**Adding Search by Genre**

Next, you'll add a feature to let users search for movies by genre. Replace the Indexmethod with the following code:

C#Copy

public ActionResult Index(string movieGenre, string searchString)

{

var GenreLst = new List<string>();

var GenreQry = from d in db.Movies

orderby d.Genre

select d.Genre;

GenreLst.AddRange(GenreQry.Distinct());

ViewBag.movieGenre = new SelectList(GenreLst);

var movies = from m in db.Movies

select m;

if (!String.IsNullOrEmpty(searchString))

{

movies = movies.Where(s => s.Title.Contains(searchString));

}

if (!string.IsNullOrEmpty(movieGenre))

{

movies = movies.Where(x => x.Genre == movieGenre);

}

return View(movies);

}

This version of the Index method takes an additional parameter, namely movieGenre. The first few lines of code create a List object to hold movie genres from the database.

The following code is a LINQ query that retrieves all the genres from the database.

C#Copy

var GenreQry = from d in db.Movies

                   orderby d.Genre

                   select d.Genre;

The code uses the AddRange method of the generic List collection to add all the distinct genres to the list. (Without the Distinct modifier, duplicate genres would be added — for example, comedy would be added twice in our sample). The code then stores the list of genres in the ViewBag.MovieGenre object. Storing category data (such a movie genre's) as a [SelectList](https://msdn.microsoft.cus/library/system.web.mvc.selectlist(v=vs.108).aspx) object in a ViewBag, then accessing the category data in a dropdown list box is a typical approach for MVC applications.

The following code shows how to check the movieGenre parameter. If it's not empty, the code further constrains the movies query to limit the selected movies to the specified genre.

C#Copy

if (!string.IsNullOrEmpty(movieGenre))

{

movies = movies.Where(x => x.Genre == movieGenre);

}

As stated previously, the query is not run on the data base until the movie list is iterated over (which happens in the View, after the Index action method returns).

**Adding Markup to the Index View to Support Search by Genre**

Add an Html.DropDownList helper to the *Views\Movies\Index.cshtml* file, just before the TextBox helper. The completed markup is shown below:

CSHTMLCopy

@model IEnumerable<MvcMovie.Models.Movie>

@{

ViewBag.Title = "Index";

}

<h2>Index</h2>

<p>

@Html.ActionLink("Create New", "Create")

@using (Html.BeginForm("Index", "Movies", FormMethod.Get))

{

<p>

Genre: @Html.DropDownList("movieGenre", "All")

Title: @Html.TextBox("SearchString")

<input type="submit" value="Filter" />

</p>

}

</p>

<table class="table">

In the following code:

CSHTMLCopy

@Html.DropDownList("movieGenre", "All")

The parameter "MovieGenre" provides the key for the DropDownList helper to find a IEnumerable<SelectListItem> in the ViewBag. The ViewBag was populated in the action method:

C#Copy

public ActionResult Index(string movieGenre, string searchString)

{

var GenreLst = new List<string>();

var GenreQry = from d in db.Movies

orderby d.Genre

select d.Genre;

GenreLst.AddRange(GenreQry.Distinct());

ViewBag.movieGenre = new SelectList(GenreLst);

var movies = from m in db.Movies

select m;

if (!String.IsNullOrEmpty(searchString))

{

movies = movies.Where(s => s.Title.Contains(searchString));

}

if (!string.IsNullOrEmpty(movieGenre))

{

movies = movies.Where(x => x.Genre == movieGenre);

}

return View(movies);

}

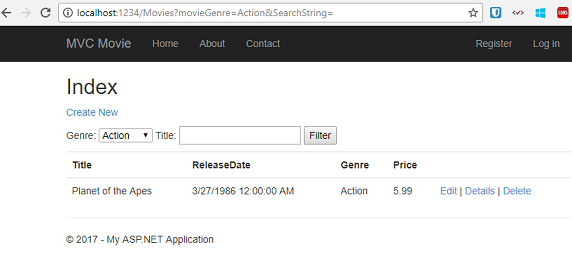
The parameter "All" provides an option label. If you inspect that choice in your browser, you'll see that its "value" attribute is empty. Since our controller only filters if the string is not null or empty, submitting an empty value for movieGenre shows all genres.

You can also set an option to be selected by default. If you wanted "Comedy" as your default option, you would change the code in the Controller like so:

CSHTMLCopy

ViewBag.movieGenre = new SelectList(GenreLst, "Comedy");

Run the application and browse to */Movies/Index*. Try a search by genre, by movie name, and by both criteria.



In this section you created a search action method and view that let users search by movie title and genre. In the next section, you'll look at how to add a property to the Movie model and how to add an initializer that will automatically create a test database.

**Adding a New Field**

* 10/17/2013
* 10 minutes to read
* Contributors
  + [Rick Anderson](https://github.com/Rick-Anderson)

* + [Scott Addie](https://github.com/scottaddie)

* + [Daya Adianto](https://github.com/addianto)

* + [Luke Latham](https://github.com/guardrex)

* + [Platon Efimov](https://github.com/SemperPeritus)

* + [all](https://github.com/aspnet/Docs/blob/master/aspnet/mvc/overview/getting-started/introduction/adding-a-new-field.md)

by [Rick Anderson](https://github.com/Rick-Anderson)

**Note**

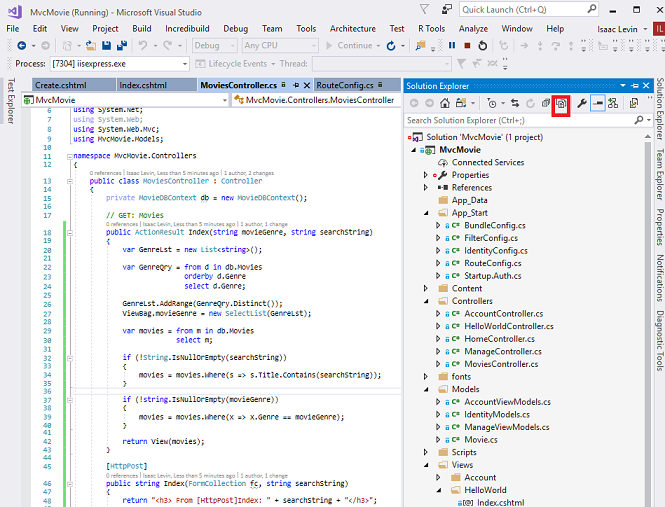
This document is part of the [**Getting Started with ASP.NET MVC 5**](https://docs.microsoft.com/en-us/aspnet/mvc/overview/getting-started/introduction/getting-started) tutorial. Final Source for tutorial located on [**GitHub**](https://github.com/aspnet/Docs/tree/master/aspnet/mvc/overview/getting-started/introduction/sample/MvcMovie/MvcMovie)

In this section you'll use Entity Framework Code First Migrations to migrate some changes to the model classes so the change is applied to the database.

By default, when you use Entity Framework Code First to automatically create a database, as you did earlier in this tutorial, Code First adds a table to the database to help track whether the schema of the database is in sync with the model classes it was generated from. If they aren't in sync, the Entity Framework throws an error. This makes it easier to track down issues at development time that you might otherwise only find (by obscure errors) at run time.

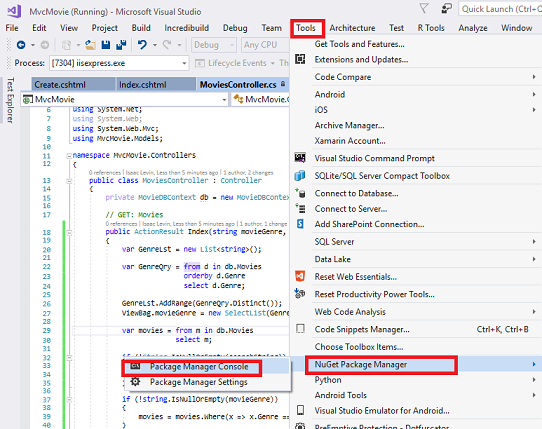
**Setting up Code First Migrations for Model Changes**

Navigate to Solution Explorer. Right click on the *Movies.mdf* file and select **Delete** to remove the movies database. If you don't see the *Movies.mdf* file, click on the **Show All Files** icon shown below in the red outline.



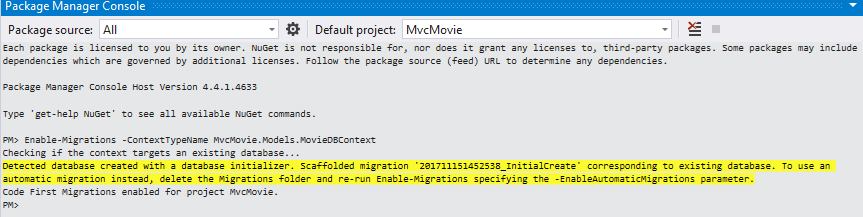
Build the application to make sure there are no errors.

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

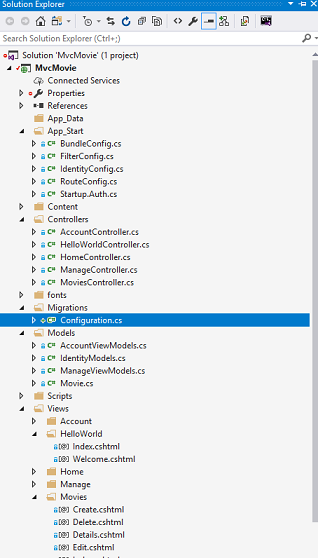


In the **Package Manager Console** window at the PM> prompt enter

Enable-Migrations -ContextTypeName MvcMovie.Models.MovieDBContext



The **Enable-Migrations** command (shown above) creates a *Configuration.cs* file in a new *Migrations* folder.



Visual Studio opens the *Configuration.cs* file. Replace the Seed method in the *Configuration.cs* file with the following code:

C#Copy

protected override void Seed(MvcMovie.Models.MovieDBContext context)

{

context.Movies.AddOrUpdate( i => i.Title,

new Movie

{

Title = "When Harry Met Sally",

ReleaseDate = DateTime.Parse("1989-1-11"),

Genre = "Romantic Comedy",

Price = 7.99M

},

new Movie

{

Title = "Ghostbusters ",

ReleaseDate = DateTime.Parse("1984-3-13"),

Genre = "Comedy",

Price = 8.99M

},

new Movie

{

Title = "Ghostbusters 2",

ReleaseDate = DateTime.Parse("1986-2-23"),

Genre = "Comedy",

Price = 9.99M

},

new Movie

{

Title = "Rio Bravo",

ReleaseDate = DateTime.Parse("1959-4-15"),

Genre = "Western",

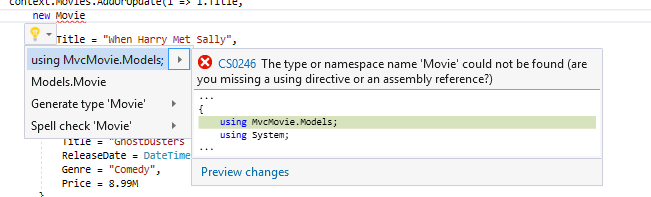
Price = 3.99M

}

);

}

Hover over the red squiggly line under Movie and click Show Potential Fixes and then click **using** **MvcMovie.Models;**



Doing so adds the following using statement:

C#Copy

using MvcMovie.Models;

**Note**

Code First Migrations calls the Seed method after every migration (that is, calling **update-database** in the Package Manager Console), and this method updates rows that have already been inserted, or inserts them if they don't exist yet.

The [**AddOrUpdate**](https://msdn.microsoft.com/library/system.data.entity.migrations.idbsetextensions.addorupdate(v=vs.103).aspx) method in the following code performs an "upsert" operation:

C#Copy

context.Movies.AddOrUpdate(i => i.Title,

new Movie

{

Title = "When Harry Met Sally",

ReleaseDate = DateTime.Parse("1989-1-11"),

Genre = "Romantic Comedy",

Rating = "PG",

Price = 7.99M

}

Because the [**Seed**](https://msdn.microsoft.com/library/hh829453(v=vs.103).aspx) method runs with every migration, you can't just insert data, because the rows you are trying to add will already be there after the first migration that creates the database. The "[**upsert**](http://en.wikipedia.org/wiki/Upsert)" operation prevents errors that would happen if you try to insert a row that already exists, but it overrides any changes to data that you may have made while testing the application. With test data in some tables you might not want that to happen: in some cases when you change data while testing you want your changes to remain after database updates. In that case you want to do a conditional insert operation: insert a row only if it doesn't already exist.

The first parameter passed to the [**AddOrUpdate**](https://msdn.microsoft.com/library/system.data.entity.migrations.idbsetextensions.addorupdate(v=vs.103).aspx) method specifies the property to use to check if a row already exists. For the test movie data that you are providing, the Title property can be used for this purpose since each title in the list is unique:

C#Copy

context.Movies.AddOrUpdate(i => i.Title,

This code assumes that titles are unique. If you manually add a duplicate title, you'll get the following exception the next time you perform a migration.

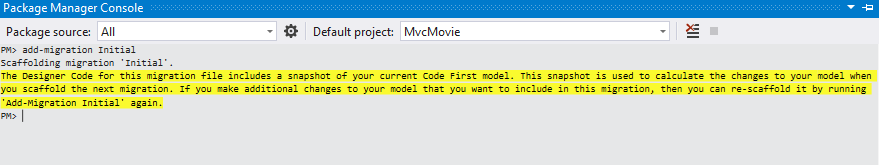
*Sequence contains more than one element*

For more information about the [**AddOrUpdate**](https://msdn.microsoft.com/library/system.data.entity.migrations.idbsetextensions.addorupdate(v=vs.103).aspx) method, see [**Take care with EF 4.3 AddOrUpdate Method**](http://thedatafarm.com/blog/data-access/take-care-with-ef-4-3-addorupdate-method/)..

**Press CTRL-SHIFT-B to build the project.**(The following steps will fail if you don't build at this point.)

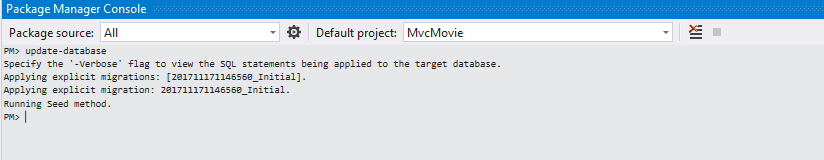
The next step is to create a DbMigration class for the initial migration. This migration creates a new database, that's why you deleted the *movie.mdf* file in a previous step.

In the **Package Manager Console** window, enter the command add-migration Initial to create the initial migration. The name "Initial" is arbitrary and is used to name the migration file created.



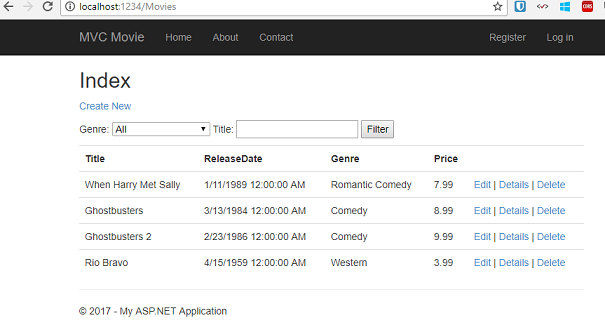
Code First Migrations creates another class file in the *Migrations* folder (with the name *{DateStamp}\_Initial.cs* ), and this class contains code that creates the database schema. The migration filename is pre-fixed with a timestamp to help with ordering. Examine the *{DateStamp}\_Initial.cs* file, it contains the instructions to create the Movies table for the Movie DB. When you update the database in the instructions below, this *{DateStamp}\_Initial.cs* file will run and create the DB schema. Then the **Seed** method will run to populate the DB with test data.

In the **Package Manager Console**, enter the command update-database to create the database and run the Seed method.



If you get an error that indicates a table already exists and can't be created, it is probably because you ran the application after you deleted the database and before you executed update-database. In that case, delete the *Movies.mdf* file again and retry the update-database command. If you still get an error, delete the migrations folder and contents then start with the instructions at the top of this page (that is delete the *Movies.mdf* file then proceed to Enable-Migrations). If you still get an eror, open SQL Server Object Explorer and remove the database from the list.

Run the application and navigate to the */Movies* URL. The seed data is displayed.



**Adding a Rating Property to the Movie Model**

Start by adding a new Rating property to the existing Movie class. Open the *Models\Movie.cs* file and add the Rating property like this one:

C#Copy

public string Rating { get; set; }

The complete Movie class now looks like the following code:

C#Copy

public class Movie

{

public int ID { get; set; }

public string Title { get; set; }

[Display(Name = "Release Date")]

[DataType(DataType.Date)]

[DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]

public DateTime ReleaseDate { get; set; }

public string Genre { get; set; }

public decimal Price { get; set; }

public string Rating { get; set; }

}

Build the application (Ctrl+Shift+B).

Because you've added a new field to the Movie class, you also need to update the binding *white list* so this new property will be included. Update the bind attribute for Create and Edit action methods to include the Rating property:

C#Copy

[Bind(Include = "ID,Title,ReleaseDate,Genre,Price,Rating")]

You also need to update the view templates in order to display, create and edit the new Rating property in the browser view.

Open the *\Views\Movies\Index.cshtml* file and add a <th>Rating</th> column heading just after the **Price** column. Then add a <td> column near the end of the template to render the @item.Rating value. Below is what the updated *Index.cshtml* view template looks like:

CSHTMLCopy

@model IEnumerable<MvcMovie.Models.Movie>

@{

ViewBag.Title = "Index";

}

<h2>Index</h2>

<p>

@Html.ActionLink("Create New", "Create")

@using (Html.BeginForm("Index", "Movies", FormMethod.Get))

{

<p>

Genre: @Html.DropDownList("movieGenre", "All")

Title: @Html.TextBox("SearchString")

<input type="submit" value="Filter" />

</p>

}

</p>

<table class="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.Rating)

</th>

<th></th>

</tr>

@foreach (var item in Model) {

<tr>

<td>

@Html.DisplayFor(modelItem => item.Title)

</td>

<td>

@Html.DisplayFor(modelItem => item.ReleaseDate)

</td>

<td>

@Html.DisplayFor(modelItem => item.Genre)

</td>

<td>

@Html.DisplayFor(modelItem => item.Price)

</td>

<td>

@Html.DisplayFor(modelItem => item.Rating)

</td>

<td>

@Html.ActionLink("Edit", "Edit", new { id=item.ID }) |

@Html.ActionLink("Details", "Details", new { id=item.ID }) |

@Html.ActionLink("Delete", "Delete", new { id=item.ID })

</td>

</tr>

}

</table>

Next, open the *\Views\Movies\Create.cshtml* file and add the Rating field with the following highlighed markup. This renders a text box so that you can specify a rating when a new movie is created.

CSHTMLCopy

<div class="form-group">

@Html.LabelFor(model => model.Price, new { @class = "control-label col-md-2" })

<div class="col-md-10">

@Html.EditorFor(model => model.Price, new { htmlAttributes = new { @class = "form-control" } })

@Html.ValidationMessageFor(model => model.Price, "", new { @class = "text-danger" })

</div>

</div>

<div class="form-group">

@Html.LabelFor(model => model.Rating, new { @class = "control-label col-md-2" })

<div class="col-md-10">

@Html.EditorFor(model => model.Rating, new { htmlAttributes = new { @class = "form-control" } })

@Html.ValidationMessageFor(model => model.Rating, "", new { @class = "text-danger" })

</div>

</div>

<div class="form-group">

<div class="col-md-offset-2 col-md-10">

<input type="submit" value="Create" class="btn btn-default" />

</div>

</div>

</div>

}

<div>

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

</div>

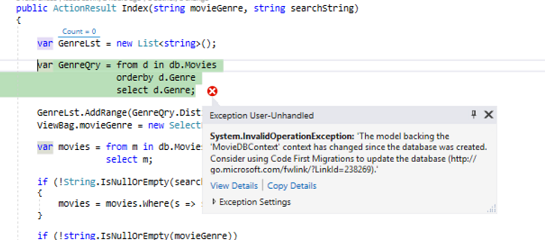
@section Scripts {

@Scripts.Render("~/bundles/jqueryval")

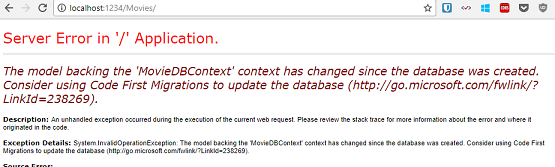
}

You've now updated the application code to support the new Rating property.

Run the application and navigate to the */Movies* URL. When you do this, though, you'll see one of the following errors:



The model backing the 'MovieDBContext' context has changed since the database was created. Consider using Code First Migrations to update the database (<https://go.microsoft.com/fwlink/?LinkId=238269>).



You're seeing this error because the updated Movie model class in the application is now different than the schema of the Movie table of the existing database. (There's no Rating column in the database table.)

There are a few approaches to resolving the error:

1. Have the Entity Framework automatically drop and re-create the database based on the new model class schema. This approach is very convenient early in the development cycle when you are doing active development on a test database; it allows you to quickly evolve the model and database schema together. The downside, though, is that you lose existing data in the database — so you *don't*want to use this approach on a production database! Using an initializer to automatically seed a database with test data is often a productive way to develop an application. For more information on Entity Framework database initializers, see [ASP.NET MVC/Entity Framework tutorial](https://docs.microsoft.com/en-us/aspnet/mvc/overview/getting-started/getting-started-with-ef-using-mvc/creating-an-entity-framework-data-model-for-an-asp-net-mvc-application).
2. Explicitly modify the schema of the existing database so that it matches the model classes. The advantage of this approach is that you keep your data. You can make this change either manually or by creating a database change script.
3. Use Code First Migrations to update the database schema.

For this tutorial, we'll use Code First Migrations.

Update the Seed method so that it provides a value for the new column. Open Migrations\Configuration.cs file and add a Rating field to each Movie object.

C#Copy

new Movie

{

Title = "When Harry Met Sally",

ReleaseDate = DateTime.Parse("1989-1-11"),

Genre = "Romantic Comedy",

Rating = "PG",

Price = 7.99M

},

Build the solution, and then open the **Package Manager Console** window and enter the following command:

add-migration Rating

The add-migration command tells the migration framework to examine the current movie model with the current movie DB schema and create the necessary code to migrate the DB to the new model. The name *Rating* is arbitrary and is used to name the migration file. It's helpful to use a meaningful name for the migration step.

When this command finishes, Visual Studio opens the class file that defines the new DbMigration derived class, and in the Up method you can see the code that creates the new column.

C#Copy

public partial class AddRatingMig : DbMigration

{

public override void Up()

{

AddColumn("dbo.Movies", "Rating", c => c.String());

}

public override void Down()

{

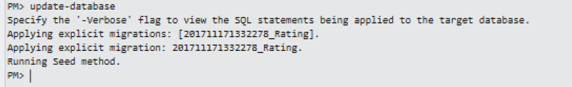
DropColumn("dbo.Movies", "Rating");

}

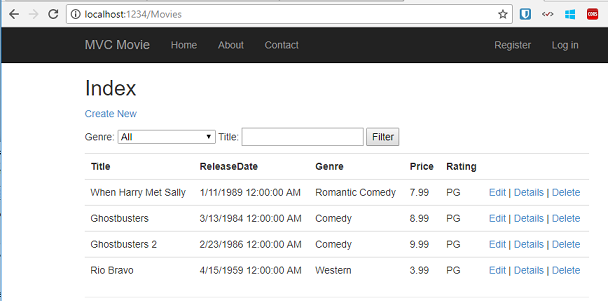
}

Build the solution, and then enter the update-database command in the **Package Manager Console** window.

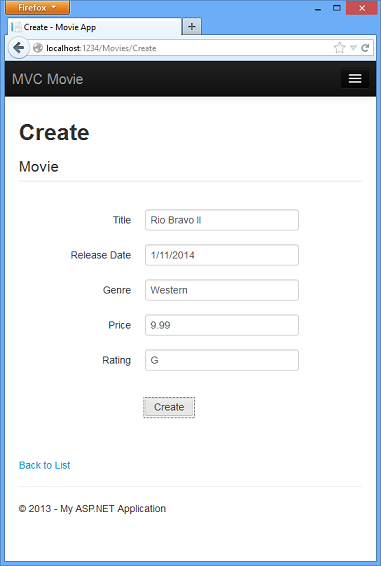
The following image shows the output in the **Package Manager Console** window (The date stamp prepending *Rating* will be different.)



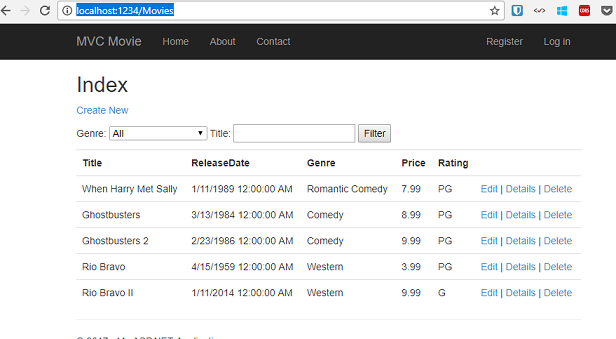
Re-run the application and navigate to the /Movies URL. You can see the new Rating field.



Click the **Create New** link to add a new movie. Note that you can add a rating.



Click **Create**. The new movie, including the rating, now shows up in the movies listing:



Now that the project is using migrations, you won't need to drop the database when you add a new field or otherwise update the schema. In the next section, we'll make more schema changes and use migrations to update the database.

You should also add the Rating field to the Edit, Details, and Delete view templates.

You could enter the "update-database" command in the **Package Manager Console**window again and no migration code would run, because the schema matches the model. However, running "update-database" will run the Seed method again, and if you changed any of the Seed data, the changes will be lost because the Seed method upserts data. You can read more about the Seed method in Tom Dykstra's popular [ASP.NET MVC/Entity Framework tutorial](https://docs.microsoft.com/en-us/aspnet/mvc/overview/getting-started/getting-started-with-ef-using-mvc/creating-an-entity-framework-data-model-for-an-asp-net-mvc-application).

In this section you saw how you can modify model objects and keep the database in sync with the changes. You also learned a way to populate a newly created database with sample data so you can try out scenarios. This was just a quick introduction to Code First, see [Creating an Entity Framework Data Model for an ASP.NET MVC Application](https://docs.microsoft.com/en-us/aspnet/mvc/overview/getting-started/getting-started-with-ef-using-mvc/creating-an-entity-framework-data-model-for-an-asp-net-mvc-application) for a more complete tutorial on the subject. Next, let's look at how you can add richer validation logic to the model classes and enable some business rules to be enforced.

**Adding Validation**

* 10/17/2013
* 11 minutes to read
* Contributors
  + [Rick Anderson](https://github.com/Rick-Anderson)

* + [Scott Addie](https://github.com/scottaddie)

* + [Zack ](https://github.com/zunderscore)

* + [Dustan Vodvarka](https://github.com/DustanVodvarka)

* + [sfoo](https://github.com/GuessWhoSamFoo)

* + [all](https://github.com/aspnet/Docs/blob/master/aspnet/mvc/overview/getting-started/introduction/adding-validation.md)

by [Rick Anderson](https://github.com/Rick-Anderson)

**Note**

This document is part of the [**Getting Started with ASP.NET MVC 5**](https://docs.microsoft.com/en-us/aspnet/mvc/overview/getting-started/introduction/getting-started) tutorial. Final Source for tutorial located on [**GitHub**](https://github.com/aspnet/Docs/tree/master/aspnet/mvc/overview/getting-started/introduction/sample/MvcMovie/MvcMovie)

In this section you'll add validation logic to the Movie model, and you'll ensure that the validation rules are enforced any time a user attempts to create or edit a movie using the application.

**Keeping Things DRY**

One of the core design tenets of ASP.NET MVC is [DRY](http://en.wikipedia.org/wiki/Don%27t_repeat_yourself) ("Don't Repeat Yourself"). ASP.NET MVC encourages you to specify functionality or behavior only once, and then have it be reflected everywhere in an application. This reduces the amount of code you need to write and makes the code you do write less error prone and easier to maintain.

The validation support provided by ASP.NET MVC and Entity Framework Code First is a great example of the DRY principle in action. You can declaratively specify validation rules in one place (in the model class) and the rules are enforced everywhere in the application.

Let's look at how you can take advantage of this validation support in the movie application.

**Adding Validation Rules to the Movie Model**

You'll begin by adding some validation logic to the Movie class.

Open the *Movie.cs* file. Notice the [System.ComponentModel.DataAnnotations](https://msdn.microsoft.com/library/system.componentmodel.dataannotations.aspx) namespace does not contain System.Web. DataAnnotations provides a built-in set of validation attributes that you can apply declaratively to any class or property. (It also contains formatting attributes like [DataType](https://msdn.microsoft.com/library/system.componentmodel.dataannotations.datatype.aspx) that help with formatting and don't provide any validation.)

Now update the Movie class to take advantage of the built-in [Required](https://msdn.microsoft.com/library/system.componentmodel.dataannotations.requiredattribute.aspx), [StringLength](https://msdn.microsoft.com/library/system.componentmodel.dataannotations.stringlengthattribute.aspx), [RegularExpression](https://msdn.microsoft.com/library/system.componentmodel.dataannotations.regularexpressionattribute.aspx), and [Range](https://msdn.microsoft.com/library/system.componentmodel.dataannotations.rangeattribute.aspx) validation attributes. Replace the Movie class with the following:

C#Copy

public class Movie

{

    public int ID { get; set; }

    [StringLength(60, MinimumLength = 3)]

    public string Title { get; set; }

    [Display(Name = "Release Date")]

    [DataType(DataType.Date)]

    [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]

    public DateTime ReleaseDate { get; set; }

    [RegularExpression(@"^[A-Z]+[a-zA-Z'\s]\*$")]

    [Required]

    [StringLength(30)]

    public string Genre { get; set; }

    [Range(1, 100)]

    [DataType(DataType.Currency)]

    public decimal Price { get; set; }

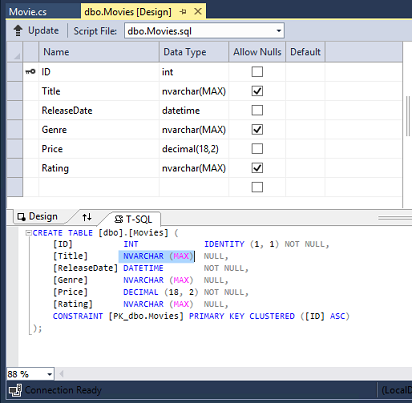
    [RegularExpression(@"^[A-Z]+[a-zA-Z'\s]\*$")]

    [StringLength(5)]

    public string Rating { get; set; }

}

The [StringLength](https://msdn.microsoft.com/library/system.componentmodel.dataannotations.stringlengthattribute.aspx) attribute sets the maximum length of the string, and it sets this limitation on the database, therefore the database schema will change. Right click on the **Movies** table in **Server explorer** and click **Open Table Definition**:



In the image above, you can see all the string fields are set to [NVARCHAR (MAX)](https://technet.microsoft.com/library/ms186939.aspx). We will use migrations to update the schema. Build the solution, and then open the **Package Manager Console** window and enter the following commands:

consoleCopy

add-migration DataAnnotations

update-database

When this command finishes, Visual Studio opens the class file that defines the new DbMIgration derived class with the name specified (DataAnnotations), and in the Upmethod you can see the code that updates the schema constraints:

C#Copy

public override void Up()

{

    AlterColumn("dbo.Movies", "Title", c => c.String(maxLength: 60));

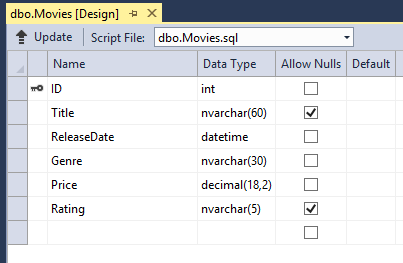
    AlterColumn("dbo.Movies", "Genre", c => c.String(nullable: false, maxLength: 30));

    AlterColumn("dbo.Movies", "Rating", c => c.String(maxLength: 5));

}

The Genre field is no longer nullable (that is, you must enter a value). The Rating field has a maximum length of 5 and Title has a maximum length of 60. The minimum length of 3 on Title and the range on Price did not create schema changes.

Examine the Movie schema:



The string fields show the new length limits and Genre is no longer checked as nullable.

The validation attributes specify behavior that you want to enforce on the model properties they are applied to. The Required and MinimumLength attributes indicates that a property must have a value; but nothing prevents a user from entering white space to satisfy this validation. The [RegularExpression](https://msdn.microsoft.com/library/system.componentmodel.dataannotations.regularexpressionattribute.aspx) attribute is used to limit what characters can be input. In the code above, Genre and Rating must use only letters (white space, numbers and special characters are not allowed). The [Range](https://msdn.microsoft.com/library/system.componentmodel.dataannotations.rangeattribute.aspx) attribute constrains a value to within a specified range. The StringLength attribute lets you set the maximum length of a string property, and optionally its minimum length. Value types (such as decimal, int, float, DateTime) are inherently required and don't need the Requiredattribute.

Code First ensures that the validation rules you specify on a model class are enforced before the application saves changes in the database. For example, the code below will throw a [DbEntityValidationException](https://msdn.microsoft.com/library/system.data.entity.validation.dbentityvalidationexception(v=vs.103).aspx) exception when the SaveChanges method is called, because several required Movie property values are missing:

C#Copy

MovieDBContext db = new MovieDBContext();

Movie movie = new Movie();

movie.Title = "Gone with the Wind";

db.Movies.Add(movie);

db.SaveChanges();        // <= Will throw server side validation exception

The code above throws the following exception:

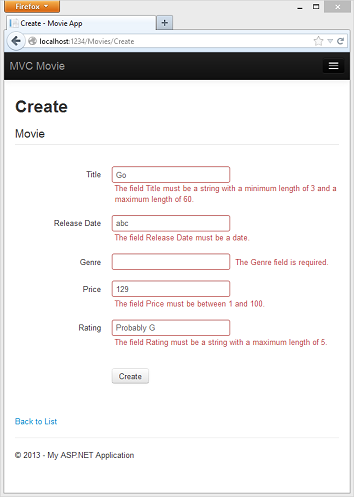
*Validation failed for one or more entities. See 'EntityValidationErrors' property for more details.*

Having validation rules automatically enforced by the .NET Framework helps make your application more robust. It also ensures that you can't forget to validate something and inadvertently let bad data into the database.

**Validation Error UI in ASP.NET MVC**

Run the application and navigate to the */Movies* URL.

Click the **Create New** link to add a new movie. Fill out the form with some invalid values. As soon as jQuery client side validation detects the error, it displays an error message.



**Note**

to support jQuery validation for non-English locales that use a comma (",") for a decimal point, you must include the NuGet globalize as described previously in this tutorial.

Notice how the form has automatically used a red border color to highlight the text boxes that contain invalid data and has emitted an appropriate validation error message next to each one. The errors are enforced both client-side (using JavaScript and jQuery) and server-side (in case a user has JavaScript disabled).

A real benefit is that you didn't need to change a single line of code in the MoviesController class or in the *Create.cshtml* view in order to enable this validation UI. The controller and views you created earlier in this tutorial automatically picked up the validation rules that you specified by using validation attributes on the properties of the Movie model class. Test validation using the Edit action method, and the same validation is applied.

The form data is not sent to the server until there are no client side validation errors. You can verify this by putting a break point in the HTTP Post method, by using the [fiddler tool](http://fiddler2.com/fiddler2/), or the IE [F12 developer tools](https://msdn.microsoft.com/ie/aa740478).

**How Validation Occurs in the Create View and Create Action Method**

You might wonder how the validation UI was generated without any updates to the code in the controller or views. The next listing shows what the Create methods in the MovieController class look like. They're unchanged from how you created them earlier in this tutorial.

C#Copy

public ActionResult Create()

{

    return View();

}

// POST: /Movies/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 = "ID,Title,ReleaseDate,Genre,Price,Rating")] Movie movie)

{

    if (ModelState.IsValid)

    {

        db.Movies.Add(movie);

        db.SaveChanges();

        return RedirectToAction("Index");

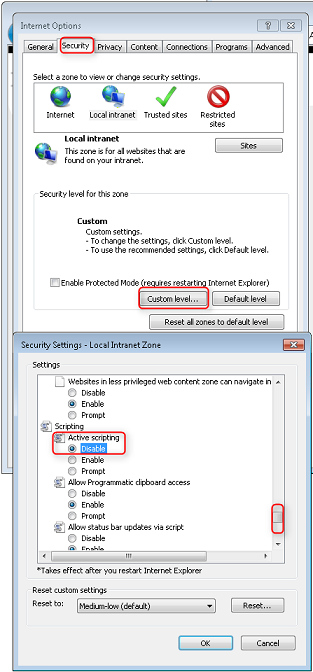
    }

    return View(movie);

}

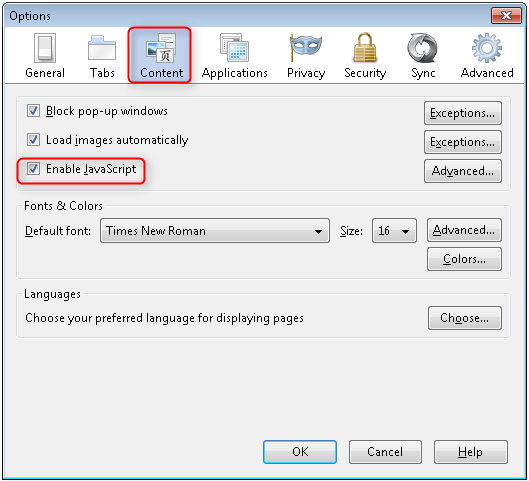
The first (HTTP GET) Create action method displays the initial Create form. The second ([HttpPost]) version handles the form post. The second Create method (The HttpPostversion) calls ModelState.IsValid to check whether the movie has any validation errors. Calling this method evaluates any validation attributes that have been applied to the object. If the object has validation errors, the Create method re-displays the form. If there are no errors, the method saves the new movie in the database. In our movie example, **the form is not posted to the server when there are validation errors detected on the client side; the second** Create**method is never called**. If you disable JavaScript in your browser, client validation is disabled and the HTTP POST Create method calls ModelState.IsValid to check whether the movie has any validation errors.

You can set a break point in the HttpPost Create method and verify the method is never called, client side validation will not submit the form data when validation errors are detected. If you disable JavaScript in your browser, then submit the form with errors, the break point will be hit. You still get full validation without JavaScript. The following image shows how to disable JavaScript in Internet Explorer.





The following image shows how to disable JavaScript in the FireFox browser.



The following image shows how to disable JavaScript in the Chrome browser.



Below is the *Create.cshtml* view template that you scaffolded earlier in the tutorial. It's used by the action methods shown above both to display the initial form and to redisplay it in the event of an error.

CSHTMLCopy

@model MvcMovie.Models.Movie

@{

    ViewBag.Title = "Create";

}

<h2>Create</h2>

@using (Html.BeginForm())

{

    @Html.AntiForgeryToken()

    <div class="form-horizontal">

        <h4>Movie</h4>

        <hr />

        @Html.ValidationSummary(true)

        <div class="form-group">

            @Html.LabelFor(model => model.Title, new { @class = "control-label col-md-2" })

            <div class="col-md-10">

                @Html.EditorFor(model => model.Title)

                @Html.ValidationMessageFor(model => model.Title)

            </div>

        </div>

        @\*Fields removed for brevity.\*@

        <div class="form-group">

            <div class="col-md-offset-2 col-md-10">

                <input type="submit" value="Create" class="btn btn-default" />

            </div>

        </div>

    </div>

}

<div>

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

</div>

@section Scripts {

    @Scripts.Render("~/bundles/jqueryval")

}

Notice how the code uses an Html.EditorFor helper to output the <input> element for each Movie property. Next to this helper is a call to the Html.ValidationMessageForhelper method. These two helper methods work with the model object that's passed by the controller to the view (in this case, a Movie object). They automatically look for validation attributes specified on the model and display error messages as appropriate.

What's really nice about this approach is that neither the controller nor the Create view template knows anything about the actual validation rules being enforced or about the specific error messages displayed. The validation rules and the error strings are specified only in the Movie class. These same validation rules are automatically applied to the Edit view and any other views templates you might create that edit your model.

If you want to change the validation logic later, you can do so in exactly one place by adding validation attributes to the model (in this example, the movie class). You won't have to worry about different parts of the application being inconsistent with how the rules are enforced — all validation logic will be defined in one place and used everywhere. This keeps the code very clean, and makes it easy to maintain and evolve. And it means that you'll be fully honoring the *DRY* principle.

**Using DataType Attributes**

Open the *Movie.cs* file and examine the Movie class. The [System.ComponentModel.DataAnnotations](https://msdn.microsoft.com/library/system.componentmodel.dataannotations.aspx) namespace provides formatting attributes in addition to the built-in set of validation attributes. We've already applied a [DataType](https://msdn.microsoft.com/library/system.componentmodel.dataannotations.datatype.aspx)enumeration value to the release date and to the price fields. The following code shows the ReleaseDate and Price properties with the appropriate [DataType](https://msdn.microsoft.com/library/system.componentmodel.dataannotations.datatype.aspx) attribute.

C#Copy

[DataType(DataType.Date)]

public DateTime ReleaseDate { get; set; }

[DataType(DataType.Currency)]

public decimal Price { get; set; }

The [DataType](https://msdn.microsoft.com/library/system.componentmodel.dataannotations.datatypeattribute.aspx) attributes only provide hints for the view engine to format the data (and supply attributes such as <a> for URL's and <a href="mailto:EmailAddress.com"> for email. You can use the [RegularExpression](https://msdn.microsoft.com/library/system.componentmodel.dataannotations.regularexpressionattribute.aspx) attribute to validate the format of the data. The [DataType](https://msdn.microsoft.com/library/system.componentmodel.dataannotations.datatypeattribute.aspx) attribute is used to specify a data type that is more specific than the database intrinsic type, they are ***not*** validation attributes. In this case we only want to keep track of the date, not the date and time. The [DataType Enumeration](https://msdn.microsoft.com/library/system.componentmodel.dataannotations.datatype.aspx) provides for many data types, such as *Date, Time, PhoneNumber, Currency, EmailAddress* and more. The DataTypeattribute can also enable the application to automatically provide type-specific features. For example, a mailto: link can be created for [DataType.EmailAddress](https://msdn.microsoft.com/library/system.componentmodel.dataannotations.datatype.aspx), and a date selector can be provided for [DataType.Date](https://msdn.microsoft.com/library/system.componentmodel.dataannotations.datatype.aspx) in browsers that support [HTML5](http://html5.org/). The [DataType](https://msdn.microsoft.com/library/system.componentmodel.dataannotations.datatypeattribute.aspx) attributes emits HTML 5 [data-](http://ejohn.org/blog/html-5-data-attributes/) (pronounced *data dash*) attributes that HTML 5 browsers can understand. The [DataType](https://msdn.microsoft.com/library/system.componentmodel.dataannotations.datatypeattribute.aspx) attributes do not provide any validation.

DataType.Date does not specify the format of the date that is displayed. By default, the data field is displayed according to the default formats based on the server's [CultureInfo](https://msdn.microsoft.com/library/vstudio/system.globalization.cultureinfo(v=vs.110).aspx).

The DisplayFormat attribute is used to explicitly specify the date format:

C#Copy

[DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]

public DateTime EnrollmentDate { get; set; }

The ApplyFormatInEditMode setting specifies that the specified formatting should also be applied when the value is displayed in a text box for editing. (You might not want that for some fields — for example, for currency values, you might not want the currency symbol in the text box for editing.)

You can use the [DisplayFormat](https://msdn.microsoft.com/library/system.componentmodel.dataannotations.displayformatattribute.aspx) attribute by itself, but it's generally a good idea to use the [DataType](https://msdn.microsoft.com/library/system.componentmodel.dataannotations.datatypeattribute.aspx) attribute also. The DataType attribute conveys the *semantics* of the data as opposed to how to render it on a screen, and provides the following benefits that you don't get with DisplayFormat:

* The browser can enable HTML5 features (for example to show a calendar control, the locale-appropriate currency symbol, email links, etc.).
* By default, the browser will render data using the correct format based on your [locale](https://msdn.microsoft.com/library/vstudio/wyzd2bce.aspx).
* The [DataType](https://msdn.microsoft.com/library/system.componentmodel.dataannotations.datatypeattribute.aspx) attribute can enable MVC to choose the right field template to render the data (the [DisplayFormat](https://msdn.microsoft.com/library/system.componentmodel.dataannotations.displayformatattribute.aspx) if used by itself uses the string template). For more information, see Brad Wilson's [ASP.NET MVC 2 Templates](http://bradwilson.typepad.com/blog/2009/10/aspnet-mvc-2-templates-part-1-introduction.html). (Though written for MVC 2, this article still applies to the current version of ASP.NET MVC.)

If you use the DataType attribute with a date field, you have to specify the DisplayFormat attribute also in order to ensure that the field renders correctly in Chrome browsers. For more information, see [this StackOverflow thread](http://stackoverflow.com/questions/12633471/mvc4-datatype-date-editorfor-wont-display-date-value-in-chrome-fine-in-ie).

**Note**

jQuery validation does not work with the [**Range**](https://msdn.microsoft.com/library/system.componentmodel.dataannotations.rangeattribute.aspx) attribute and [**DateTime**](https://msdn.microsoft.com/library/system.datetime.aspx). For example, the following code will always display a client side validation error, even when the date is in the specified range:

C#Copy

[Range(typeof(DateTime), "1/1/1966", "1/1/2020")]

You will need to disable jQuery date validation to use the [**Range**](https://msdn.microsoft.com/library/system.componentmodel.dataannotations.rangeattribute.aspx) attribute with [**DateTime**](https://msdn.microsoft.com/library/system.datetime.aspx). It's generally not a good practice to compile hard dates in your models, so using the [**Range**](https://msdn.microsoft.com/library/system.componentmodel.dataannotations.rangeattribute.aspx) attribute and [**DateTime**](https://msdn.microsoft.com/library/system.datetime.aspx) is discouraged.

The following code shows combining attributes on one line:

C#Copy

public class Movie

{

   public int ID { get; set; }

   [Required,StringLength(60, MinimumLength = 3)]

   public string Title { get; set; }

   [Display(Name = "Release Date"),DataType(DataType.Date)]

   public DateTime ReleaseDate { get; set; }

   [Required]

   public string Genre { get; set; }

   [Range(1, 100),DataType(DataType.Currency)]

   public decimal Price { get; set; }

   [Required,StringLength(5)]

   public string Rating { get; set; }

}

In the next part of the series, we'll review the application and make some improvements to the automatically generated Details and Delete methods.

**Examining the Details and Delete Methods**

* 03/26/2015
* 4 minutes to read
* Contributors
  + [Rick Anderson](https://github.com/Rick-Anderson)

* + [Scott Addie](https://github.com/scottaddie)

* + [Dustan Vodvarka](https://github.com/DustanVodvarka)

* + [Luke Latham](https://github.com/guardrex)

* + [Isaac Levin](https://github.com/isaac2004)

* + [all](https://github.com/aspnet/Docs/blob/master/aspnet/mvc/overview/getting-started/introduction/examining-the-details-and-delete-methods.md)

by [Rick Anderson](https://github.com/Rick-Anderson)

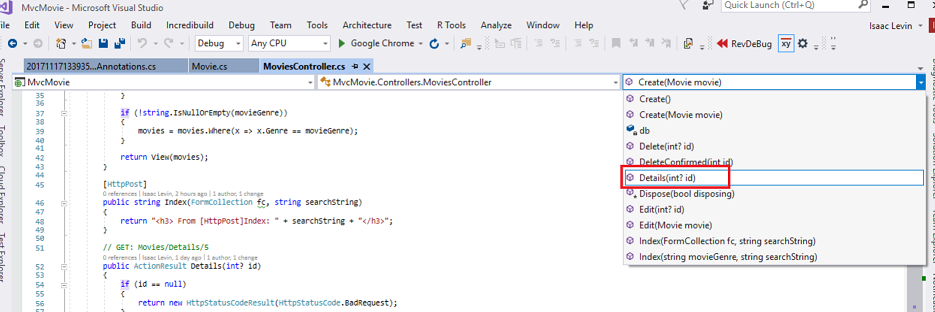
**Note**

This document is part of the [**Getting Started with ASP.NET MVC 5**](https://docs.microsoft.com/en-us/aspnet/mvc/overview/getting-started/introduction/getting-started) tutorial. Final Source for tutorial located on [**GitHub**](https://github.com/aspnet/Docs/tree/master/aspnet/mvc/overview/getting-started/introduction/sample/MvcMovie/MvcMovie)

In this part of the tutorial, you'll examine the automatically generated Details and Delete methods.

**Examining the Details and Delete Methods**

Open the Movie controller and examine the Details method.



C#Copy

public ActionResult Details(int? id)

{

if (id == null)

{

return new HttpStatusCodeResult(HttpStatusCode.BadRequest);

}

Movie movie = db.Movies.Find(id);

if (movie == null)

{

return HttpNotFound();

}

return View(movie);

}

The MVC scaffolding engine that created this action method adds a comment showing a HTTP request that invokes the method. In this case it's a GET request with three URL segments, the Movies controller, the Details method and a ID value.

Code First makes it easy to search for data using the Find method. An important security feature built into the method is that the code verifies that the Find method has found a movie before the code tries to do anything with it. For example, a hacker could introduce errors into the site by changing the URL created by the links from http://localhost:xxxx/Movies/Details/1 to something like http://localhost:xxxx/Movies/Details/12345 (or some other value that doesn't represent an actual movie). If you did not check for a null movie, a null movie would result in a database error.

Examine the Delete and DeleteConfirmed methods.

C#Copy

// GET: /Movies/Delete/5

public ActionResult Delete(int? id)

{

if (id == null)

{

return new HttpStatusCodeResult(HttpStatusCode.BadRequest);

}

Movie movie = db.Movies.Find(id);

if (movie == null)

{

return HttpNotFound();

}

return View(movie);

}

// POST: /Movies/Delete/5

[HttpPost, ActionName("Delete")]

[ValidateAntiForgeryToken]

public ActionResult DeleteConfirmed(int id)

{

Movie movie = db.Movies.Find(id);

db.Movies.Remove(movie);

db.SaveChanges();

return RedirectToAction("Index");

}

Note that the HTTP GET Delete method doesn't delete the specified movie, it returns a view of the movie where you can submit (HttpPost) the deletion. Performing a delete operation in response to a GET request (or for that matter, performing an edit operation, create operation, or any other operation that changes data) opens up a security hole. For more information about this, see Stephen Walther's blog entry [ASP.NET MVC Tip #46 — Don't use Delete Links because they create Security Holes](http://stephenwalther.com/blog/archive/2009/01/21/asp.net-mvc-tip-46-ndash-donrsquot-use-delete-links-because.aspx).

The HttpPost method that deletes the data is named DeleteConfirmed to give the HTTP POST method a unique signature or name. The two method signatures are shown below:

C#Copy

// GET: /Movies/Delete/5

public ActionResult Delete(int? id)

//

// POST: /Movies/Delete/5

[HttpPost, ActionName("Delete")]

public ActionResult DeleteConfirmed(int id)

The common language runtime (CLR) requires overloaded methods to have a unique parameter signature (same method name but different list of parameters). However, here you need two Delete methods -- one for GET and one for POST -- that both have the same parameter signature. (They both need to accept a single integer as a parameter.)

To sort this out, you can do a couple of things. One is to give the methods different names. That's what the scaffolding mechanism did in the preceding example. However, this introduces a small problem: ASP.NET maps segments of a URL to action methods by name, and if you rename a method, routing normally wouldn't be able to find that method. The solution is what you see in the example, which is to add the ActionName("Delete") attribute to the DeleteConfirmed method. This effectively performs mapping for the routing system so that a URL that includes */Delete/* for a POST request will find the DeleteConfirmed method.

Another common way to avoid a problem with methods that have identical names and signatures is to artificially change the signature of the POST method to include an unused parameter. For example, some developers add a parameter type FormCollection that is passed to the POST method, and then simply don't use the parameter:

C#Copy

public ActionResult Delete(FormCollection fcNotUsed, int id = 0)

{

Movie movie = db.Movies.Find(id);

if (movie == null)

{

return HttpNotFound();

}

db.Movies.Remove(movie);

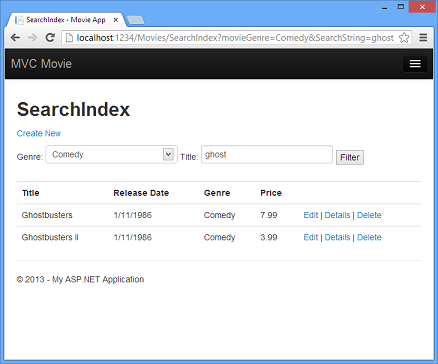
db.SaveChanges();

return RedirectToAction("Index");

}

**Summary**

You now have a complete ASP.NET MVC application that stores data in a local DB database. You can create, read, update, delete, and search for movies.



**Next Steps**

After you have built and tested a web application, the next step is to make it available to other people to use over the Internet. To do that, you have to deploy it to a web hosting provider. Microsoft offers free web hosting for up to 10 web sites in a [free Azure trial account](https://www.windowsazure.com/pricing/free-trial/?WT.mc_id=A443DD604). I suggest you next follow my tutorial [Deploy a Secure ASP.NET MVC app with Membership, OAuth, and SQL Database to Azure](https://docs.microsoft.com/aspnet/core/security/authorization/secure-data). An excellent tutorial is Tom Dykstra's intermediate-level [Creating an Entity Framework Data Model for an ASP.NET MVC Application](https://docs.microsoft.com/en-us/aspnet/mvc/overview/getting-started/getting-started-with-ef-using-mvc/creating-an-entity-framework-data-model-for-an-asp-net-mvc-application). [Stackoverflow](http://stackoverflow.com/help) and the [ASP.NET MVC forums](https://forums.asp.net/1146.aspx) are a great places to ask questions. Follow [me](https://twitter.com/RickAndMSFT) on twitter so you can get updates on my latest tutorials.

Feedback is welcome.