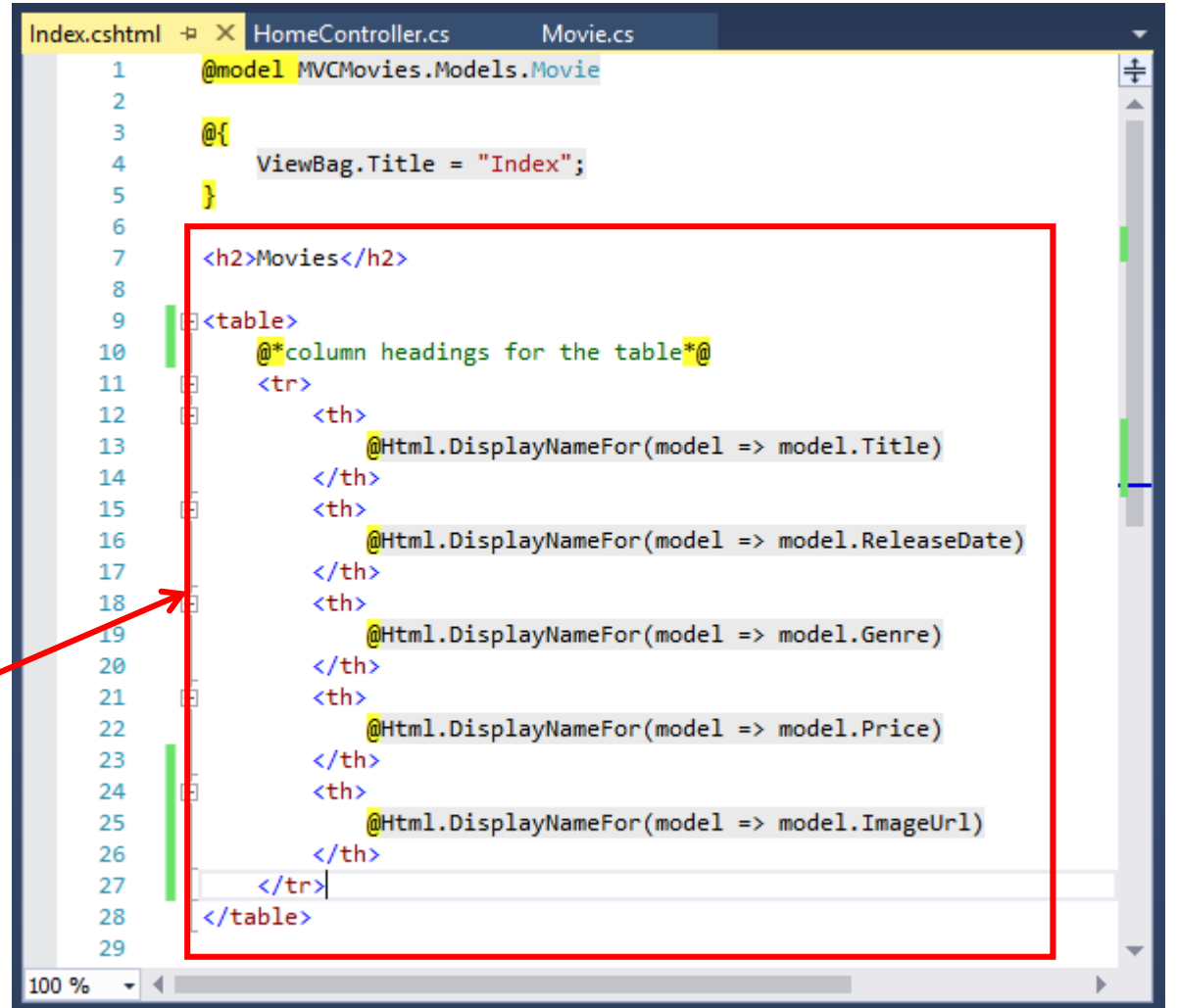
# Coding the Index View

- You will see a View with a few lines of C# and HTML



```
1 @model MVCMovies.Models.Movie
2
3 @{
4     ViewBag.Title = "Index";
5 }
6
7 <h2>Index</h2>
8
9
```

- Then add the code for the page and table headings (listed and explained on the next slide)



```
1 @model MVCMovies.Models.Movie
2
3 @{
4     ViewBag.Title = "Index";
5 }
6
7 <h2>Movies</h2>
8
9 <table>
10     @*column headings for the table*@
11     <tr>
12         <th>
13             @Html.DisplayNameFor(model => model.Title)
14         </th>
15         <th>
16             @Html.DisplayNameFor(model => model.ReleaseDate)
17         </th>
18         <th>
19             @Html.DisplayNameFor(model => model.Genre)
20         </th>
21         <th>
22             @Html.DisplayNameFor(model => model.Price)
23         </th>
24         <th>
25             @Html.DisplayNameFor(model => model.ImageUrl)
26         </th>
27     </tr>
28 </table>
29
```

# Coding the Index View

- You can see that the View is very like a page of HTML
  - You are creating an HTML table here
    - <table> is the tag for the table
    - <tr> is table row
    - <th> is table header
- All lines of C# start with a @, to tell the Razor View Engine that you have changed from HTML to C# (HTML tags tell Razor that you have changed back to HTML)
- @Html.DisplayNameFor(model => model.Title) is an HTML helper
  - HTML helpers are C# methods in MVC which take parameters and insert them into appropriate HTML for display in the browser

```
<h2>Movies</h2>
```

```
<table>
```

```
    @*column headings for the table*@
```

```
    <tr>
```

```
        <th>
```

```
            @Html.DisplayNameFor(model => model.Title)
```

```
        </th>
```

```
        <th>
```

```
            @Html.DisplayNameFor(model => model.ReleaseDate)
```

```
        </th>
```

```
        <th>
```

```
            @Html.DisplayNameFor(model => model.Genre)
```

```
        </th>
```

```
        <th>
```

```
            @Html.DisplayNameFor(model => model.Price)
```

```
        </th>
```

```
        <th>
```

```
            @Html.DisplayNameFor(model => model.ImageUrl)
```

```
        </th>
```

```
    </tr>
```

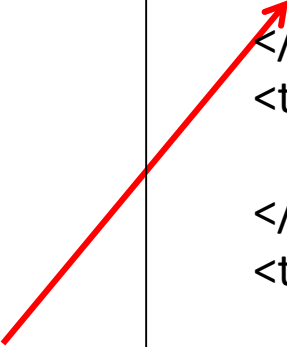
```
</table>
```

# Coding the Index View

- `@Html.DisplayNameFor(model => model.Title)` produces HTML when the application is run
  - You will be able to see this in a few slides' time when it's possible to run this code without an error
- `@Html.DisplayNameFor` is a strongly-typed HTML helper
  - It is used in strongly-typed views
  - By convention, strongly-typed HTML helpers have For in their names
  - Many take a lambda expression as one of their parameters
  - The field names in the lambda expression are from the Movie class, which was created when the database was attached to the project

```
<h2>Movies</h2>

<table>
  @*column headings for the table*@
  <tr>
    <th>
      @Html.DisplayNameFor(model => model.Title)
    </th>
    <th>
      @Html.DisplayNameFor(model => model.ReleaseDate)
    </th>
    <th>
      @Html.DisplayNameFor(model => model.Genre)
    </th>
    <th>
      @Html.DisplayNameFor(model => model.Price)
    </th>
    <th>
      @Html.DisplayNameFor(model => model.ImageUrl)
    </th>
  </tr>
</table>
```



# Coding the Index View

- `model => model.Title` is a lambda expression
  - They are a form of anonymous function, but it's a big topic and you don't need to understand them for now, as you will only be using existing examples, not creating them yourself
- `@* *@` encloses a comment
- If you run the code now, you will get an error
  - You will find out how to fix this on the next slide

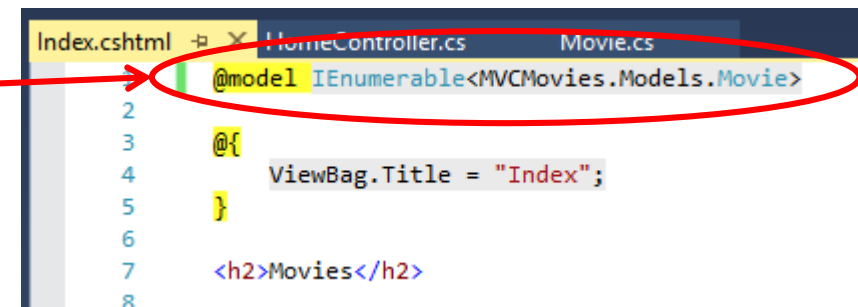
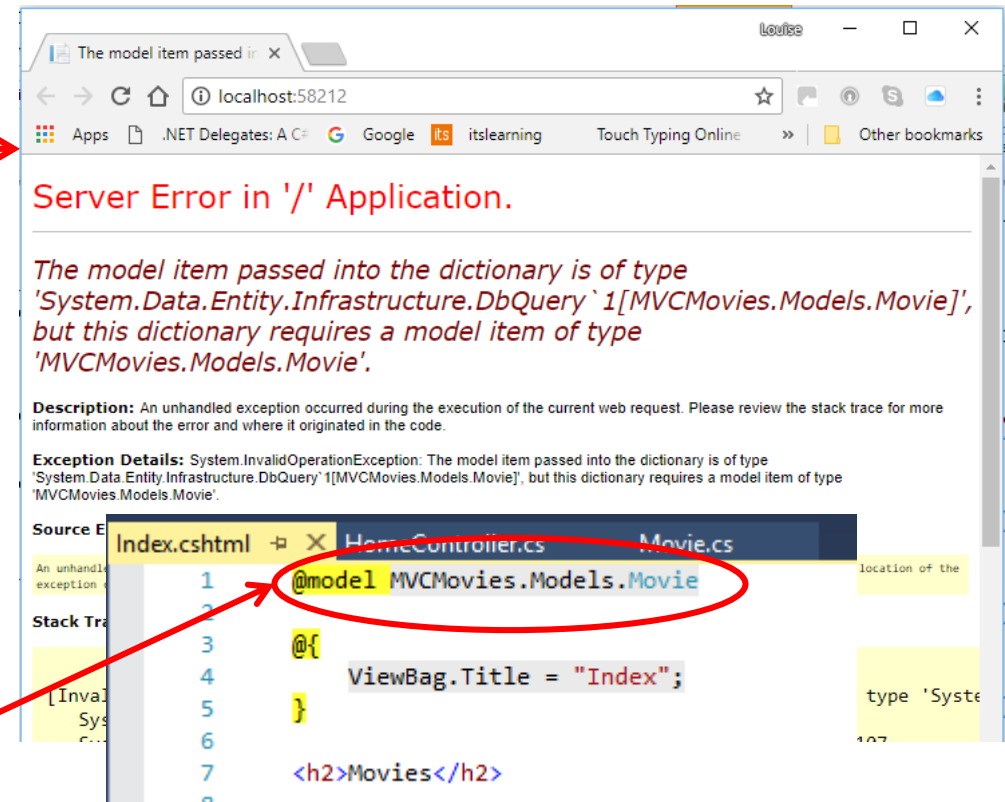
```
<h2>Movies</h2>

<table>
  @*column headings for the table*@
  <tr>
    <th>
      @Html.DisplayNameFor(model => model.Title)
    </th>
    <th>
      @Html.DisplayNameFor(model => model.ReleaseDate)
    </th>
    <th>
      @Html.DisplayNameFor(model => model.Genre)
    </th>
    <th>
      @Html.DisplayNameFor(model => model.Price)
    </th>
    <th>
      @Html.DisplayNameFor(model => model.ImageUrl)
    </th>
  </tr>
</table>
```



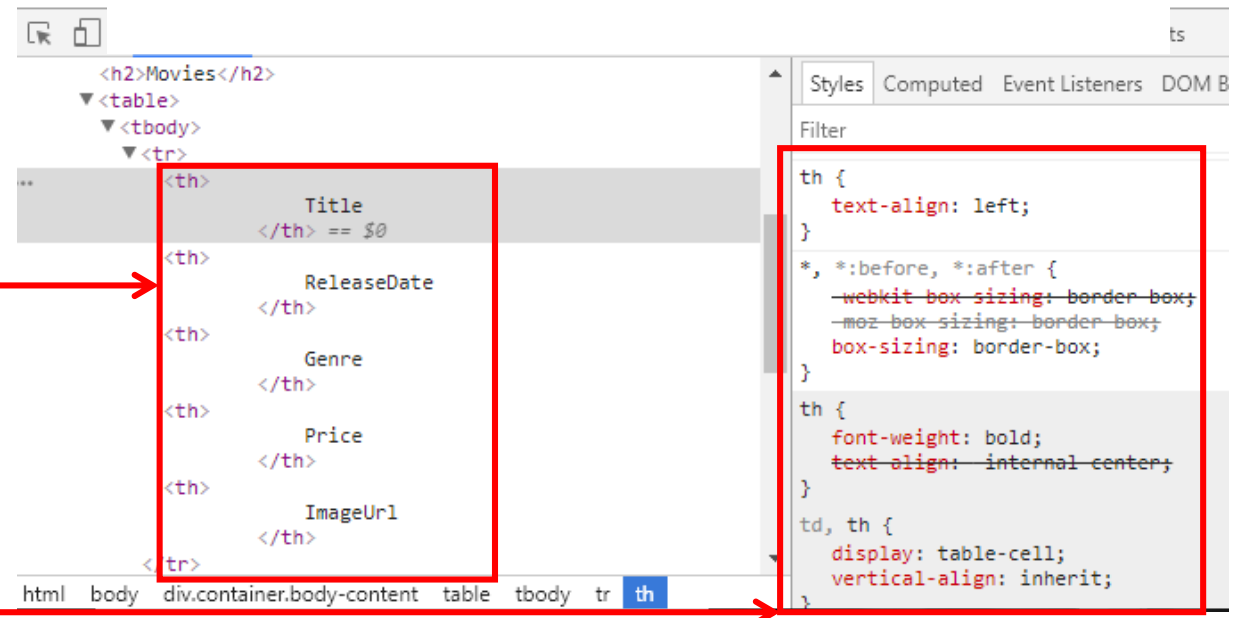
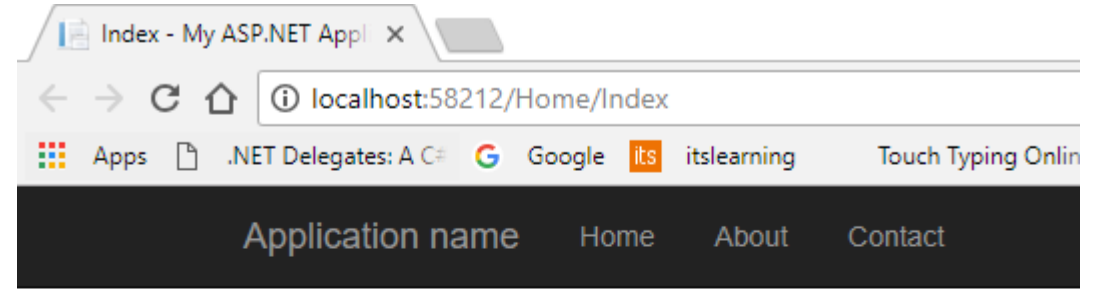
# Coding the Index View

- If you ran the code now, you would get this error
  - There is a mismatch between the type of the data that we passed in from the Index action method in the Home Controller and the model statement at the top of the Index view
  - This happens because we got some kind of list of data from the database query in the Index action method and passed this to the Index view
    - The Index view is expecting a single Movie object as its input, not any kind of list
    - The model statement at the top of the Index view makes this clear
      - @model MVCMovies.Models.Movie says that a single object is expected
- You want all the movies to be passed from the Index action method to the Index view, so you need to change the model statement to:
  - @model IEnumerable<MVCMovies.Models.Movie>
  - IEnumerable is an interface that all lists and collections inherit from, so this is an acceptable data type for passing in the Movies from the database query in the Index action method to the Index view



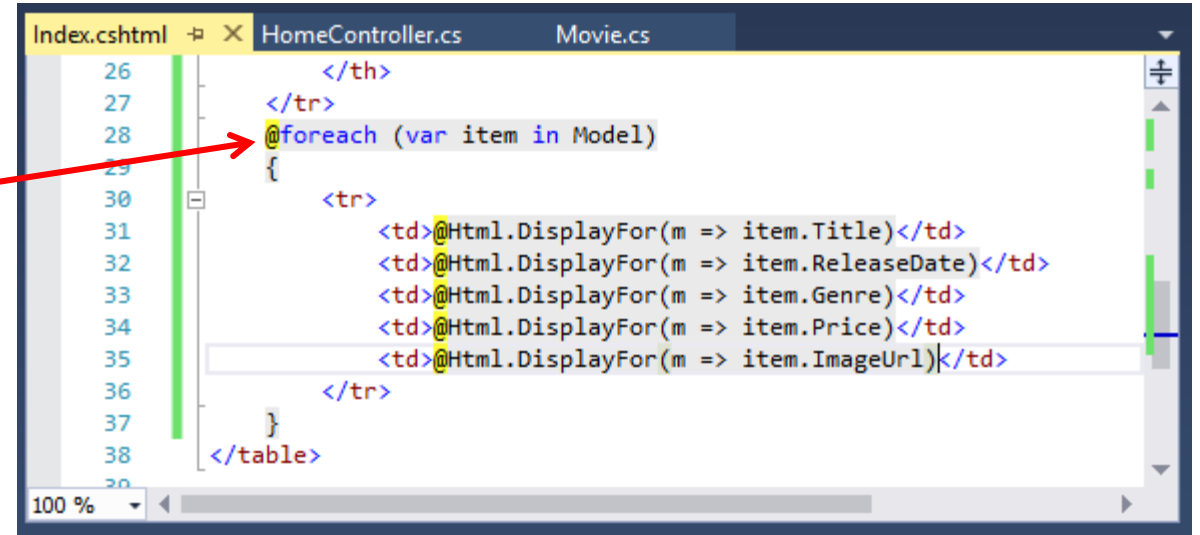
# Coding the Index View

- If you run the code now, you will see the page heading and the table headings
  - The table headings show that some data is being successfully retrieved from the database
  - It's not properly formatted, but that can be done later when everything is working
- You can also look in the developers' tools (F12 in most browsers) to see the HTML that the Html.DisplayFor helper produced
  - The HTML is quite simple – just the text of the column name – but there is quite also a bit of CSS added, such as text alignment and font weight



# Coding the Index View

- Now it's time to do the rest of the table
  - You need a foreach loop as there will be several rows in the table
    - The IEnumerable that was added to the model statement is also essential for making the foreach loop work
- Each iteration of the foreach loop creates one row of the table
  - <td> is the HTML tag for table data
  - Html.DisplayFor is the strongly-typed Html helper that displays the contents of a field from a class
    - It takes a lambda expression as a parameter and each lambda expression includes a field name from the Movie class



```
Index.cshtml | HomeController.cs | Movie.cs
26      </th>
27    </tr>
28    @foreach (var item in Model)
29    {
30      <tr>
31        <td>@Html.DisplayFor(m => item.Title)</td>
32        <td>@Html.DisplayFor(m => item.ReleaseDate)</td>
33        <td>@Html.DisplayFor(m => item.Genre)</td>
34        <td>@Html.DisplayFor(m => item.Price)</td>
35        <td>@Html.DisplayFor(m => item.ImageUrl)</td>
36      </tr>
37    }
38  </table>
```

```
@foreach (var item in Model)
{
    <tr>
        <td>@Html.DisplayFor(m => item.Title)</td>
        <td>@Html.DisplayFor(m => item.ReleaseDate)</td>
        <td>@Html.DisplayFor(m => item.Genre)</td>
        <td>@Html.DisplayFor(m => item.Price)</td>
        <td>@Html.DisplayFor(m => item.ImageUrl)</td>
    </tr>
}
```

# Coding the Index View

- If you run the code, you will now see the table, containing the data that you entered in SQL Server Management Studio
  - This means that the database is being queried correctly
  - The format is untidy, but this will be fixed later
  - At the moment there is no data for the imageUrl field, but this will be entered later
- The HTML that is produced by the Html.DisplayFor helper is straightforward – just the contents of the fields in the Movie class
  - The helper has allowed us to pass the data in to be rendered as HTML
  - You will see more powerful uses of HTML helpers later in the tutorial

Application name Home About Contact

## Movies

Title	ReleaseDate	Genre	Price	imageUrl
Groundhog Day	12/02/1993 00:00:00	Comedy	3.99	
Grosse Pointe Blank	11/04/1997 00:00:00	Comedy	4.99	
The Force Awakens	18/12/2015 00:00:00	Science Fiction	9.99	

```
<div class="navbar navbar-inverse navbar-fixed-top">...</div>
<div class="container body-content">
  ::before
  <h2>Movies</h2>
  <table>
    <tbody>
      <tr>...</tr>
      <tr>
        <td>Groundhog Day</td> == $0
        <td>12/02/1993 00:00:00</td>
        <td>Comedy</td>
        <td>3.99</td>
        <td></td>
      </tr>
      <tr>...</tr>
      <tr>...</tr>
    </tbody>
  </table>
  <hr>
  <footer>...</footer>
  ::after
</div>
```

html body div.container.body-content table tbody tr td

# Coding the Index View

- There are a few more things to do to finish the basic coding of the Index view:
  - Adding a link for creating a new record
  - Adding links for getting the details of a record or editing or deleting it
  - Putting the application name on the menu bar, browser tab and footer
- Link for creating a new record:
  - In the Index view, under `<h2>Movies</h2>`, add an `Html.ActionLink` helper
    - This creates an HTML anchor tag with the parameters entered
    - The first parameter is the text for the link and the second parameter is the action method that the link will go to (this hasn't been written yet)
    - Run the code and you will see the link under the Movies page heading
    - In developer tools, you can see that an HTML anchor tag has been created

```
7 <h2>Movies</h2>
8
9 @*link for creating a new record - the first parameter is the text for the link
10 and the second is the action method that it goes to*@
11 @Html.ActionLink("Create new movie", "Create")
12
13 <table>
14     @*column headings for the table*@
15     <tr>
16         <th>
```

```
@Html.ActionLink("Create new movie", "Create")
```

## Movies

[Create new movie](#)

Title	ReleaseDate
Groundhog Day	12/02/1993 00:(

```
<h2>Movies</h2>
<a href="/Home/Create">Create new movie</a>
<table>...</table>
```

# Coding the Index View

- Links for getting the details of a record or editing or deleting it
  - These will be added to each line of the table so it's clear which record is being edited, deleted, etc.
  - Add a new pair of `<td>` tags with 3 `Html.ActionLink` helpers inside them
  - Each `Html.ActionLink` helper has 3 parameters:
    - The text of the link
    - The action method the link will go to (these haven't been written yet)
    - The id of the record that will be edited, deleted, etc.
  - If you run the code now, you will see the 3 links next to each record in the table

```
@foreach (var item in Model)
{
    <tr>
        <td>@Html.DisplayFor(m => item.Title)</td>
        <td>@Html.DisplayFor(m => item.ReleaseDate)</td>
        <td>@Html.DisplayFor(m => item.Genre)</td>
        <td>@Html.DisplayFor(m => item.Price)</td>
        <td>@Html.DisplayFor(m => item.ImageUrl)</td>
        <td>
            @Html.ActionLink("Edit Record", "Edit", new { id = item.Id })
            @Html.ActionLink("Details of Record", "Details", new { id = item.Id })
            @Html.ActionLink("Delete Record", "Delete", new { id = item.Id })
        </td>
    </tr>
}
```

```
<td>
    @Html.ActionLink("Edit Record", "Edit", new { id = item.Id })
    @Html.ActionLink("Details of Record", "Details", new { id = item.Id })
    @Html.ActionLink("Delete Record", "Delete", new { id = item.Id })
</td>
```

## Movies

[Create new movie](#)

Title	ReleaseDate	Genre	Price	ImageUrl
Groundhog Day	12/02/1993 00:00:00	Comedy	3.99	<a href="#">Edit Record</a> <a href="#">Details of Record</a> <a href="#">Delete Record</a>
Grosse Pointe Blank	11/04/1997 00:00:00	Comedy	4.99	<a href="#">Edit Record</a> <a href="#">Details of Record</a> <a href="#">Delete Record</a>
The Force Awakens	18/12/2015 00:00:00	Science Fiction	9.99	<a href="#">Edit Record</a> <a href="#">Details of Record</a> <a href="#">Delete Record</a>



# Coding the Index View

- If you go into developer tools in the browser, you will see that the URL in the anchor tag also includes the record id
  - You will see that a different id has been retrieved for each record
  - If you hover over the link in the browser you can also see the URL for the link in the bottom left hand corner of the screen

Application name   Home   About   Contact

## Movies

[Create new movie](#)

Title	ReleaseDate	Genre	Price	ImageUrl
Groundhog Day	12/02/1993 00:00:00	Comedy	3.99	<a href="#">Edit Record</a> <a href="#">Details of Record</a> <a href="#">Delete Record</a>
Grosse Pointe Blank	11/04/1997 00:00:00	Comedy	4.99	<a href="#">Edit Record</a> <a href="#">Details of Record</a> <a href="#">Delete Record</a>
The Force Awakens	18/12/2015 00:00:00	Science Fiction	9.99	<a href="#">Edit Record</a> <a href="#">Details of Record</a> <a href="#">Delete Record</a>

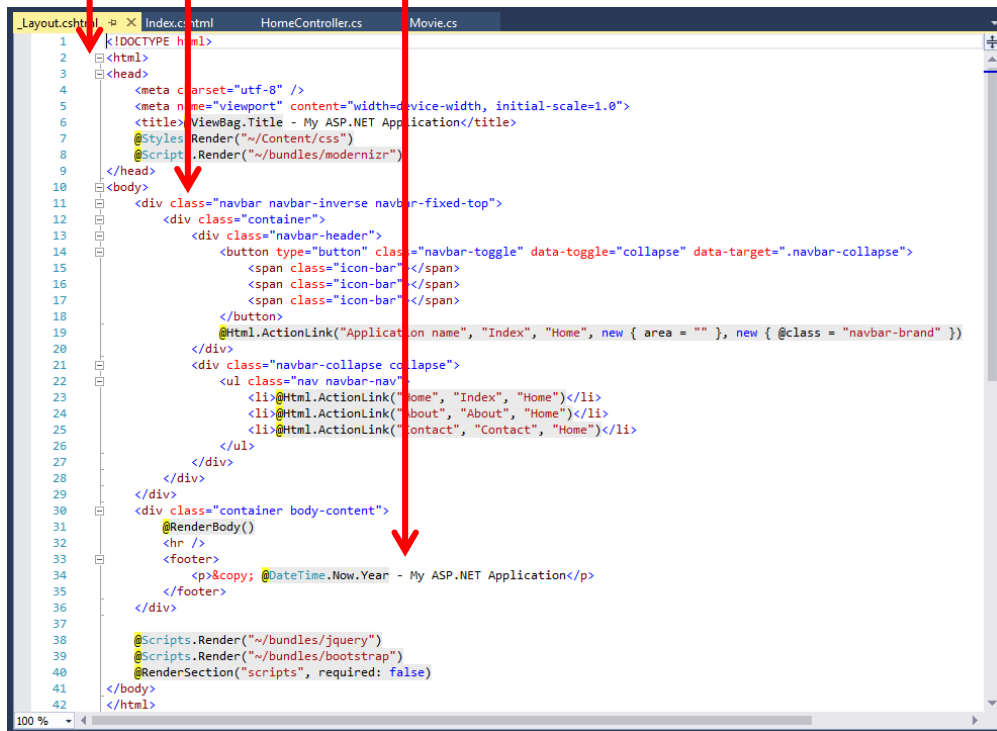
© 2018 - My ASP.NET Application

localhost:58212/Home/Edit/1

```
<tr>
  <td>Groundhog Day</td>
  <td>12/02/1993 00:00:00</td>
  <td>Comedy</td>
  <td>3.99</td>
  <td></td>
  <td>
    <a href="/Home/Edit/1">Edit Record</a> == $0
    <a href="/Home/Details/1">Details of Record</a>
    <a href="/Home/Delete/1">Delete Record</a>
  </td>
</tr>
<tr>
  <td>Grosse Pointe Blank</td>
  <td>11/04/1997 00:00:00</td>
  <td>Comedy</td>
  <td>4.99</td>
  <td></td>
  <td>
    <a href="/Home/Edit/2">Edit Record</a>
    <a href="/Home/Details/2">Details of Record</a>
    <a href="/Home/Delete/2">Delete Record</a>
  </td>
</tr>
<tr>
  <td>The Force Awakens</td>
  <td>18/12/2015 00:00:00</td>
  <td>Science Fiction</td>
  <td>9.99</td>
  <td></td>
  <td>
    <a href="/Home/Edit/3">Edit Record</a>
    <a href="/Home/Details/3">Details of Record</a>
    <a href="/Home/Delete/3">Delete Record</a>
  </td>
</tr>
```

# Coding the Index View

- The last part of the content for the Index view is putting the application name into the menu, browser tab and footer
- First, you need to know about the `_Layout.cshtml` file
- The code for the menu bar is in the `_Layout.cshtml` file in the Views/Shared folder
- The `_Layout` file contains any content that is shared between views, e.g.
  - HTML headers
  - Menus and footers

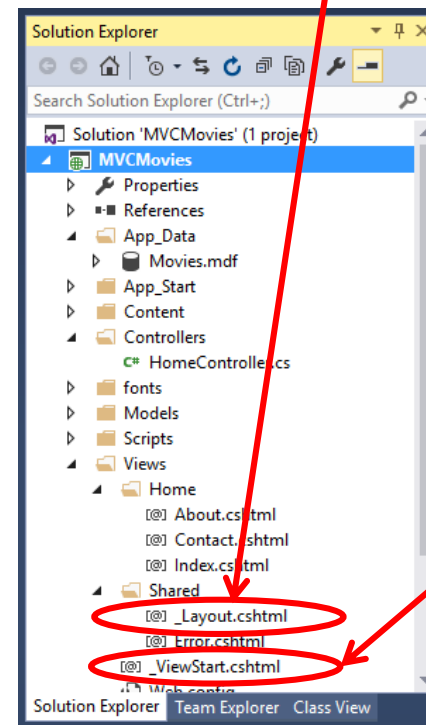


```
1 <!DOCTYPE html>
2 <html>
3 <head>
4     <meta charset="utf-8" />
5     <meta name="viewport" content="width=device-width, initial-scale=1.0">
6     <title>@ViewBag.Title - My ASP.NET Application</title>
7     @Styles.Render("~/Content/css")
8     @Scripts.Render("~/bundles/modernizr")
9 </head>
10 <body>
11     <div class="navbar navbar-inverse navbar-fixed-top">
12         <div class="container">
13             <div class="navbar-header">
14                 <button type="button" class="navbar-toggle" data-toggle="collapse" data-target=".navbar-collapse">
15                     <span class="icon-bar"></span>
16                     <span class="icon-bar"></span>
17                     <span class="icon-bar"></span>
18                 </button>
19                 @Html.ActionLink("Application name", "Index", "Home", new { area = "" }, new { @class = "navbar-brand" })
20             </div>
21             <div class="navbar-collapse collapse">
22                 <ul class="nav navbar-nav">
23                     <li>@Html.ActionLink("Home", "Index", "Home")</li>
24                     <li>@Html.ActionLink("About", "About", "Home")</li>
25                     <li>@Html.ActionLink("Contact", "Contact", "Home")</li>
26                 </ul>
27             </div>
28         </div>
29     </div>
30     <div class="container body-content">
31         @RenderBody()
32     </div>
33     <hr />
34     <p>©&copy; @DateTime.Now.Year - My ASP.NET Application</p>
35 </body>
36 </html>
37
38 @Scripts.Render("~/bundles/jquery")
39 @Scripts.Render("~/bundles/bootstrap")
40 @RenderSection("scripts", required: false)
41
42 </html>
```

- Any view can use the `_Layout` file
  - They can reference it by adding:  
Layout =  
"`~/Views/Shared/_Layout.cshtml`"  
in the View

```
@{
    Layout = "~/Views/Shared/_Layout.cshtml";
}
```

- If there is a `_ViewStart.cshtml` file in the Views folder, this contains the reference to `_Layout.cshtml` and views will pick this up by default, so the reference doesn't have to be added to each view

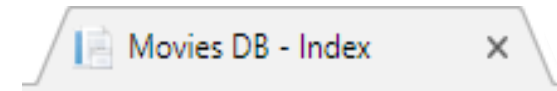




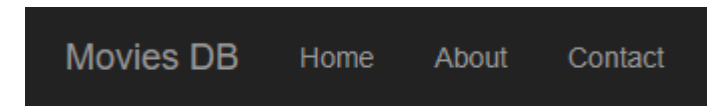
# Coding the Index View

- The last part of the content for the Index view is putting the application name into the menu, browser tab and footer
- In the `_Layout.cshtml` file:
  - In the HTML `<head>` section, change `<title>` to:  
`<title>Movies DB - @ViewBag.Title</title>`
  - In the HTML `<body>` section, in the `<div class=navbar-header>`, change the first parameter in the `Html.Action` link:  
`@Html.ActionLink("Movies DB", "Index", ...`
  - In the `<footer>` section change the `<p>` content to:  
`<p>&copy; @DateTime.Now.Year - Movies DB Ltd.</p>`
- Run the code to see the changes

```
<head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width,
  <title>Movies DB - @ViewBag.Title</title>
  @Styles.Render("~/Content/css")
  @Scripts.Render("~/bundles/modernizr")
</head>
```



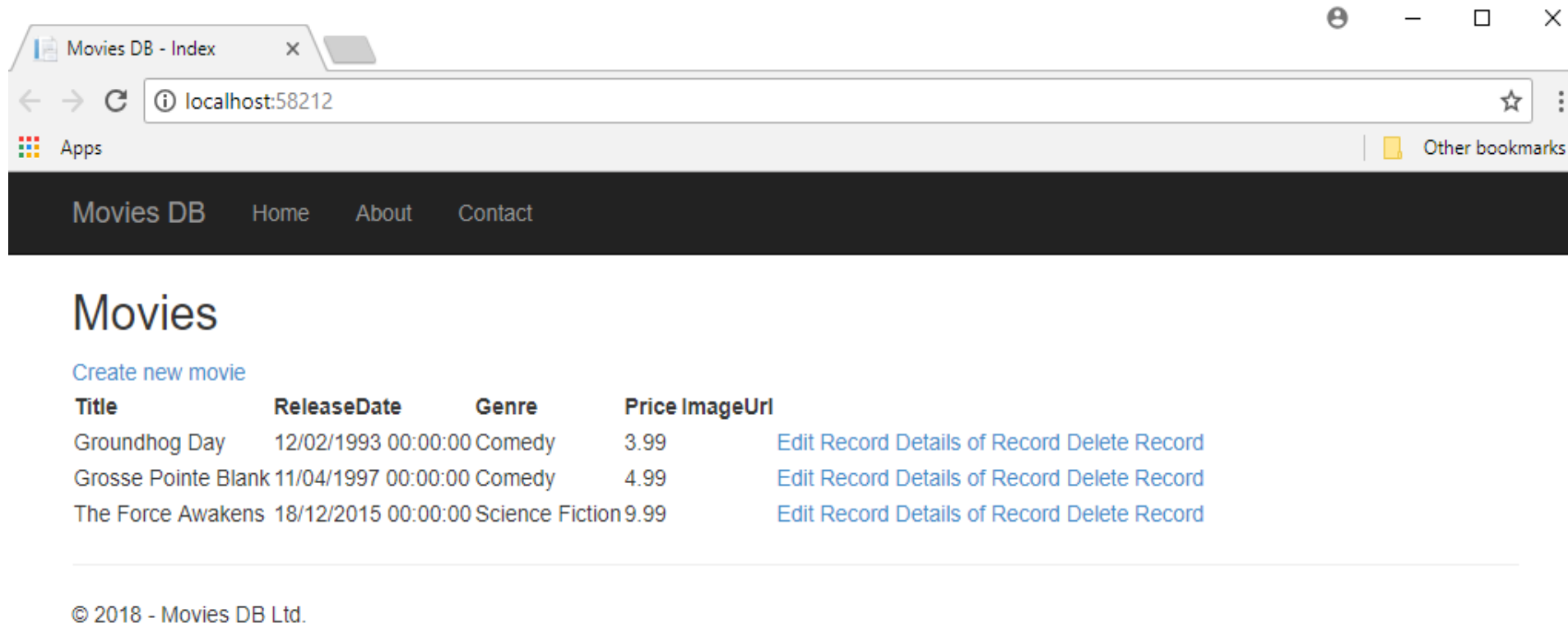
```
<div class="navbar-header">
  <button type="button" class="navbar-toggle"
    <span class="icon-bar"></span>
    <span class="icon-bar"></span>
    <span class="icon-bar"></span>
  </button>
  @Html.ActionLink("Movies DB", "Index", '
</div>
```



```
<footer>
  <p>&copy; @DateTime.Now.Year - Movies DB Ltd.</p>
</footer>
```

# The finished (for now) Index view

- The content and functionality of the Index view are complete (styling and images will be done later) and it should look something like this:



- The records in the database have been correctly retrieved and displayed
- The links are in place. They don't work at the moment – following any of them will cause an error – so the next thing is to write the action methods that the links should lead to – Create, Edit, Details and Delete

**Create**

# Coding the Create view

- For users to add records to the database, you will need a dialog that allows the information for each movie record to be added
- This means starting with an action method, which will then call a view to display the dialog
- Go to the Home Controller and add a Create action method below the Index action method using the code on the right
  - This is a very simple action method that just returns a view
- Then right-click on the action method signature and create a strongly typed view using the Movie model
  - Make sure that 'Use a layout page' is selected

```
public ActionResult Create()
{
    return View();
}
```

Add View

View name: Create

Template: Empty

Model class: Movie (MVCMovies.Models)

Data context class:

Options:

☐ Create as a partial view

☒ Reference script libraries

☒ Use a layout page:

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

Add Cancel

# Coding the Create view

- To allow users to add information through the view and pass it to the controller, a form is needed
  - This is done with an `Html.BeginForm` helper
    - By default, this returns information to an action method with the same name as the view and uses the POST method of communication
  - The using statement means that any resources (e.g. memory) associated with the form are disposed of as soon as they are no longer needed
  - The form must have a submit button and this can be an HTML button
- Two other HTML helpers are used inside the form:
  - `Html.LabelFor` is a strongly-typed helper that formats column headers as labels
  - `Html.EditorFor` is a strongly-typed helper that accepts user input and assigns it to the corresponding fields in the model class (`Movie`)
- There is also a link to return to the Index page

```
@using (Html.BeginForm())  
{  
    <h4>Movie</h4>  
  
    @Html.LabelFor(m => m.Title)  
    @Html.EditorFor(m => m.Title)  
  
    @Html.LabelFor(m => m.ReleaseDate)  
    @Html.EditorFor(m => m.ReleaseDate)  
  
    @Html.LabelFor(m => m.Genre)  
    @Html.EditorFor(m => m.Genre)  
  
    @Html.LabelFor(m => m.Price)  
    @Html.EditorFor(m => m.Price)  
  
    @Html.LabelFor(m => m.ImageUrl)  
    @Html.EditorFor(m => m.ImageUrl)  
  
    <input type="submit" value="Create Movie" />  
}  
  
@Html.ActionLink("Back to List", "Index")
```

# Coding the Create view

- Now try entering some data into the Create view and clicking 'Create Movie'
  - Nothing is saved and the input fields go blank
- At the moment, there is no code in the Home Controller that can save data, so you should do that next
- Under the Create action method, add the code on the right
  - The `HttpPost` annotation at the top of the action method tells MVC to use the HTTP POST request method to communicate with the server when executing this action method
  - It's fine to have two methods with the same name and return type because they have different parameters
    - This is called overloading
  - The data that was entered in the Create view is passed in the `Movie` object into the action method
  - `db.Movies.Add(movie)` says that the movie should be added to the database the next time database changes are requested

```
[HttpPost]
public ActionResult Create(Movie movie)
{
    db.Movies.Add(movie);
    db.SaveChanges();
    return RedirectToAction("Index");
}
```

- `db.SaveChanges()` sends any pending database changes to the server
- The return statement redirects users to a different action method and view (`Index`)
  - The user is mostly like to want to see all the records after creating one

# Coding the Create view

- Run the code and you should now be able to add a record:

[Movies DB](#) [Home](#) [About](#) [Contact](#)

## Create

Movie

**Title**  **ReleaseDate**  **Genre**  **Price**  **ImageUrl**

[Back to List](#)

[Movies DB](#) [Home](#) [About](#) [Contact](#)

## Movies

[Create new movie](#)

Title	ReleaseDate	Genre	Price	ImageUrl
Groundhog Day	12/02/1993 00:00:00	Comedy	3.99	<a href="#">Edit Record</a> <a href="#">Details of Record</a> <a href="#">Delete Record</a>
Grosse Pointe Blank	11/04/1997 00:00:00	Comedy	4.99	<a href="#">Edit Record</a> <a href="#">Details of Record</a> <a href="#">Delete Record</a>
The Force Awakens	18/12/2015 00:00:00	Science Fiction	9.99	<a href="#">Edit Record</a> <a href="#">Details of Record</a> <a href="#">Delete Record</a>
Hidden Figures	25/12/2016 00:00:00	Biographical	12.99	<a href="#">Edit Record</a> <a href="#">Details of Record</a> <a href="#">Delete Record</a>

- The Create view is finished (for now)
- User input validation and styling will be added later

**Edit**



# Coding the Edit view

- The Edit view will allow users to edit records. It is quite similar to the Create view
- You start by adding an Edit action method in the Home Controller, containing the code on the right:
  - This action method receives a record id as an input parameter (passed by the `Html.ActionLink("Edit Record", "Edit", new {id = item.Id})` code in the Index view
  - It uses the record id to look up the record in the database
  - It sends the data to the Edit view
- Right-click on the method signature and add a strongly-typed view with Movie as the model

```
public ActionResult Edit(int id)
{
    Movie movie = db.Movies.Find(id);
    return View(movie);
}
```

Add View

View name: Edit

Template: Empty

Model class: Movie (MVCMovies.Models)

Data context class:

Options:

☐ Create as a partial view

☒ Reference script libraries

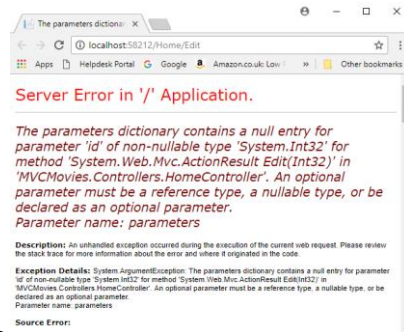
☒ Use a layout page:

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

Add Cancel

# Coding the Edit view

- To allow users to edit information in the view and pass it to the controller, add the code on the right
- It is almost identical to the code in the Create view, so it's easiest to copy that
  - The only difference is that the text on the button is "Save Changes"
- If you run the code with the Edit view visible on screen, you will get an error, as the URL doesn't have a record id



- Two workarounds are:
  - Enter a valid record id in the URL
  - Run from the Index page, so that the URL always has an id
- There is also a fix for the problem – setting a start page

```
@using (Html.BeginForm())
```

```
{
```

```
<h4>Movie</h4>
```

```
@Html.LabelFor(m => m.Title)
```

```
@Html.EditorFor(m => m.Title)
```

```
@Html.LabelFor(m => m.ReleaseDate)
```

```
@Html.EditorFor(m => m.ReleaseDate)
```

```
@Html.LabelFor(m => m.Genre)
```

```
@Html.EditorFor(m => m.Genre)
```

```
@Html.LabelFor(m => m.Price)
```

```
@Html.EditorFor(m => m.Price)
```

```
@Html.LabelFor(m => m.ImageUrl)
```

```
@Html.EditorFor(m => m.ImageUrl)
```

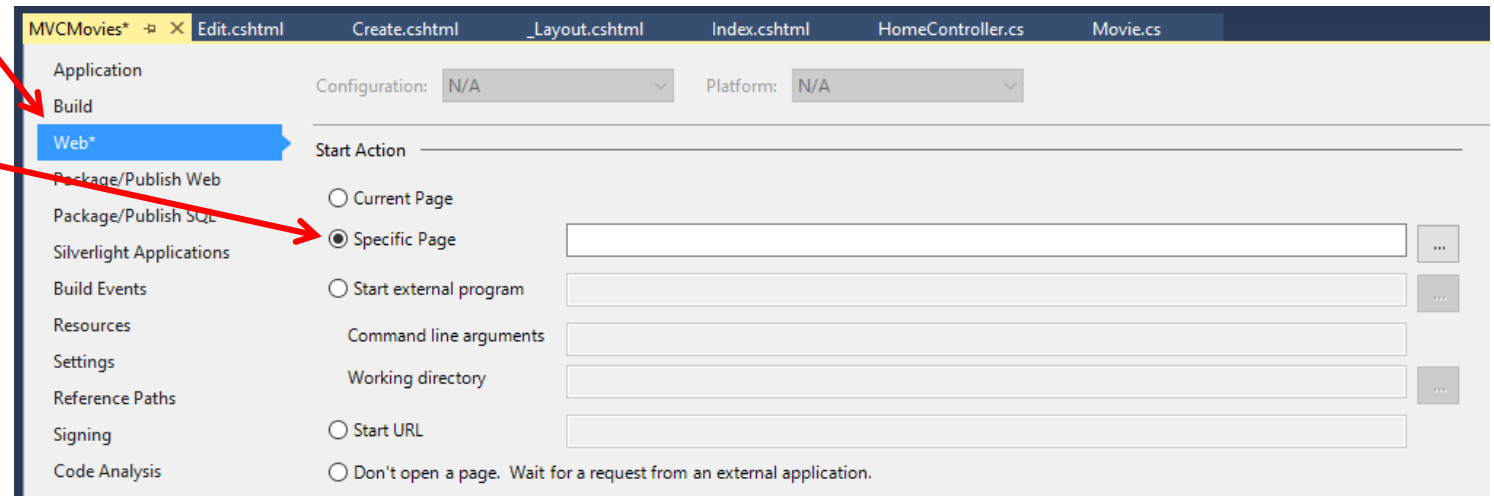
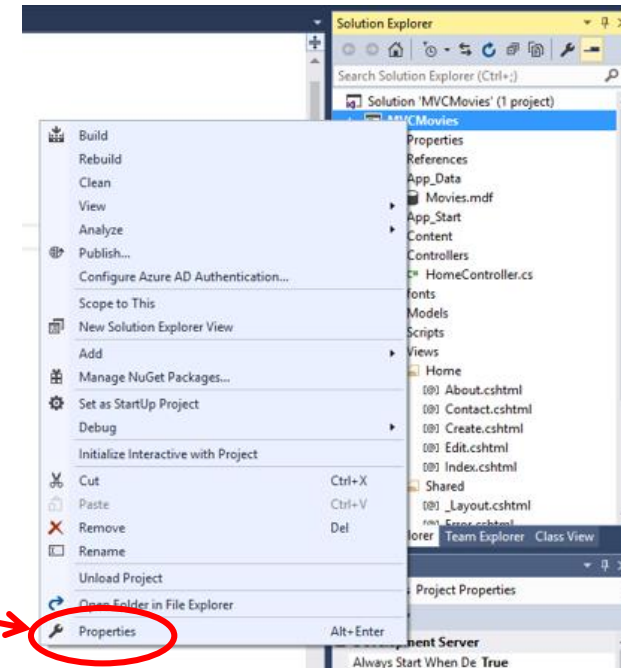
```
<input type="submit" value="Save Changes" />
```

```
}
```

```
@Html.ActionLink("Back to List", "Index")
```

# Setting a start page

- Setting a start page will fix the problem of crashes caused by running the code from the wrong page
- In the Solution Explorer right-click on the project name (the one in bold)
  - From the menu select Properties
  - When the Properties dialog opens
    - Select Web from the list on the left
    - Set the start action as Specific Page
      - Leave the input box blank, as it will use MVC routing and default to Home/Index
    - Save the change (Ctrl+S)
- Now the project will always start from the Index page



# Coding the Edit view

- Run the code and it will now start with the Index view. If you navigate to the Edit view, you should see something like this:

localhost:58212/Home/Edit/1

Apps | Other

Movies DB Home About Contact

## Edit

Movie

Title  ReleaseDate  Genre  Price

[Back to List](#)

---

© 2018 - Movies DB Ltd.

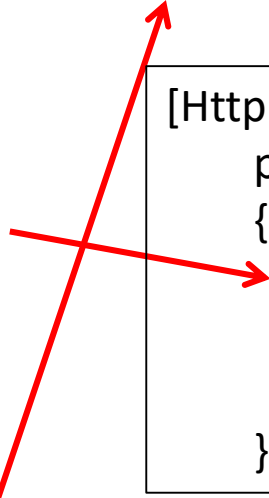
- If you look at the URL, you can see the record id
- Data for that record has been retrieved from the database, ready for editing
- If you try to make a change, it won't work yet, as there isn't any code for changing records in the database
- The HTML that has been created by each helper is the same as in the Create View

# Coding the Edit view

- A POST action method for editing is needed in the Home Controller to allow edits to be saved
  - The code for this is on the right
  - It's similar to the POST action method for Create, except for the line:
    - `db.Entry(movie).State = EntityState.Modified`
    - This modifies the existing record with the changes that were passed into the action method by the form in the view
    - You will also need to add:
      - `using System.Data.Entity` at the top of the Home Controller
- Now you should be able to edit a record

```
using System.Data.Entity;
```

```
[HttpPost]
public ActionResult Edit(Movie movie)
{
    db.Entry(movie).State = EntityState.Modified;
    db.SaveChanges();
    return RedirectToAction("Index");
}
```



# Coding the Edit view

- Run the code and you should now be able to edit a record:



- The Edit view is finished (for now)
- User input validation and styling will be added later

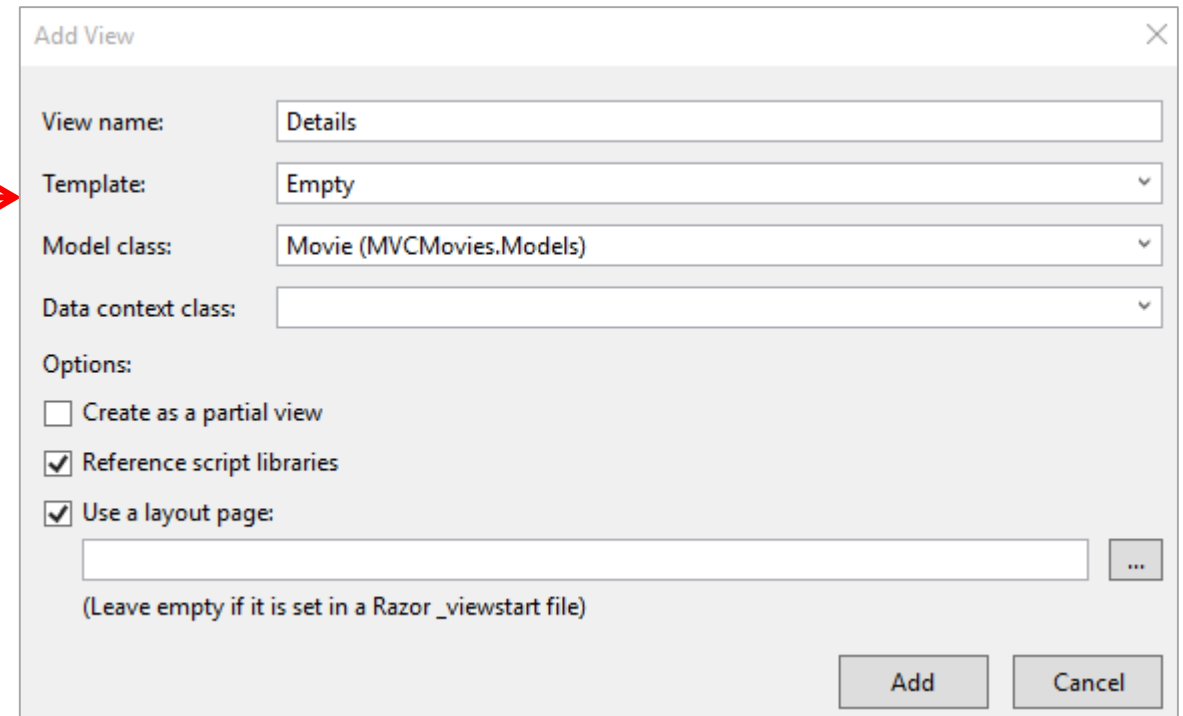
# Details

# Coding the Details view

- It would be helpful if users could look at the details of movies, so you will create a Details view next
- Start in the Home Controller and add a Details action method, using the code on the right
  - It's very similar to the GET Edit action method, because it's doing the same thing – accepting a record id, fetching the record from the database and passing the record data to a view
- Then right-click on the method signature and add a strongly-typed view called Details with Movie as the model

```
public ActionResult Details(int id)
{
    Movie movie = db.Movies.Find(id);

    return View(movie);
}
```



Add View

View name: Details

Template: Empty

Model class: Movie (MVCMovies.Models)

Data context class:

Options:

☐ Create as a partial view

☒ Reference script libraries

☒ Use a layout page:

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

Add Cancel



# Coding the Details view


- The Details view simply displays the details of the movie and doesn't allow editing
- Please add the code on the right to the Details view, just below the `<h2>Details</h2>` heading
- There are some new HTML tags here:
  - `<dl>` - description list
  - `<dt>` - description term (i.e. title of the data item in the list)
  - `<dd>` - description (i.e. the content of each field)
  - The `<dt>` and `<dd>` tags don't do much, but you can apply CSS to style your list
- All the HTML helpers are familiar, as they were used in the Index view
- Since there is no user input, there is no need for a POST action method in the Home Controller

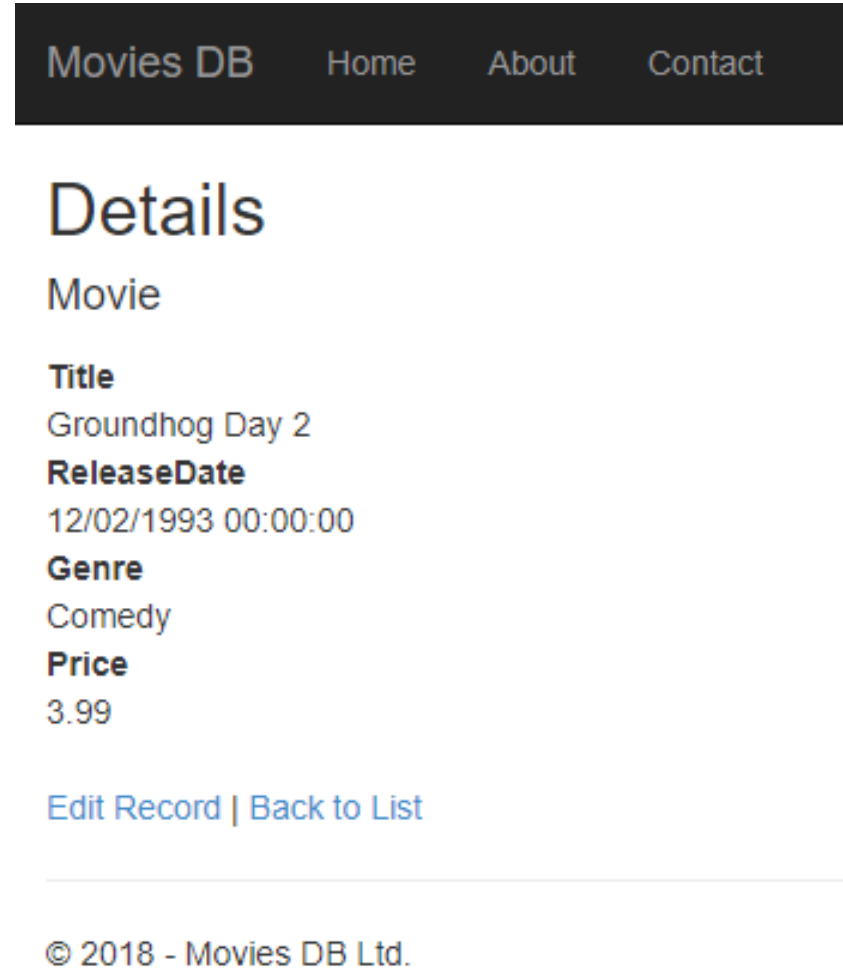


```
<h4>Movie</h4>

<dl>
  <dt>@Html.DisplayNameFor(m => m.Title)</dt>
  <dd>@Html.DisplayFor(m => m.Title)</dd>
  <dt>@Html.DisplayNameFor(m => m.ReleaseDate)</dt>
  <dd>@Html.DisplayFor(m => m.ReleaseDate)</dd>
  <dt>@Html.DisplayNameFor(m => m.Genre)</dt>
  <dd>@Html.DisplayFor(m => m.Genre)</dd>
  <dt>@Html.DisplayNameFor(m => m.Price)</dt>
  <dd>@Html.DisplayFor(m => m.Price)</dd>
</dl>
<p>
  @Html.ActionLink("Edit Record", "Edit", new { id = Model.Id }) |
  @Html.ActionLink("Back to List", "Index")
</p>
```

# Coding the Details view

- Run the code and select Details of Record in the Index view. You should now be able to see the details of a record: 
- This Details view isn't very exciting, because it just repeats the information in the Index view, but in your own projects you can make it more interesting, e.g. with extra information, pictures or videos
- The Details view is finished (for now)
- Styling will be added later



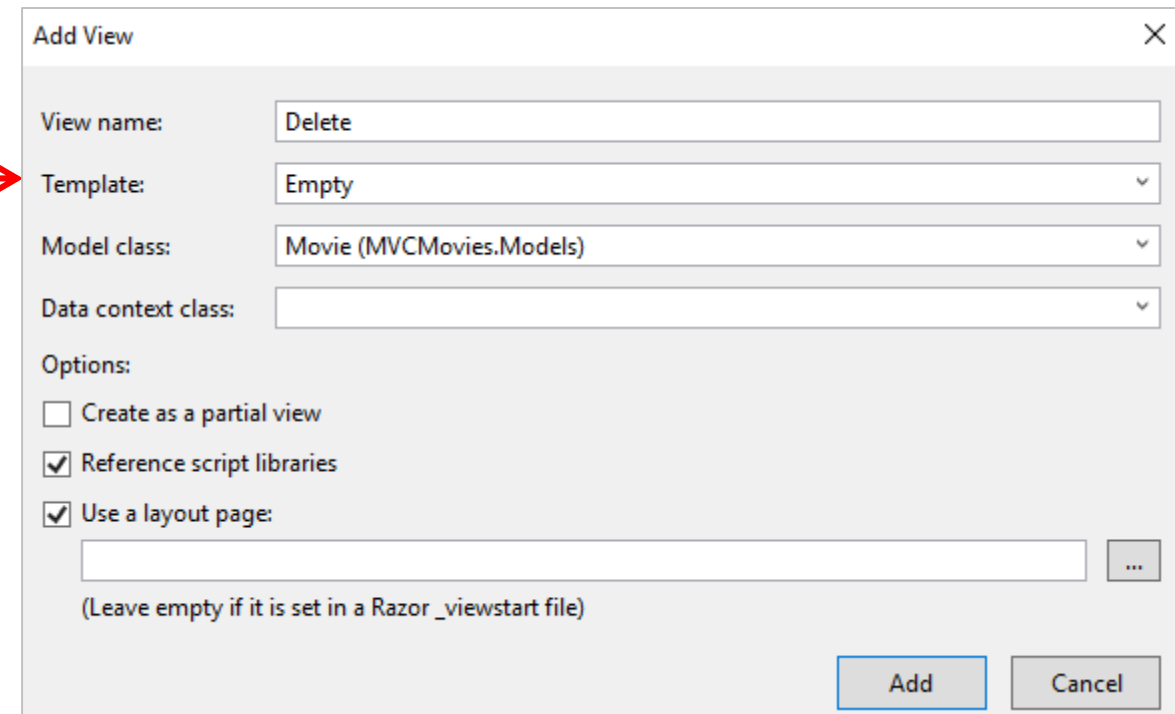
**Delete**

# Coding the Delete view

- It would be helpful if users could delete a movie, so you will create a Delete view next
- Start in the Home Controller and add a Delete action method, using the code on the right
  - It's very similar to the GET Edit and Details action methods, because it's doing the same thing – accepting a record id, fetching the record from the database and passing the record data to a view
- Then right-click on the method signature and add a strongly-typed view called Delete with Movie as the model

```
public ActionResult Delete(int id)
{
    Movie movie = db.Movies.Find(id);

    return View(movie);
}
```



Add View

View name: Delete

Template: Empty

Model class: Movie (MVCMovies.Models)

Data context class:

Options:

☐ Create as a partial view

☒ Reference script libraries

☒ Use a layout page:

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

Add Cancel

# Coding the Delete view

- The Delete view offers the user the option of deleting the record
  - It also displays the details of the movie but doesn't allow editing
    - That part of the code is identical to the Details view, so can be copied from there
  - There is also a form which passes the id of the record to the Home Controller for deletion
    - There is no need to pass the data, as the record is not being added or changed
- Please add the code on the right to the Delete view, just below the `<h2>Delete</h2>` heading
- All the HTML tags and helpers are familiar, as they were used in the Index or Details views

```
<h3>Are you sure you want to delete this?</h3>
<h4>Movie</h4>
<dl>
  <dt>@Html.DisplayNameFor(m => m.Title)</dt>
  <dd>@Html.DisplayFor(m => m.Title)</dd>
  <dt>@Html.DisplayNameFor(m => m.ReleaseDate)</dt>
  <dd>@Html.DisplayFor(m => m.ReleaseDate)</dd>
  <dt>@Html.DisplayNameFor(m => m.Genre)</dt>
  <dd>@Html.DisplayFor(m => m.Genre)</dd>
  <dt>@Html.DisplayNameFor(m => m.Price)</dt>
  <dd>@Html.DisplayFor(m => m.Price)</dd>
</dl>
@using (Html.BeginForm())
{
  <input type="submit" value="Delete Record" />
}
@Html.ActionLink("Back to List", "Index")
```

# Coding the Delete view

- Run the code and select Delete Record in the Index view. You should now be able to see the details of a record:
- Deletion won't work yet, as you need to return to the Home Controller and write a POST Delete action method to implement it



[Movies DB](#) [Home](#) [About](#) [Contact](#)

## Delete

Are you sure you want to delete this?

Movie

**Title**  
Groundhog Day 2

**ReleaseDate**  
12/02/1993 00:00:00

**Genre**  
Comedy

**Price**  
3.99

[Delete Record](#)

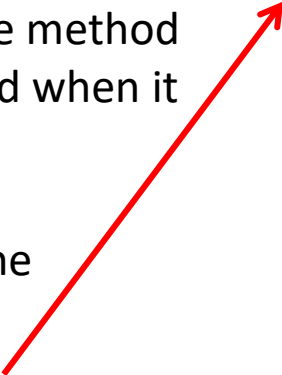
[Back to List](#)

---

© 2018 - Movies DB Ltd.

# Coding the Delete view

- In the Home Controller, add an action method called Delete with an integer parameter called id
  - You will see that there is a problem, because there is already an action method with the same signature
  - MVC offers a way around this:
    - Change the name of the second Delete action method to something else (e.g. DeleteConfirmed) and add an annotation [ActionName("Delete")] above the method
    - This makes sure that MVC can call this method when it is looking for a Delete action method
- Next, add HttpPost inside the square brackets for the annotation, as this is a POST method that will change the contents of the database
- Then, add the rest of the code on the right. This code
  - Uses the record id to find the record
  - Marks it for deletion
  - Saves changes to the database
  - Returns the user to the Index view



```
[HttpPost, ActionName("Delete")]
public ActionResult DeleteConfirmed(int id)
{
    Movie movie = db.Movies.Find(id);
    db.Movies.Remove(movie);
    db.SaveChanges();
    return RedirectToAction("Index");
}
```

# Coding the Delete view

- Now delete a record
  - Check the Index view to confirm that it has gone

Movies DB Home About Contact

## Delete

Are you sure you want to delete this?

Movie

**Title**

Groundhog Day 2

**ReleaseDate**

12/02/1993 00:00:00

**Genre**

Comedy

**Price**

3.99

Delete Record

[Back to List](#)

© 2018 - Movies DB Ltd.

Movies DB Home About Contact

## Movies

[Create new movie](#)

Title	ReleaseDate	Genre	Price	ImageUrl	
Grosse Pointe Blank	11/04/1997 00:00:00	Comedy	4.99		<a href="#">Edit Record</a> <a href="#">Details of Record</a> <a href="#">Delete Record</a>
The Force Awakens	18/12/2015 00:00:00	Science Fiction	9.99		<a href="#">Edit Record</a> <a href="#">Details of Record</a> <a href="#">Delete Record</a>
Hidden Figures	25/12/2016 00:00:00	Biographical	12.99		<a href="#">Edit Record</a> <a href="#">Details of Record</a> <a href="#">Delete Record</a>

© 2018 - Movies DB Ltd.



**CRUD**

# Your CRUD database is working

- You now have a working database application in MVC that has CRUD functionality. It can display, create, edit and delete records
- In the second part of this tutorial, you will learn how to add user input validation and filtering, do some styling and render URLs as images

Movies DB

Home

About

Contact

Movies

Create new movie

Title	ReleaseDate	Genre	Price	ImageUrl
Grosse Pointe Blank	11/04/1997 00:00:00	Comedy	4.99	<a href="#">Edit Record</a> <a href="#">Details of Record</a> <a href="#">Delete Record</a>
The Force Awakens	18/12/2015 00:00:00	Science Fiction	9.99	<a href="#">Edit Record</a> <a href="#">Details of Record</a> <a href="#">Delete Record</a>
Hidden Figures	25/12/2016 00:00:00	Biographical	12.99	<a href="#">Edit Record</a> <a href="#">Details of Record</a> <a href="#">Delete Record</a>

© 2018 - Movies DB Ltd.