

Using Entity Framework 6 in Visual Studio 2013

Overview

ADO.NET Entity Framework (EF) is an object-relational mapper (ORM) that enables you to add data access logic to your applications by programming with a conceptual application model instead of programming directly using a relational storage schema.

Entity Framework simplifies data access by avoiding working directly with the database in your code. Instead, you can retrieve data by writing queries against strongly typed classes letting Entity Framework handle the database interaction on your behalf, including persisting changes back to the database for you.

There are three different ways to work with Entity Framework:

- **Database First.** Starting with an existing database, generate a conceptual model by using the Entity Data Model Tools. This generates a default conceptual model and mapping, which you can customize by using the Entity Framework Designer.
- **Model First.** Graphically create a conceptual model by using the Entity Framework Designer, and then generate a database using the metadata built with tools from the same model.
- **Code First.** Define your object model in code. For this, Entity Framework supports two scenarios. First, It can infer a conceptual model based on the object types and additional configurations that you define. The mapping metadata is generated during run time based on a combination of how you define your domain types and additional configuration information that you provide in code. Then Entity Framework generates the database as needed based on the metadata. Alternatively, you can just use an existing database.

Besides Code First, Entity Framework also has a DbContext API, which provides a simplified abstraction over the existingObjectContext type, optimized for common tasks and coding patterns.

In this Hands-on Lab, you will create a simple MVC 5 application and learn how to use the three Entity Framework approaches to define your domain-specific entity types, database schema and mapping layer between the two. You will also learn how to expose your Entity Framework data model using the OData protocol.

Objectives

In this hands-on lab, you will learn how to use all three modeling workflows:

- Create a database from a code-based model using Code First
- Create an Entity Data Model (EDMX) from an existing database using Database First
- Create a database from an Entity Data Model (EDMX) using the Entity Framework Designer (Model First).
- Generate model classes from an EDMX using the ADO.NET DbContext Generator template

Then with an existing model and classes you will learn how to:

- Build a simple ASP.NET MVC 5 data-centric application using scaffolding features and Entity Framework
- Create a WCF Data Service to expose the Entity Framework Data Model with the OData protocol
- Use Code First Fluent API to override Entity Framework default conventions and configurations

Prerequisites

The following is required to complete this hands-on lab:

- [Microsoft Visual Studio 2013](#) (with Update 2 RC applied)
- [SQL Server Express](#)

Notes

Estimated time to complete this lab: **60** minutes.

Note: You can log into the virtual machine with user name “**User**” and password “**P2ssw0rd**”.

Note: This lab may make references to code and other assets that are needed to complete the exercises. You can find these assets on the desktop in a folder named **TechEd 2014**. Within that folder, you will find additional folders that match the name of the lab you are working on.

Exercise 1: Using Code First

The Entity Framework Code First modeling workflow allows you to use your own domain classes to represent the model that EF relies on when performing querying, change tracking, and updating functions. Using the Code First development workflow, you do not need to begin your application by creating a database or specifying schema. Instead, you can begin by writing standard .NET classes that define the most appropriate domain model objects for your application.

Code First leverages a programming pattern referred to as convention over configuration. What this means is that Code First will assume that your classes follow the conventions of the schema that Entity Framework uses in its conceptual model. Therefore, Entity Framework will be able to work out the details it needs to do its job. However, if your classes do not follow those conventions, you have the ability to add configurations to your classes to provide Entity Framework with the necessary information. Code First can recognize common relationships between classes based on properties that point to child collections or to a single class. When your classes do not use foreign keys, Code First can infer database foreign keys for you.

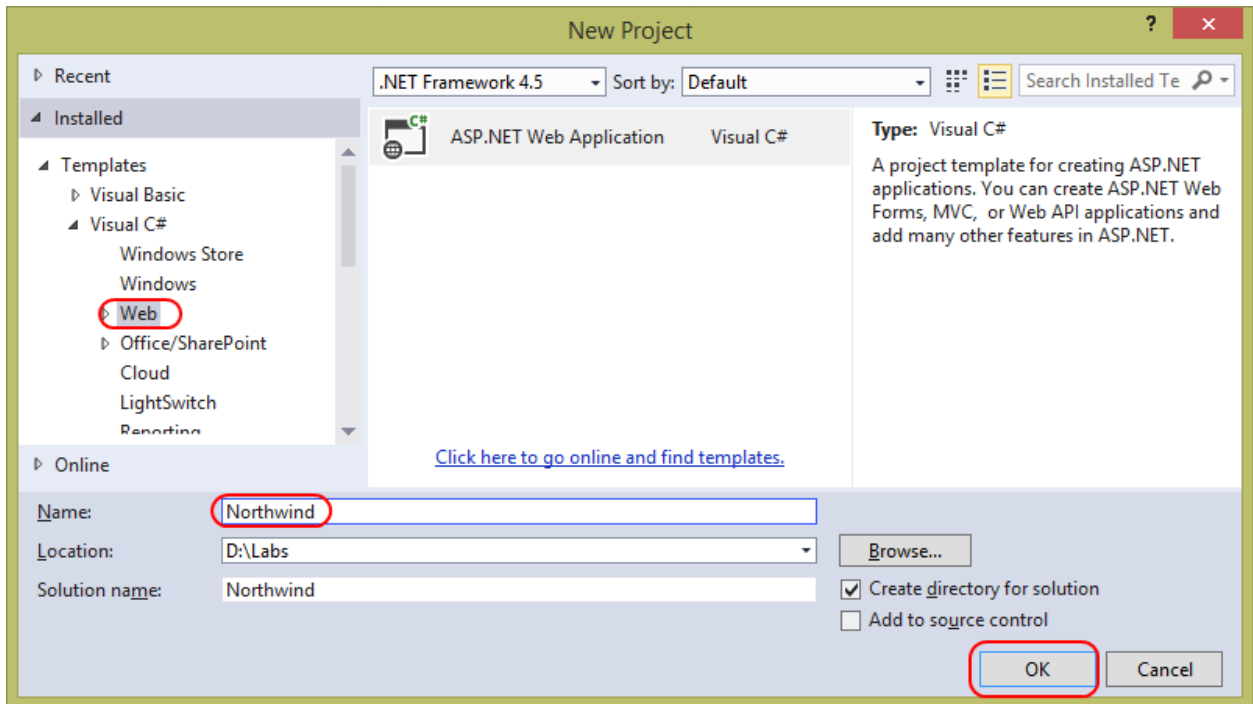
In this exercise, you will be introduced to Entity Framework Code First. The exercise will demonstrate how to use the default mapping conventions to create a new database and use it within an ASP.NET MVC 5 web application. You will also learn how to configure your model classes with Data Annotations to override Entity Framework conventions.

Task 1 – Creating the MVC 5 Application

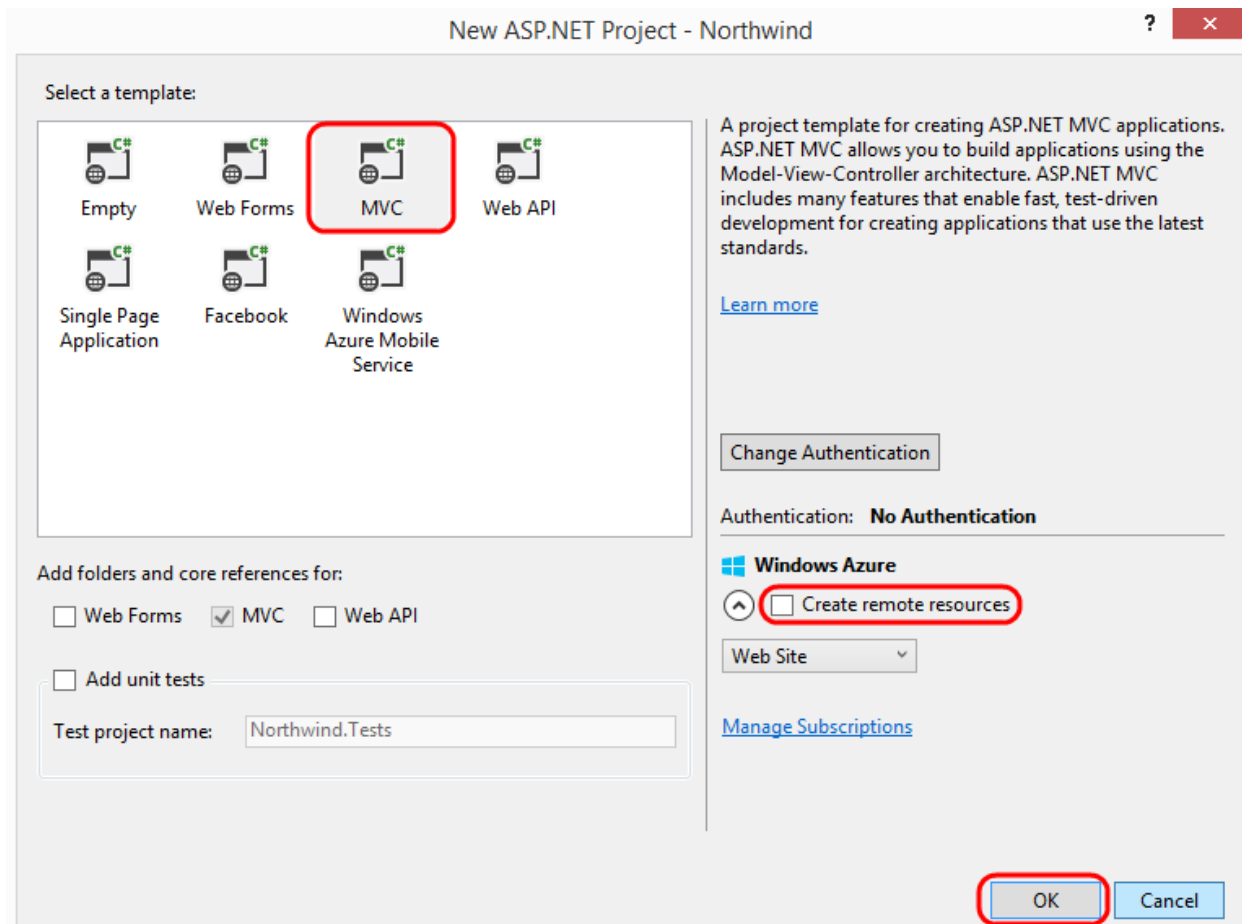
In this task, you will create an MVC 5 Web application and install Entity Framework.

1. Open **Visual Studio 2013**.
2. Select **File | New | Project**.

3. In the **New Project** dialog, select **Visual C# | Web** from the left pane. Name the project **“Northwind”**, select a location (or leave the default value), and click **OK**.



4. In the **New ASP.NET Project** dialog, select the **MVC** template, **de-select** the option to **“Create remote resources”** and click **OK**. This will create our Web application.



5. Now you will add the Entity Framework to the project. From the main menu, select **Tools | NuGet Package Manager | Package Manager Console**.
6. Enter the following to install Entity Framework into the Northwind project.

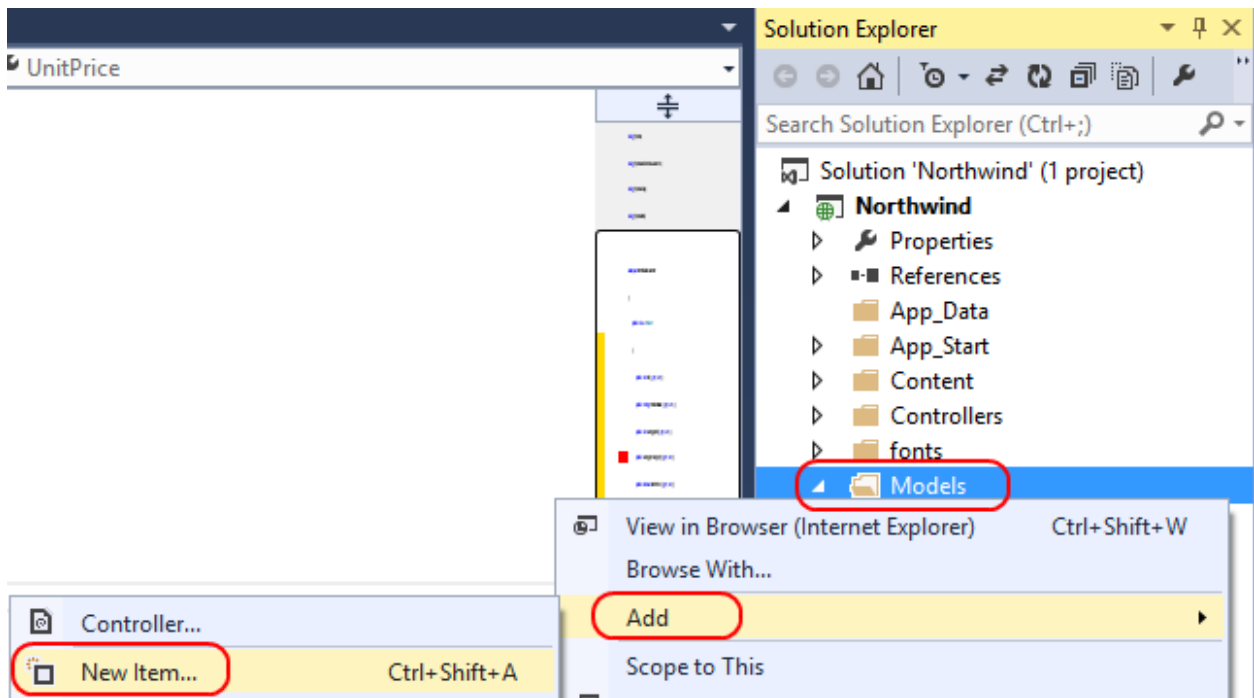
Text

```
Install-Package EntityFramework -Version 6.0.0 -project Northwind
```

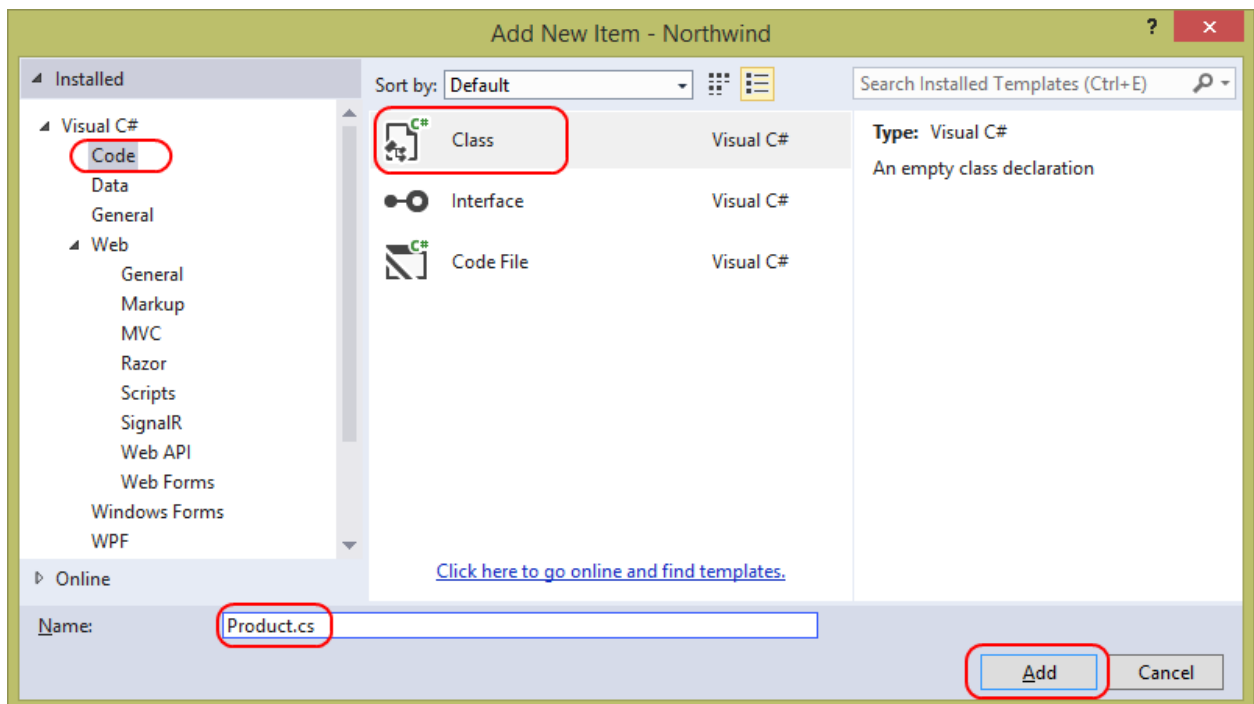
Task 2 – Creating the Model Classes

In this task, you will create your domain model using POCO (Plain Old CLR Object) classes, which will have no dependency on Entity Framework. You will also create a context class using the DbContext and DbSet classes from Entity Framework. These classes will enable you to map your POCO model classes to and from database tables.

1. Right-click the **Models** folder and select **Add | New Item....**



2. In the **Add New Item** dialog, select the **Code** category from the left pane, select the **Class** template, and enter the **Name "Product.cs"**. Click **Add** to add the class.



When using a code-first development workflow, you do not need to begin your application by creating a database or specifying schema. Instead, you can begin with POCO which defines the domain model objects that are most appropriate for your application without having to worry about the data persistence logic within them. In this case you will define two POCO classes: "Product" and

“Category”. These classes will be used to represent the “Products” and “Categories” tables within a Northwind database.

3. Add the following properties to the **Product** class:

```
C#
public int Id { get; set; }
public string ProductName { get; set; }
public int CategoryId { get; set; }
public Category Category { get; set; }
public decimal UnitPrice { get; set; }
```

Note: If your class defines a property whose name is “ID” or “Id”, or a class name followed by “ID” or “Id”, Entity Framework treats this property as a key by convention. To learn more about Entity Framework conventions see Exercise 5.

4. Create another class in the **Models** folder using the steps above and name it “**Category.cs**”.
5. Add the following properties to the **Category** class.

```
C#
public int Id { get; set; }
public string CategoryName { get; set; }
public ICollection<Product> Products { get; set; }
```

Code First makes it easy to take advantage of primary key and foreign key relationships within the database and expose properties on model classes that enable you to traverse across them. In the previous code, you have exposed a Products property on the Category class to retrieve all the products within a category.

Notice how the properties themselves are still POCO properties and do not require any specific Entity Framework collection type to define them.

6. Create a new class in the **Models** folder and name it “**NorthwindContext.cs**”.
7. Add the following **using** statement at the top of the file to include the **System.Data.Entity** namespace.

```
C#
using System.Data.Entity;
```

Note: **System.Data.Entity** is the default namespace for Entity Framework Code First.

8. Modify the **NorthwindContext** class definition to inherit from **DbContext**.

```
C#
public class NorthwindContext : DbContext
```

9. Add the following properties to define the **DbSet** collections for each entity set.

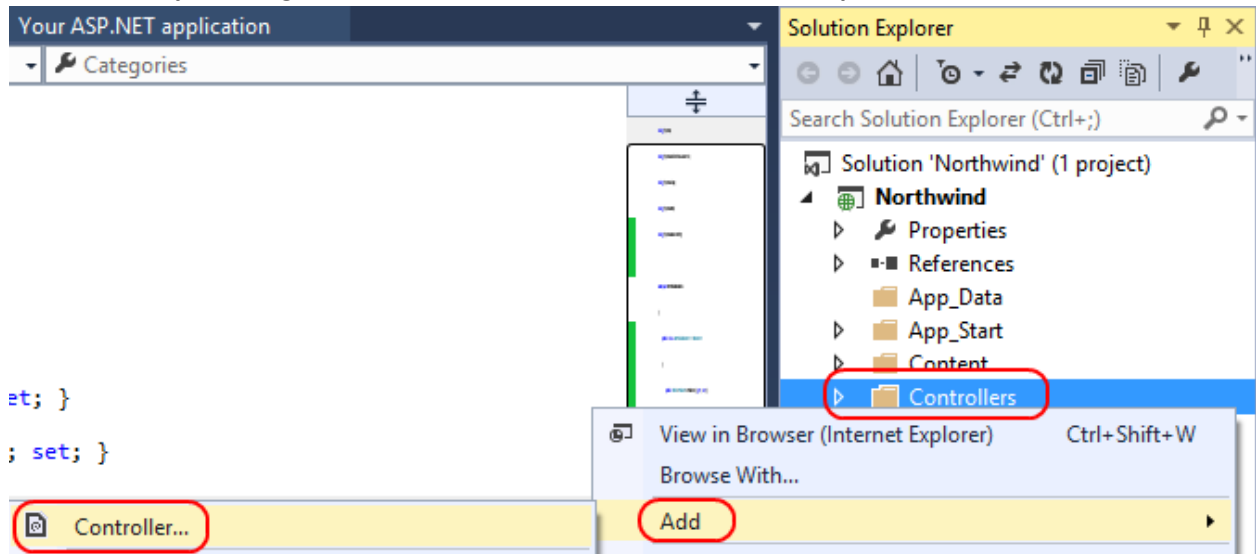
```
C#  
public DbSet<Product> Products { get; set; }  
public DbSet<Category> Categories { get; set; }
```

The **NorthwindContext** class is used to map the **Product** and **Category** classes to and from the database. It derives from **DbContext** and exposes two **DbSet** properties for the root entities of the model: **Products** and **Categories**. These sets are automatically initialized when the **NorthwindContext** class instance is created.

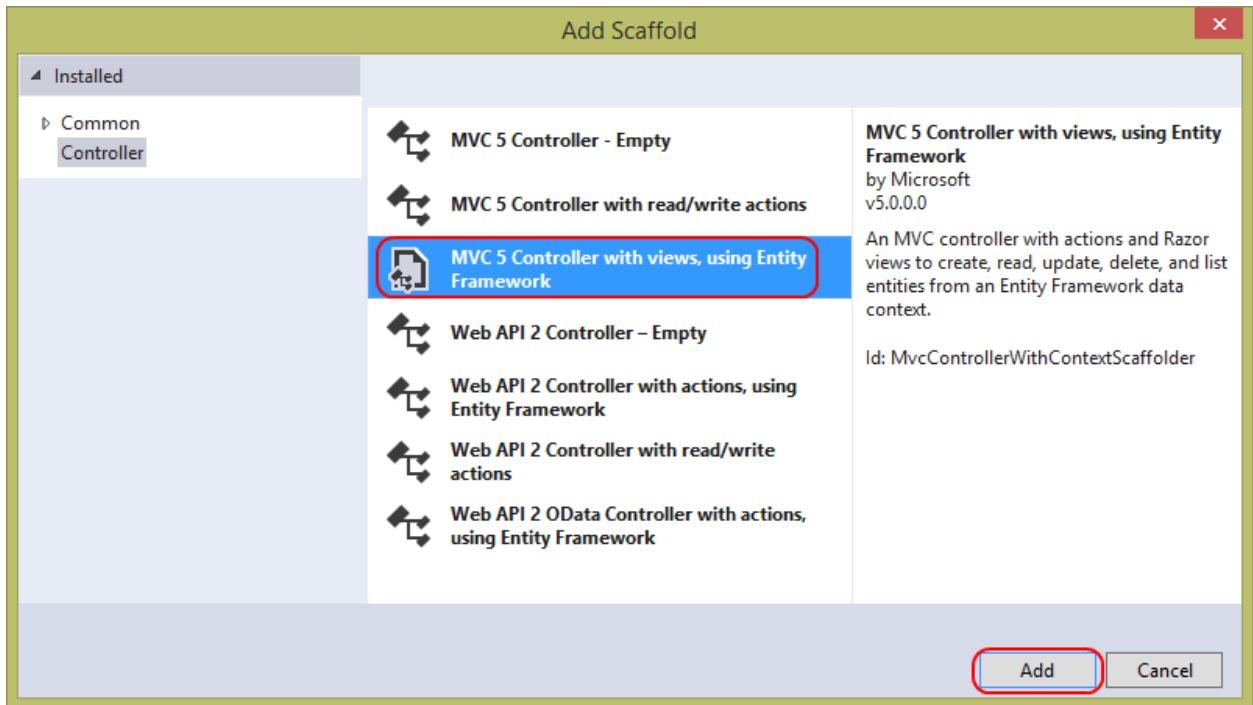
Task 3 – Generating a Controller and the Views from the Products entity

In this task, you will use MVC scaffolding to generate the controller, the actions, and the views for the **Product** entity.

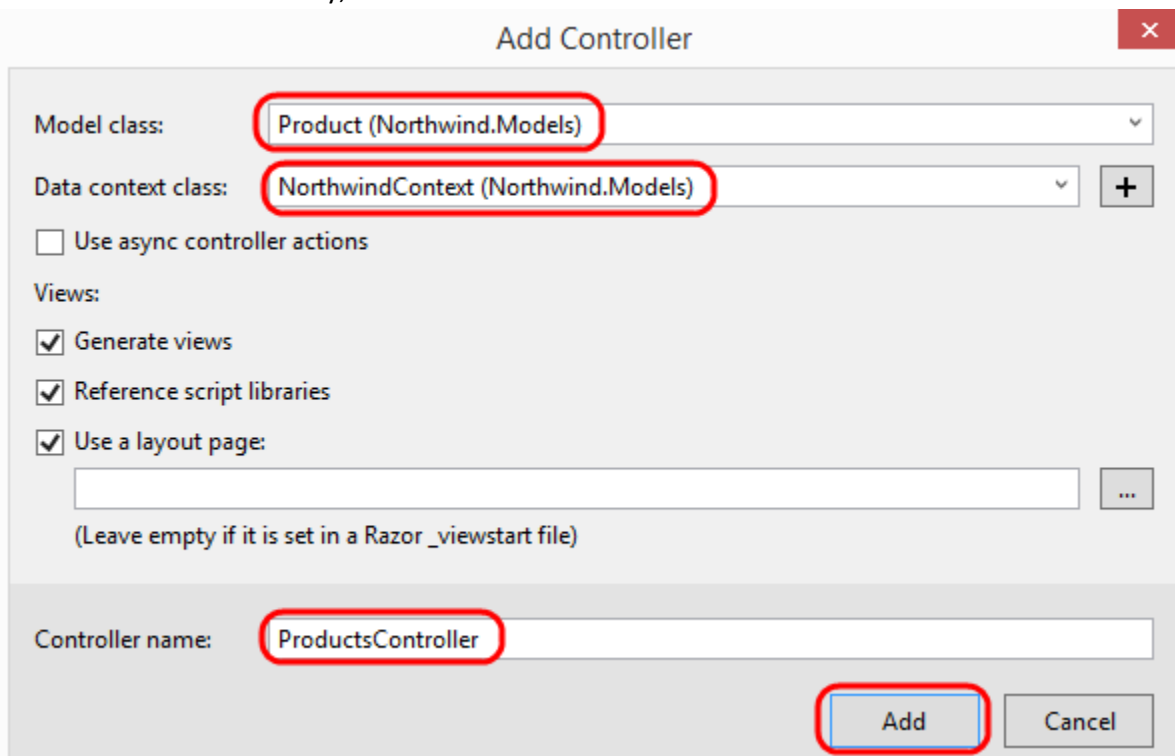
1. Select **Build | Build Solution** from the main menu to build the solution.
2. In **Solution Explorer**, right-click the **Controllers** folder and select **Add | Controller**.



3. In the **Add Scaffold** dialog, select **MVC 5 Controller with views, using Entity Framework** and click **Add**.

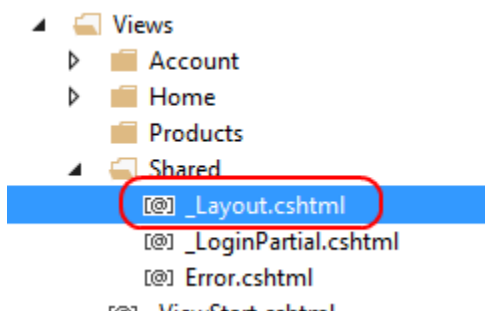


4. Set the **Controller name** to “**ProductsController**”. For the **Model class** field, begin to type “**Product**” and it will autocomplete for you. The same is true when setting the **Data context class** to “**NorthwindContext**”. Finally, click **Add**.



Note: The ASP.NET MVC scaffolding feature helps to generate CRUD methods for controllers, and create appropriate views. The scaffolding feature supports Entity Framework models, enabling ASP.NET MVC to generate working code that you can use immediately. Although the generated code might need some tweaking and modifications, it is still useful because it saves you some time.

5. Open the **_Layout.cshtml** file located in the **Views\Shared** folder.



6. Add a link to the **Index** action of the **Products** controller in the menu.

CSHTML

```
<li>@Html.ActionLink("Home", "Index", "Home")</li>
<!-- Insert this Products link after the existing Home link above. -->
<li>@Html.ActionLink("Products", "Index", "Products")</li>
```

Task 4 – Generating the Database

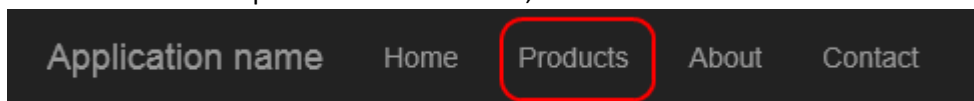
In this task, you will generate the database from your code by simply running the application. Then, you will verify that the generated database schema corresponds with the classes you created.

1. In **Solution Explorer**, double-click **Web.config** in the root of the MVC project.
2. Add the following XML snippet to the **<configuration>** section, just after the closing **</configSections>** tag.

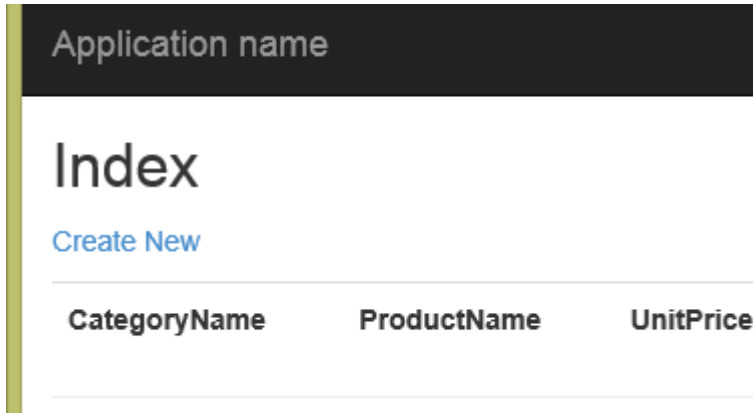
XML

```
<connectionStrings>
  <add name="NorthwindContext" connectionString="Data
Source=(LocalDb)\v11.0;AttachDbFilename=|DataDirectory|\aspnet-
Northwind.mdf;Initial Catalog=aspnet-Northwind;Integrated Security=True"
  providerName="System.Data.SqlClient" />
</connectionStrings>
```

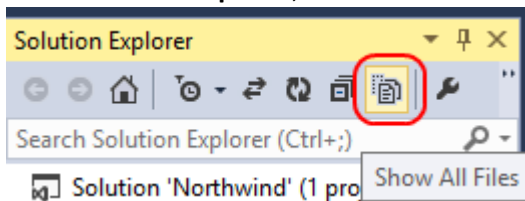
3. Press **F5** to run the application.
4. When the browser opens and the site loads, click the **Products** link on the menu.



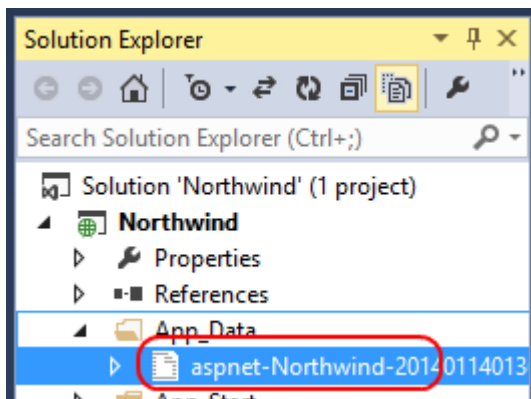
- When the **Products** page load, it will have column headers generated from the model, but since there is no data yet, the rows will be empty. You may notice a slight delay because by default, since the database file does not yet exist, Entity Framework Code First will create it.



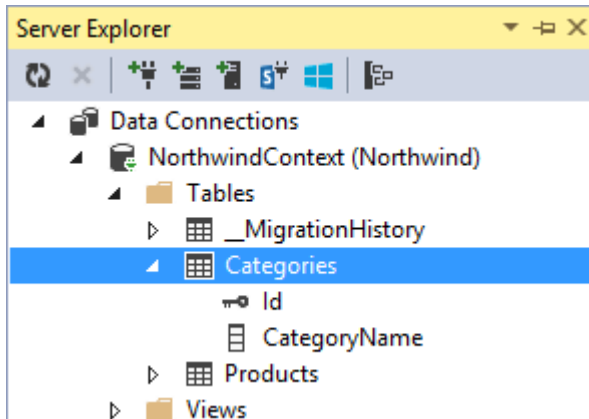
- Close the browser and return to **Visual Studio**.
- In the **Solution Explorer**, click the **Show All Files** button to show all files.



- Expand the **App_Data** folder and double-click the **MDF** file inside it. This file is our database, and it was automatically generated by Entity Framework. Note that the exact file name may differ from the one in the screenshot.



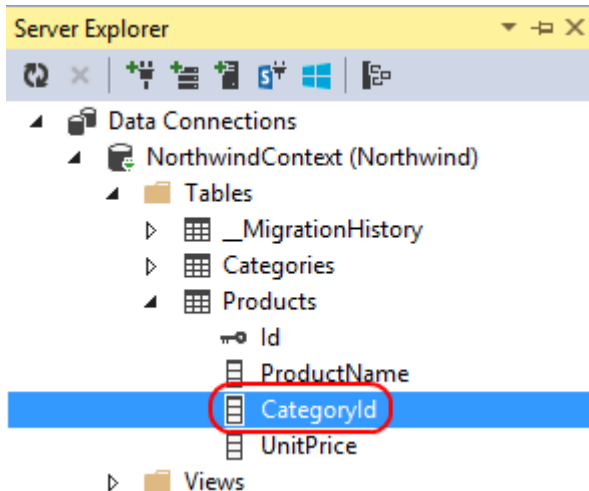
- The database will open in the **Server Explorer** under the **Data Connections** node. Expand **NorthwindContext | Tables | Categories** to see the columns generated for that table.



10. Explore the database schema by expanding the tables and its columns. Notice that the database schema matches the entities you created. You did not specify any custom persistence mapping rules; the database follows the default naming convention of Entity Framework Code First to map the schema.

Note: Entity Framework pluralizes the table names by default. In this article you can learn how to override this behavior: <http://msdn.microsoft.com/en-US/data/jj679962>

11. Notice the **CategoryId** foreign key in the **Products** table. Entity Framework automatically inferred the relationship from the **Category** property in the **Product** class.



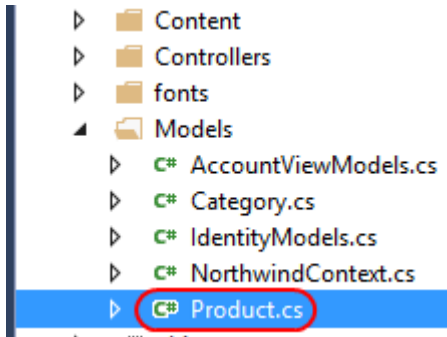
12. In **Server Explorer**, right-click **NorthwindContext** and select **Close Connection**. If you don't do this, Visual Studio will maintain a handle to the database and the Entity Framework will be unable to delete and recreate the database in future steps.

Task 5 – Adding Data Annotations

In this task, you will add data annotations to provide more information to Entity Framework (and to MVC 5) about the classes and the database to which they map. For example, you can specify that a property be used as the primary key, or you can set the length of a text field, which overrides the default length. These data annotations will also serve as validation rules for your model.

Entity Framework Code First allows you to use the **System.ComponentModel.DataAnnotations** namespace to provide additional information about classes, properties, and validation rules. Later on, you will see that these validation rules are also understood by a number of .NET frameworks, such as ASP.NET MVC or Dynamic Data, which enables you to leverage the same annotations for client-side and server-side validations.

1. Open the **Product.cs** class located in the **Models** folder.



2. Add the following using statement to include the **System.ComponentModel.DataAnnotations** namespace. This namespace provides attribute classes to define metadata for entity classes.

```
C#  
using System.ComponentModel.DataAnnotations;
```

3. Add the following attributes to the **ProductName** property. The **Required** annotation indicates that a property is required, and the **StringLength** annotation defines the maximum length of a string property.

```
C#  
[Display(Name="Name")]  
[Required(ErrorMessage="Product name must be specified")]  
[StringLength(50, ErrorMessage="Name too long")]  
public string ProductName { get; set; }
```

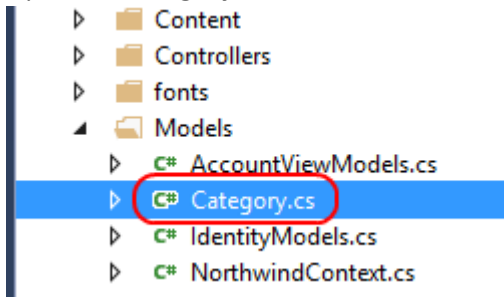
4. Add the following attributes to the **UnitPrice** property.

```
C#  
[Display(Name = "Price")]  
[Required(ErrorMessage = "Price must be specified")]  
[DataType(DataType.Currency)]  
public decimal UnitPrice { get; set; }
```

Note: The **Display** and the **DataType** annotations will be used only by ASP.NET MVC, and they do not affect the Entity Framework model. You can learn more about Data Annotations in this article:

<http://msdn.microsoft.com/en-US/data/jj591583>.

5. Open the **Category.cs** class located in the Models folder.



6. Add the following using statement to include the **System.ComponentModel.DataAnnotations** namespace.

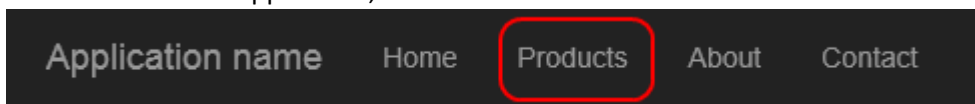
C#
`using System.ComponentModel.DataAnnotations;`

7. Add the following attribute to the CategoryName property.

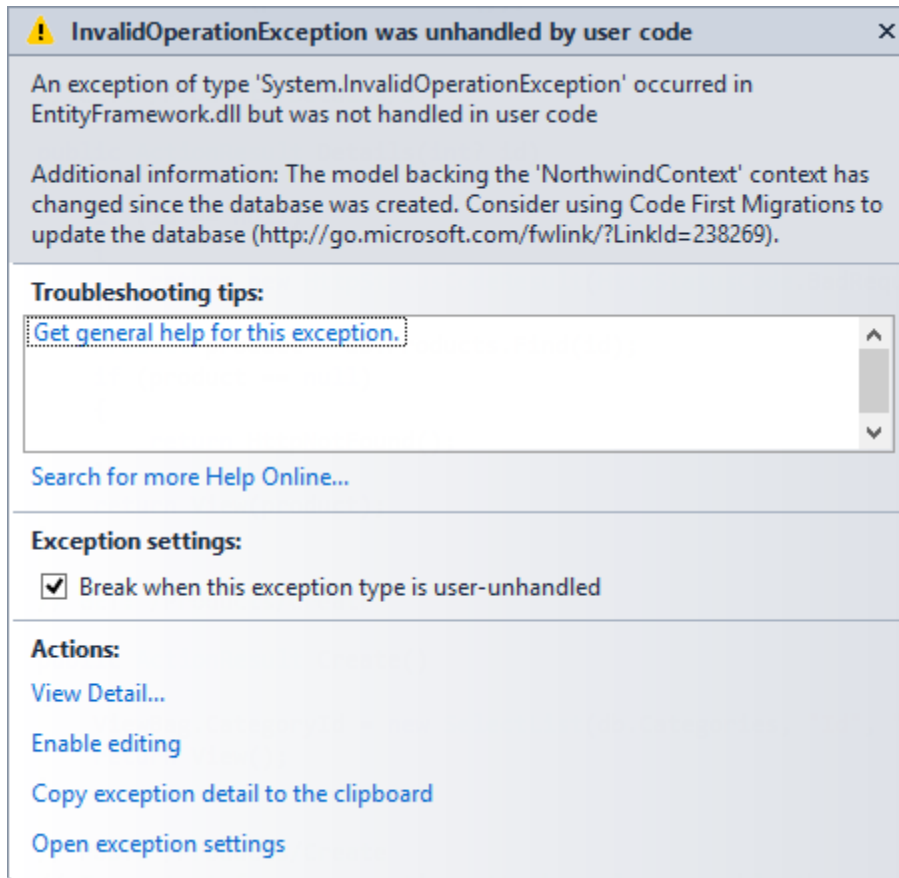
C#
`[Display(Name = "Category")]
public string CategoryName { get; set; }`

While Data Annotations are very flexible, keep in mind that they provide only the most commonly needed configuration changes for your Code First mapping. To configure your classes for some of the edge cases, you should look to an alternate configuration mechanism, the **Code First Fluent API**, which provides you with a way to describe configurations imperatively, in code.

8. Press **F5** to run the application, and click the **Products** menu item.



Notice that an exception is thrown when the application tries to use the Northwind context. This occurs because you have updated your entities and they do not match the database schema anymore. In the next task, you will fix this issue by configuring a database initializer.



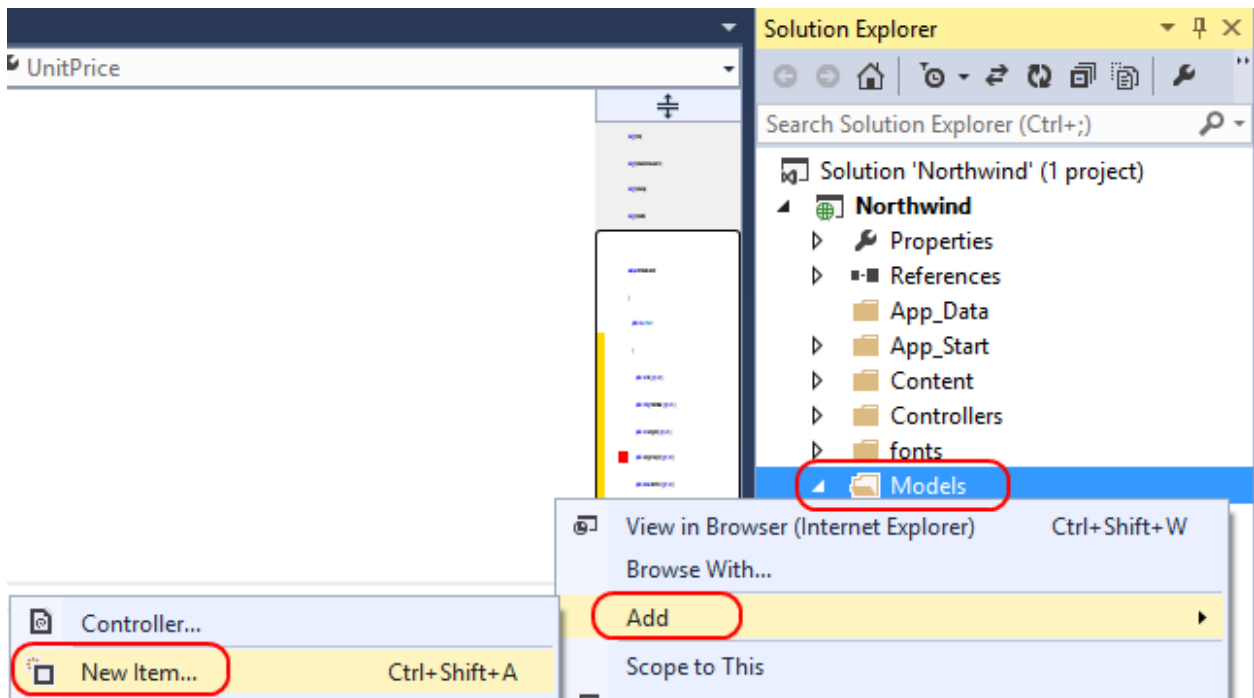
9. Close the browser.

Note: To learn more about Code First Fluent API, check out Exercise 5 from this lab.

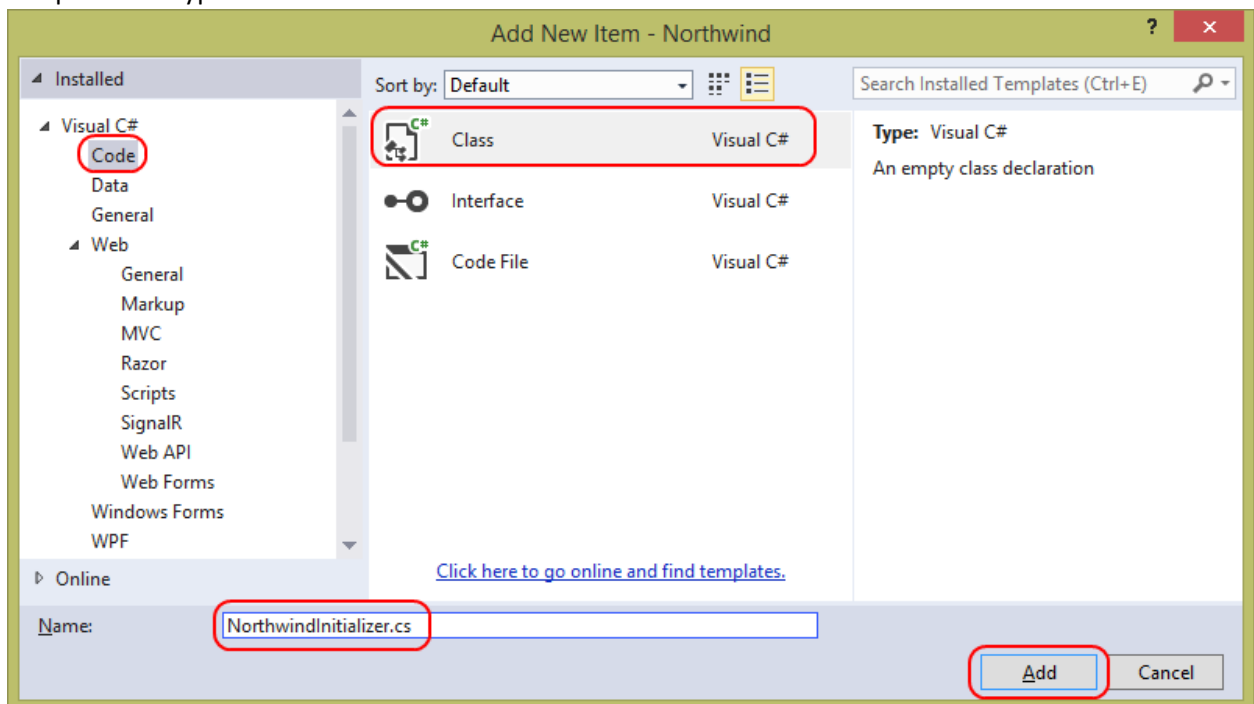
Task 6 – Creating a Database_INITIALIZER

In this task, you will configure a database initializer to automatically update the database schema and reflect the changes in the domain model. Within the initializer, you will implement a seed method to pre-populate the model with data when the database is created.

1. Right-click the **Models** directory and select **Add | New Item....**



2. In the **Add New Item** dialog, select the **Code** category from the left pane. Then select the **Class** template and type **"NorthwindInitializer.cs"** as the **Name** and click **Add**.



3. At the top of the file, add the following using declaration to reference the **System.Data.Entity** namespace.

```
C#
using System.Data.Entity;
```

4. Modify the **NorthwindInitializer** class to inherit from **DropCreateDatabaseIfModelChanges<NorthwindContext>**.

C#

```
public class NorthwindInitializer :  
DropCreateDatabaseIfModelChanges<NorthwindContext>
```

The **DropCreateDatabaseIfModelChanges<TContext>** class is an implementation of **IDatabaseInitializer<TContext>** that will delete, recreate, and optionally reseed the database with data only if the model has changed since the database was created. This is achieved by writing a hash of the store model to the database when it is created and then comparing that hash with one generated from the current model.

Note: Alternatively, you can use **CreateDatabaseIfNotExists<TContext>**, which recreates and optionally re-seeds the database with data only if the database does not exist. This is the default initializer when one is not specified. You can also use **DropCreateDatabaseAlways<TContext>**, which always recreates and optionally re-seeds the database with data the first time that a context is used in the application domain.

5. Implement an **AddCategory** function to insert categories into the database.

C#

```
private void AddCategory(NorthwindContext context, string categoryName)  
{  
    context.Categories.Add(  
        new Category()  
        {  
            CategoryName = categoryName  
        });  
}
```

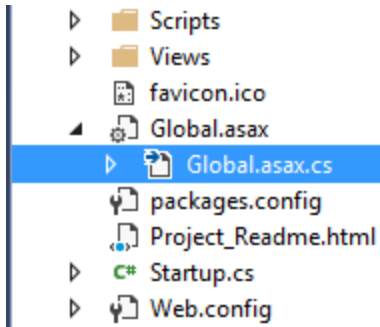
6. Override the **Seed** function and call the **AddCategory** function to pre-populate the **Categories** set with some sample data.

C#

```
protected override void Seed(NorthwindContext context)  
{  
    this.AddCategory(context, "Beverages");  
    this.AddCategory(context, "Condiments");  
    this.AddCategory(context, "Confections");  
    this.AddCategory(context, "Dairy Products");  
    this.AddCategory(context, "Grains/Cereals");  
    this.AddCategory(context, "Meat/Poultry");  
    this.AddCategory(context, "Produce");  
    this.AddCategory(context, "Seafood");  
}
```


The virtual **Seed** method is called after the database is created; enabling you to insert data after the database has been created. An instance of the context type is passed on to the method, so you can specify seed data in terms of your domain objects and use the context object to persist them.

7. Open the **Global.asax.cs** file.



8. Add the following using statements to include the **System.Data.Entity** and **Northwind.Models** namespaces.

```
C#  
using System.Data.Entity;  
using Northwind.Models;
```

9. Find the **Application_Start** function and add the following line of code at the end to enable the database initializer.

```
C#  
protected void Application_Start()  
{  
    ...  
    Database.SetInitializer<NorthwindContext>(new NorthwindInitializer());  
}
```

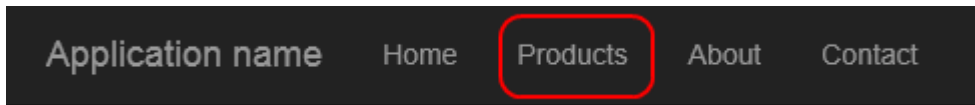
The **Database.SetInitializer<TContext>** method allows you to set the database initialization strategy that will be called when the **DbContext** instance is initialized. In web applications, the suggested place for initialization is the **Application_Start** method.

The feature shown above makes it very easy to evolve and refactor your code at development time without having to use tools or run scripts, which manually keep your database in sync with your code changes. However, this behavior is not intended for production scenarios where you want to “migrate” existing data from one schema to another. Instead, it is designed for development scenarios where you want the database to be quickly and automatically updated for you. It can also be beneficial for automated integration testing as the database can be recreated for each new test.

For scenarios where you are working with production data and want to incrementally evolve your database together with your Code First model, Entity Framework has a feature called Code First Migrations. You can learn more about it in this article: <http://msdn.microsoft.com/en-us/data/jj591621>

Task 7 – Verification

1. Press **F5** to run the Web application.
2. Go to the **Products** link on the menu.



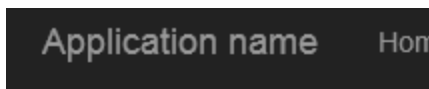
Notice the table headers now show the names that you specified in the **Display** data annotations.

Index

[Create New](#)

| Category | Name | Price |
|----------|------|-------|
|----------|------|-------|

3. Click **Create New**.



Index

[Create New](#)

Category

4. Go ahead and click **Create** without filling the text boxes.

Create

Product

Name

CategoryId

Price

Create

[Back to List](#)

You will see the validation error messages next to the fields. The errors are displayed for the fields that were annotated with the **Required** attribute.

Create

Product

Name

Product name must be specified

CategoryId

Price

Price must be specified

Create

[Back to List](#)

5. In the **Create** view, type **"Cookies"** as the **Name**, select **"Confections"** as the **CategoryId**, and type **"3"** as the **Price**. Click **Create** to create the record, which will also return you to the products list.

Create

Product

Name

CategoryId

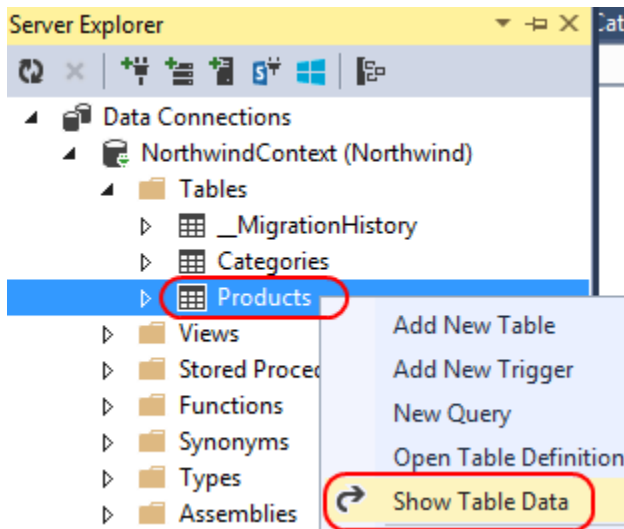
Price

Create

[Back to List](#)

Notice that the categories drop-down list is already populated with the categories you have inserted in the seed method of the database initializer.

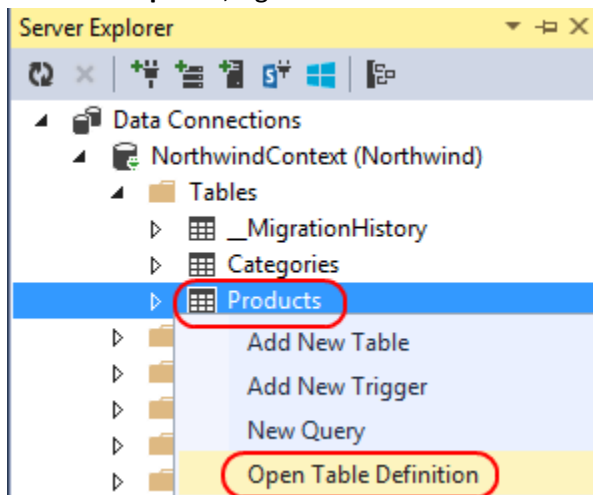
6. Close the browser, and go back to Visual Studio.
7. In **Server Explorer**, right-click on the **Products** table and select **Show Table Data**.



This will show the table data with the record we just added.

| Max Rows: 1000 | | | | |
|----------------|------|-------------|------------|-----------|
| | Id | ProductName | CategoryId | UnitPrice |
| | 1 | Cookies | 3 | 3.00 |
| * | NULL | NULL | NULL | NULL |

8. In **Server Explorer**, right-click the **Products** table and select **Open Table Definition**.



This will open the SQL used to define the table, where you can see the size of the **ProductName** column has been set to **50** based on the data annotation set in the class.

```

CREATE TABLE [dbo].[Products] (
    [Id] INT IDENTITY (1, 1) NOT NULL,
    [ProductName] NVARCHAR (50) NOT NULL,
    [CategoryId] INT NOT NULL,
    [UnitPrice] DECIMAL (18, 2) NOT NULL,
    CONSTRAINT [PK_dbo.Products] PRIMARY KEY CLUSTERED ([Id] ASC),
    CONSTRAINT [FK_dbo.Products_dbo.Categories_CategoryId] FOREIGN KEY ([CategoryId]) REFERENCES [dbo].[Categories] ([Id])
);

GO

CREATE NONCLUSTERED INDEX [IX_CategoryId]
ON [dbo].[Products]([CategoryId] ASC);

```

Note: ASP.NET MVC and Entity Framework Code First enable you to keep your validation and business rules in one place, within your model layer. This helps ensure that you do not duplicate validation/business logic as you add more Controllers and Views to your application.

Exercise 2: Using Database First

Another way to define the model of your entities is using the **Database First** workflow. In Database First, Entity Framework generates the model from an existing database and then uses code generation to create the classes

You should use this approach when your project is using legacy databases or any other external data source and you favor working with a visual designer. Entity Framework will not only generate the object classes for you, but also the mapping logic between the database tables and the entity classes.

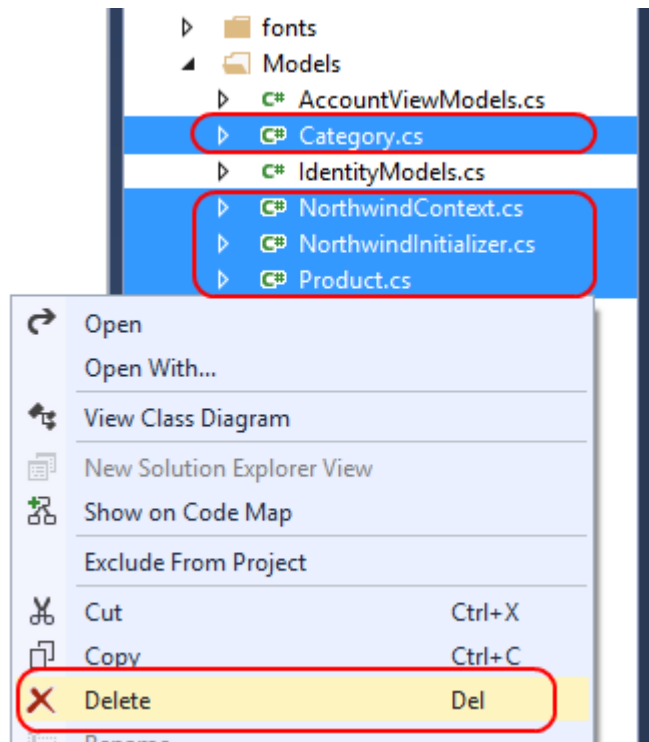
Throughout this exercise, you will learn how to use and configure Entity Framework to generate the model classes with Database First.

Task 1 – Creating the Model From the Database

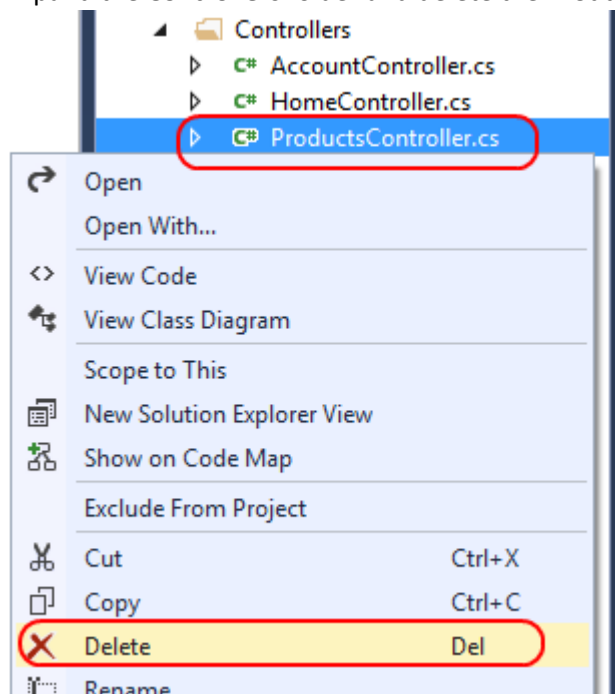
In this task, you will generate Entity Framework model classes, which represent data entities. First, you will remove the existent model classes from the previous exercise.

1. In **Solution Explorer**, expand the **Models** folder and delete the following classes:

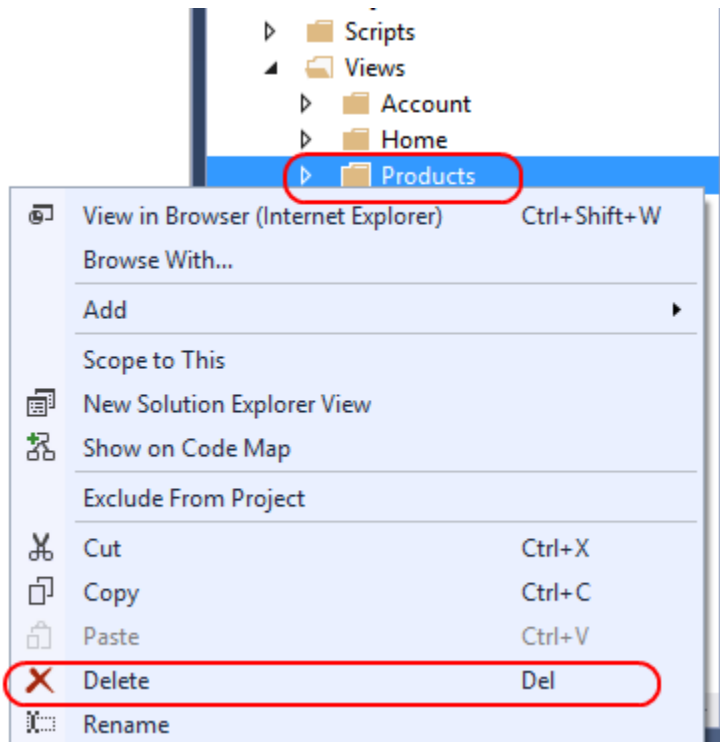
- **Category.cs**
- **NorthwindContext.cs**
- **NorthwindInitializer.cs**
- **Product.cs**



2. Expand the **Controllers** folder and delete the **ProductsController.cs** file.



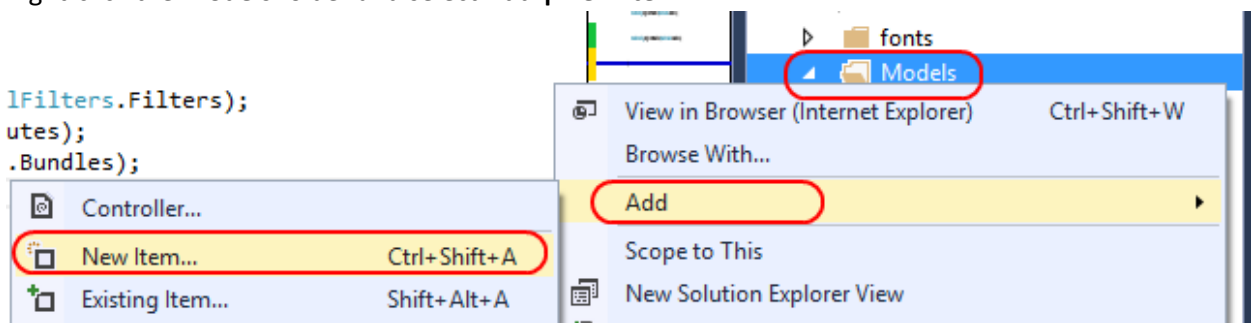
3. Expand the **Views** folder and delete the **Products** subfolder.



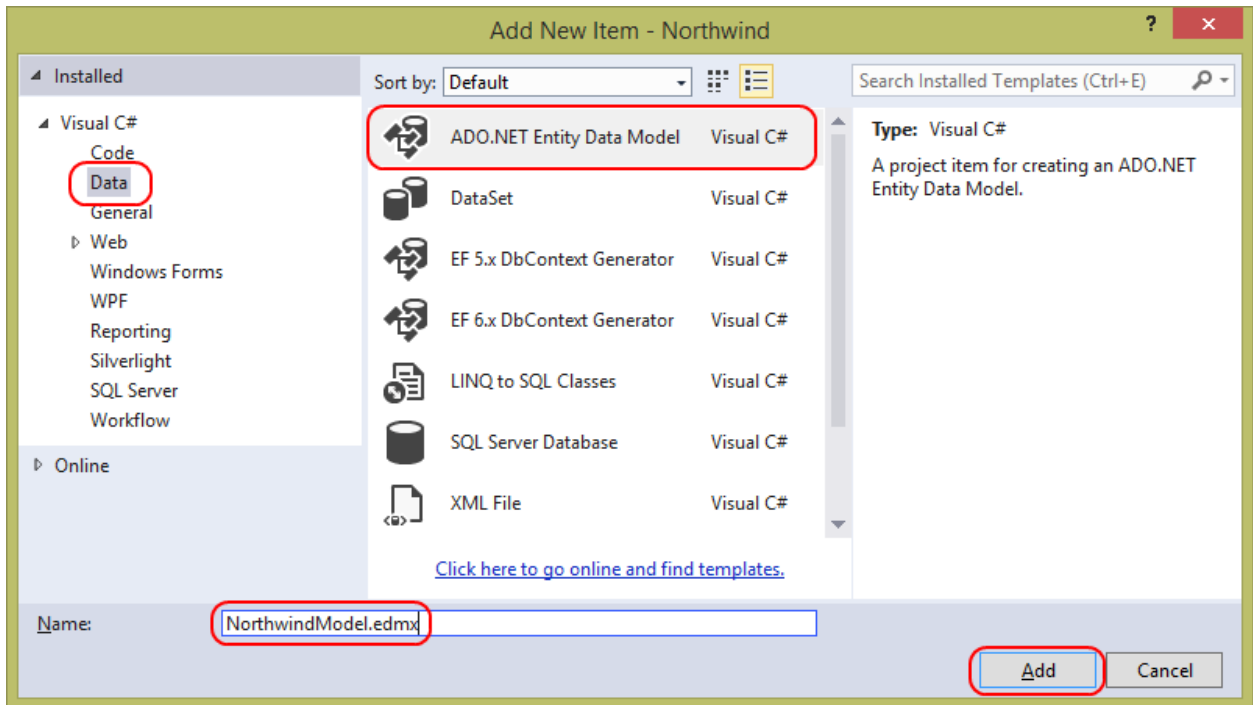
4. Open the **Global.asax.cs** file. In the **Application_Start** method, remove the line of code that calls **Database.SetInitializer**.

Note: The Database First approach does not use database initialization strategies from Code First. The approach assumes that you will use an existing database.

5. Right-click the **Models** folder and select **Add | New Item**.

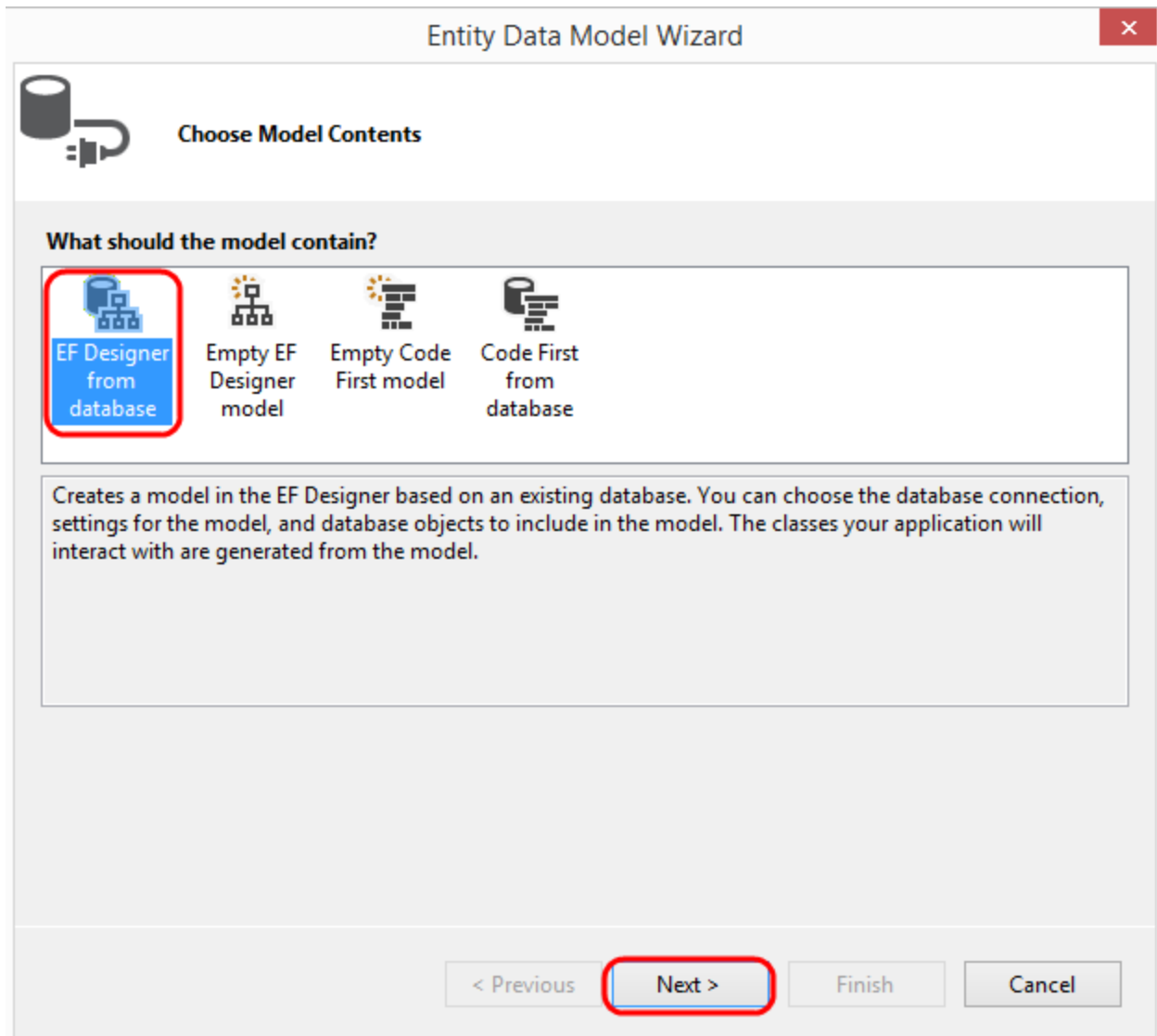


6. In the **Add New Item** dialog, select **Data** in the **Visual C#** node, and then select **ADO.NET Entity Data Model**. Name it "**NorthwindModel.edmx**". Click **Add** to open the **Entity Data Model Wizard**.

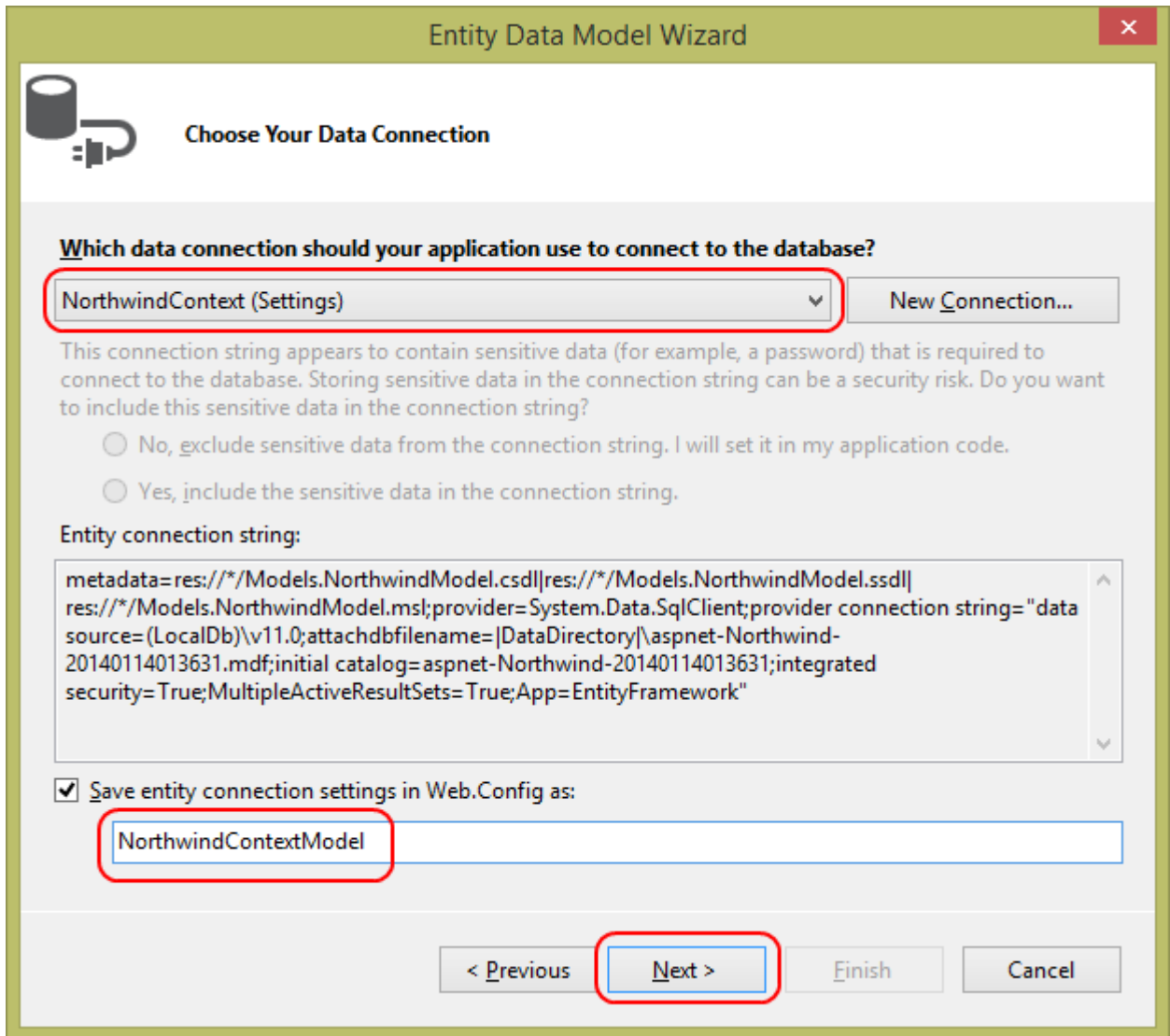


Note: The Entity Data Model (EDM) is a set of concepts that describe the structure of your data, regardless of its stored form. The Entity Designer works with an .edmx file, which is the combination of three metadata files: the conceptual schema definition language (CSDL), store schema definition language (SSDL), and mapping specification language (MSL) files. For more information, check out this article: [http://msdn.microsoft.com/en-us/library/bb399604\(v=VS.90\).aspx](http://msdn.microsoft.com/en-us/library/bb399604(v=VS.90).aspx).

7. In the **Choose Model Contents** step, select **EF Designer from database** to specify that you will create the model using the Database First approach. Click **Next** to continue.



8. In the **Choose your Data Connection** step, choose **NorthwindContext (Settings)** and type the name **"NorthwindContextModel"** as the key to use in Web.config. Click **Next** to continue. This will create an entry in the Web.Config with the connection string. For this exercise, you'll continue to use the database created in the previous exercise.



The screenshot shows the 'Entity Data Model Wizard' window, specifically the 'Choose Your Data Connection' step. The window has a title bar with a close button. Below the title bar is a header area with a database icon and the text 'Choose Your Data Connection'. The main content area asks 'Which data connection should your application use to connect to the database?'. A dropdown menu is set to 'NorthwindContext (Settings)', which is highlighted with a red rectangle. To the right of the dropdown is a 'New Connection...' button. Below this, a warning message states: '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?'. There are two radio buttons: 'No, exclude sensitive data from the connection string. I will set it in my application code.' and 'Yes, include the sensitive data in the connection string.'. Below the radio buttons is a text box labeled 'Entity connection string:' containing a long connection string. At the bottom, there is a checkbox labeled 'Save entity connection settings in Web.Config as:' which is checked. Next to it is a text box containing 'NorthwindContextModel', also highlighted with a red rectangle. At the very bottom are four buttons: '< Previous', 'Next >' (highlighted with a red rectangle), 'Finish', and 'Cancel'.

Entity Data Model Wizard

Choose Your Data Connection

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

NorthwindContext (Settings) 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.

Entity connection string:

```

metadata=res://*/Models.NorthwindModel.csdl|res://*/Models.NorthwindModel.ssdl|
res://*/Models.NorthwindModel.msl;provider=System.Data.SqlClient;provider connection string="data
source=(LocalDb)\v11.0;attachdbfilename=|DataDirectory|\aspnet-Northwind-
20140114013631.mdf;initial catalog=aspnet-Northwind-20140114013631;integrated
security=True;MultipleActiveResultSets=True;App=EntityFramework"

```

☒ Save entity connection settings in Web.Config as:

NorthwindContextModel

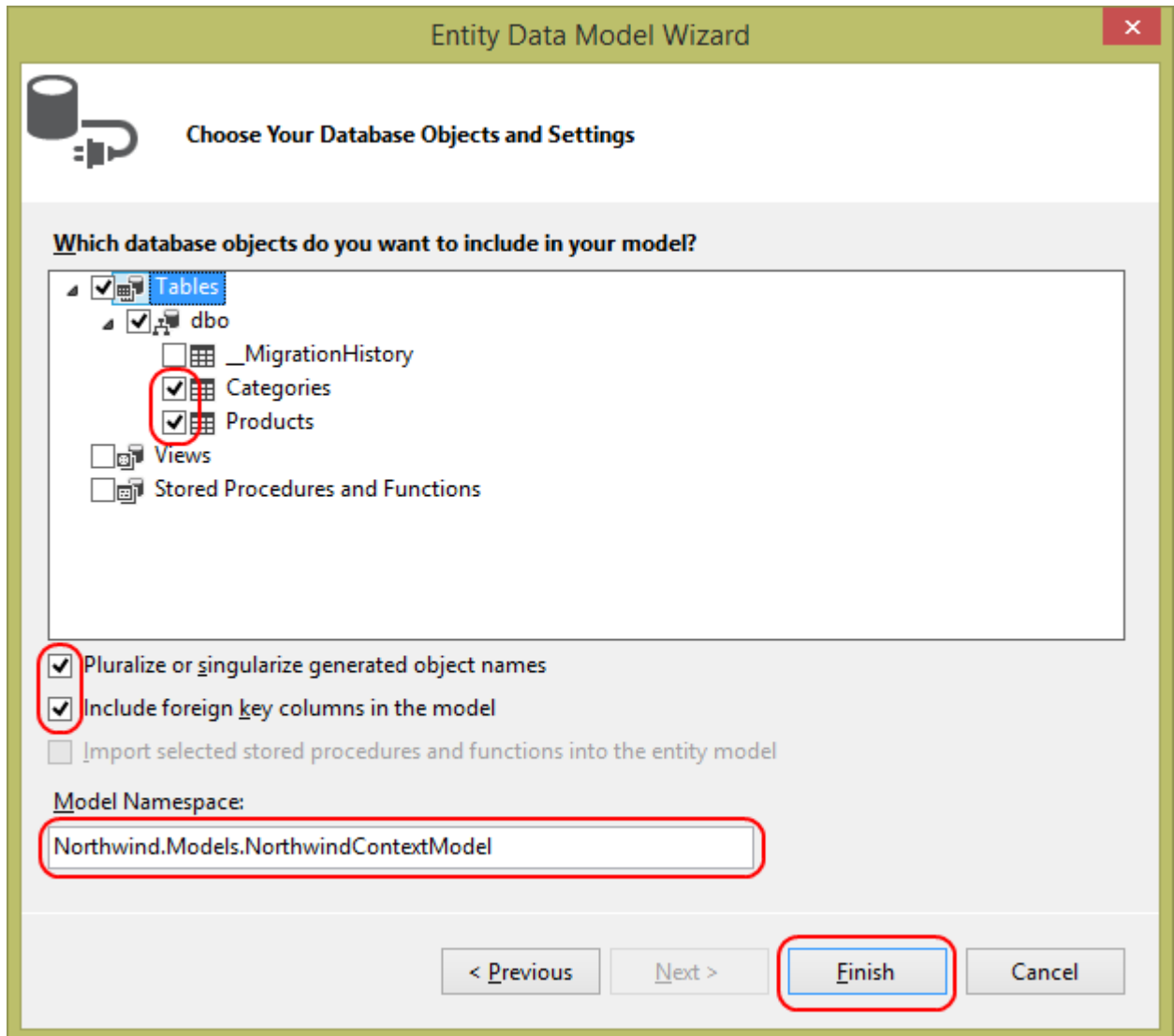
< Previous Next > Finish Cancel

9. In the **Choose Your Database Objects and Settings** step, expand **Tables | dbo** and select **Categories** and **Products**. The **_MigrationHistory** table, generated by Code First, should not be selected.
10. Make sure the **Pluralize or singularize generated object names** option is selected. When selected, Entity Framework will rename your entities according to the following rules:
 - The **EntityType** names, e.g. **Product**, are singular.
 - The **EntitySet** names, e.g. **Categories**, are plural.
 - The **NavigationProperty** types are singular when they return one entity and plural when they return more than one entity.

If you do not select this option, and the database tables have plural names, Entity Framework will generate plural entity classes.

11. Make sure the **Include foreign key columns in the model** option is selected. When selected, Entity Framework will generate properties on entity types that correspond to foreign key columns in the database.

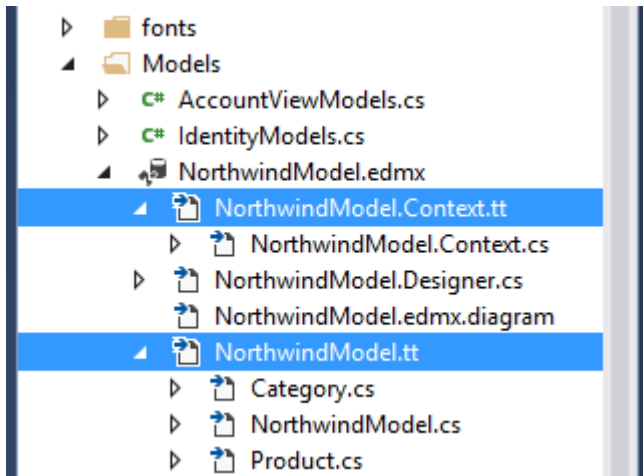
12. Set the **Model Namespace** to “**Northwind.Models.NorthwindContextModel**”.
13. Click **Finish** to create the model.



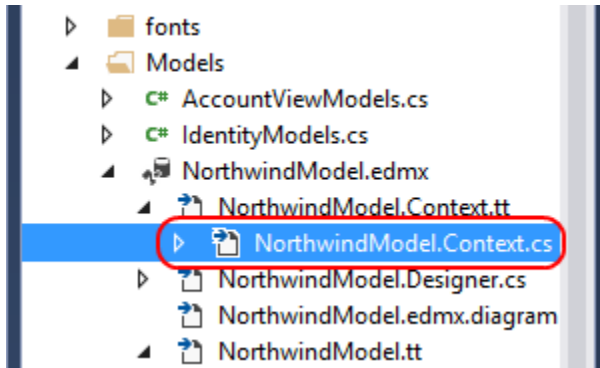
The image shows the 'Entity Data Model Wizard' dialog box with the title 'Entity Data Model Wizard'. The main heading is 'Choose Your Database Objects and Settings'. Below this, the question 'Which database objects do you want to include in your model?' is displayed. A tree view shows 'Tables' selected, with 'dbo' expanded. Under 'dbo', the following objects are listed: '_MigrationHistory' (unchecked), 'Categories' (checked), 'Products' (checked), 'Views' (unchecked), and 'Stored Procedures and Functions' (unchecked). Below the tree view, there are three checkboxes: 'Pluralize or singularize generated object names' (checked), 'Include foreign key columns in the model' (checked), and 'Import selected stored procedures and functions into the entity model' (unchecked). The 'Model Namespace' field is a text box containing 'Northwind.Models.NorthwindContextModel'. At the bottom, there are four buttons: '< Previous' (disabled), 'Next >' (disabled), 'Finish' (active), and 'Cancel' (disabled). Red circles highlight the 'Categories' and 'Products' checkboxes, the 'Pluralize or singularize generated object names' checkbox, the 'Model Namespace' text box, and the 'Finish' button.

Entity Framework will generate an **edmx** model file. Notice that the entities have singular names (**Category-Product**). In addition, notice that the **Product** entity inherits a **CategoryId** as the result of the one-to-many association between categories and products.

14. In **Solution Explorer**, expand **Models | NorthwindModel.edmx** as well as its **NorthwindModel.Context.tt** and **NorthwindModel.tt** child nodes. Notice that **Visual Studio** generated the **NorthwindContextModel**, **NorthwindModel**, **Category**, and **Product** classes.



15. Open **NorthwindModel.Context.cs**.



Notice that **NorthwindContextModel** is based on **DbContext** and that the code to create the classes is nearly identical to the first exercise.

```

namespace Northwind.Models
{
    using System;
    using System.Data.Entity;
    using System.Data.Entity.Infrastructure;

    1 reference
    public partial class NorthwindContextModel : DbContext
    {
        0 references
        public NorthwindContextModel()
            : base("name=NorthwindContextModel")
        {
        }

        0 references
        protected override void OnModelCreating(DbModelBuilder modelBuilder)
        {
            throw new UnintentionalCodeFirstException();
        }

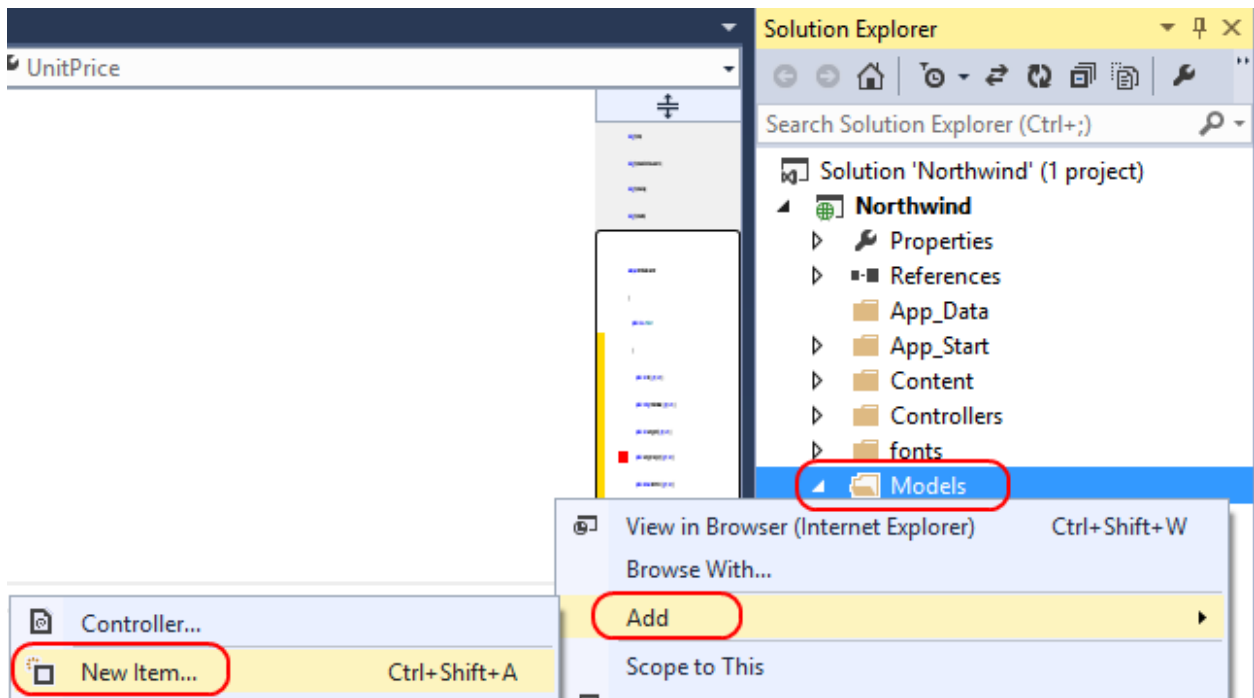
        0 references
        public virtual DbSet<Category> Categories { get; set; }
        0 references
        public virtual DbSet<Product> Products { get; set; }
    }
}

```

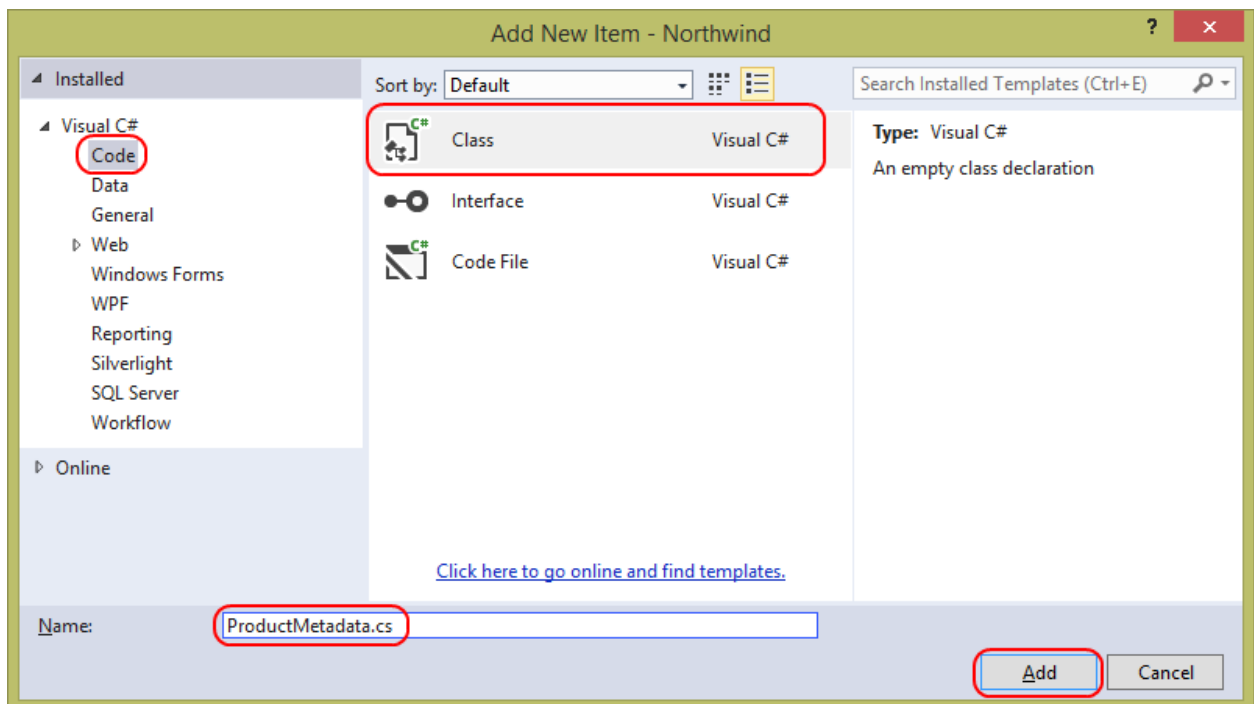
Task 2 – Adding Metadata

You should not edit the code generated by Entity Framework, as the changes applied to these files will be lost if you regenerate the model. For that reason, the entity classes are defined as partial so that you can extend the entities by adding other partial classes. In the next steps, you will use this strategy to add data annotations.

1. Right-click the **Models** folder and select **Add | New Item....**



2. Select the **Code** category from the left pane and select the **Class** template. Set the **Name** to "**ProductMetadata.cs**" and click **Add**.



As the model entity classes generated by Entity Framework do not include data annotations, you will now add the data annotations in a separate partial class.

3. In **ProductMetadata.cs**, add the following using statement.

C#

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

4. Add the **ProductName** and **UnitPrice** properties including the data annotations (same as Exercise 1) as shown in the following code snippet.

C#

```
[Display(Name = "Name")]
[Required(ErrorMessage = "Product name must be specified")]
[StringLength(50, ErrorMessage = "Name too long")]
public object ProductName { get; set; }

[Display(Name = "Price")]
[Required(ErrorMessage = "Price must be specified")]
[DataType(DataType.Currency)]
public object UnitPrice { get; set; }
```

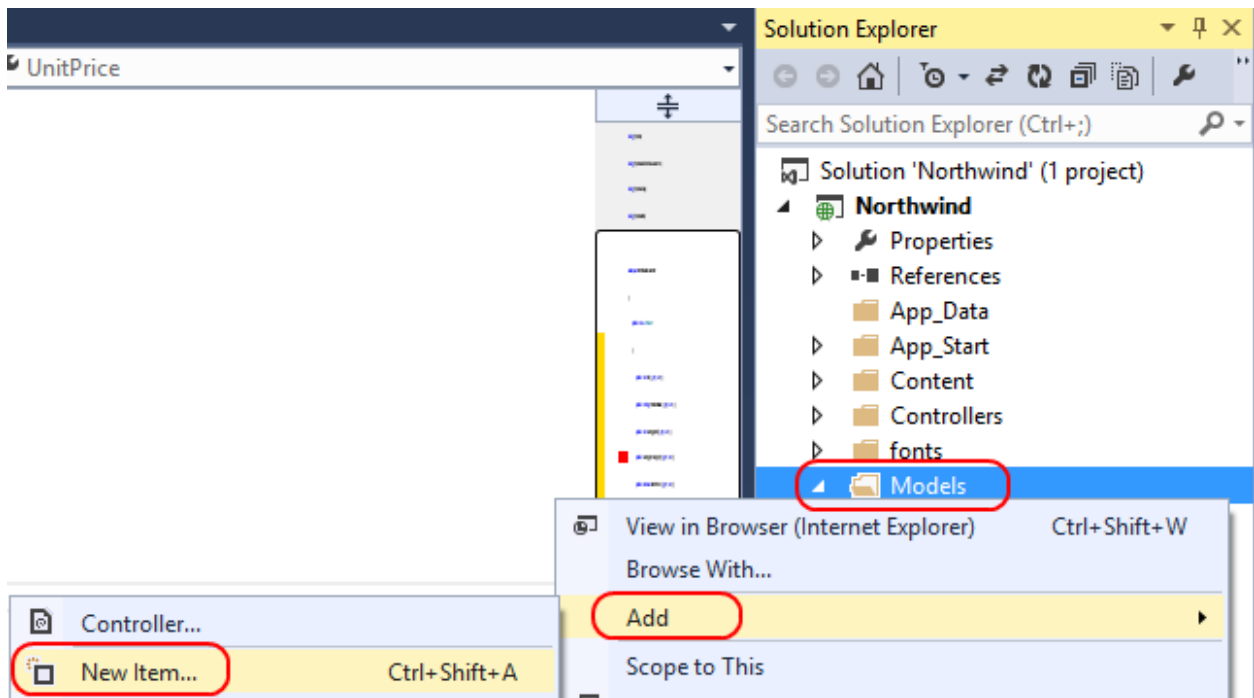
Note: It's not necessary to specify the type for each of the properties as in the entity class; the properties are defined using the `Object` class. This is because Entity Framework matches each property only by its name.

5. In the same file, add the following definition of a **Product** partial class at the end of the file, just before the namespace closing brace. It includes a **MetadataType** attribute that provides the **ProductMetadata** class type as argument. The **MetadataTypeAttribute** attribute enables you to associate a class with a data-model partial class. In this associated class, you provide additional metadata information that is not in the data model.

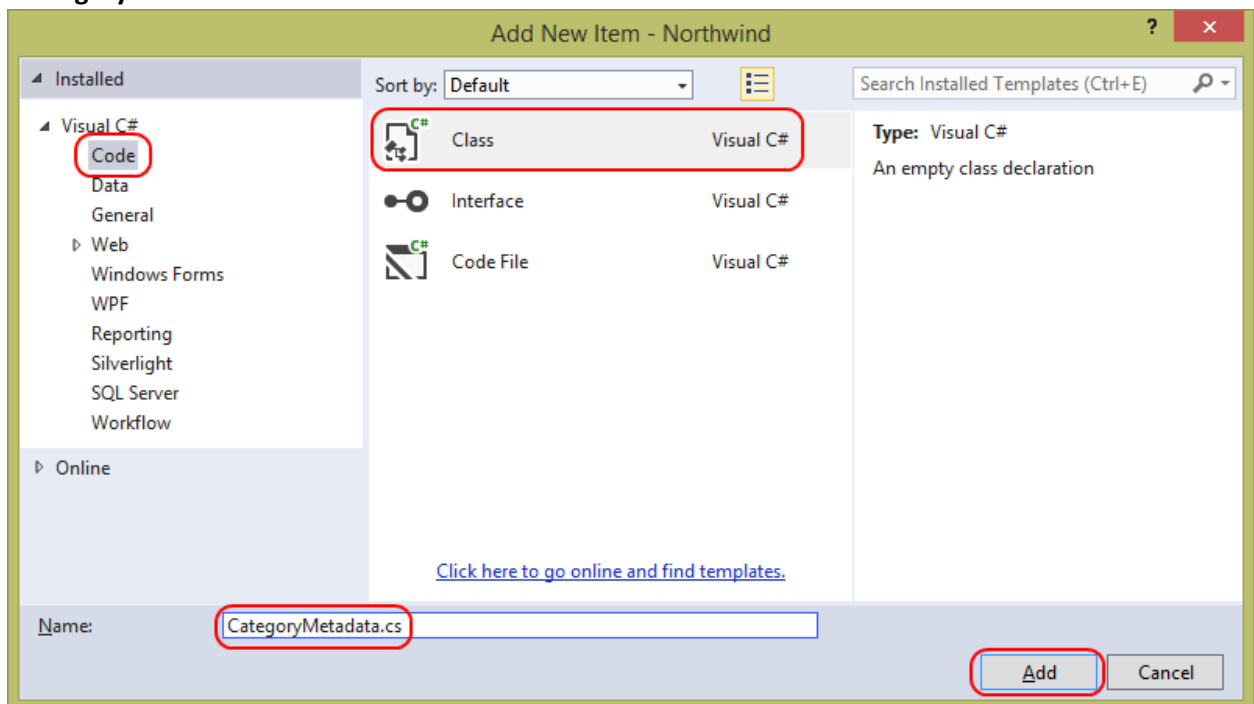
C#

```
[MetadataType(typeof(ProductMetadata))]
public partial class Product
{
}
```

6. Right-click the **Models** folder and select **Add | New Item....**



7. Select the **Code** category from the left pane and select the **Class** template. Set the **Name** to "**CategoryMetadata.cs**" and click **Add**.



8. In **CategoryMetadata.cs**, add the following using statement:

C#
 using System.ComponentModel.DataAnnotations;

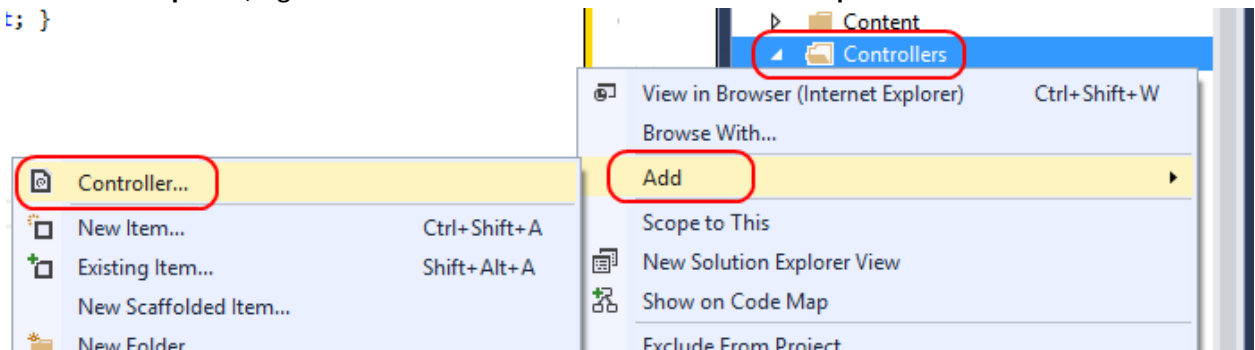
9. Add the following data annotations to the **CategoryMetadata** class.

```
C#  
[Display(Name = "Category")]  
public object CategoryName { get; set; }
```

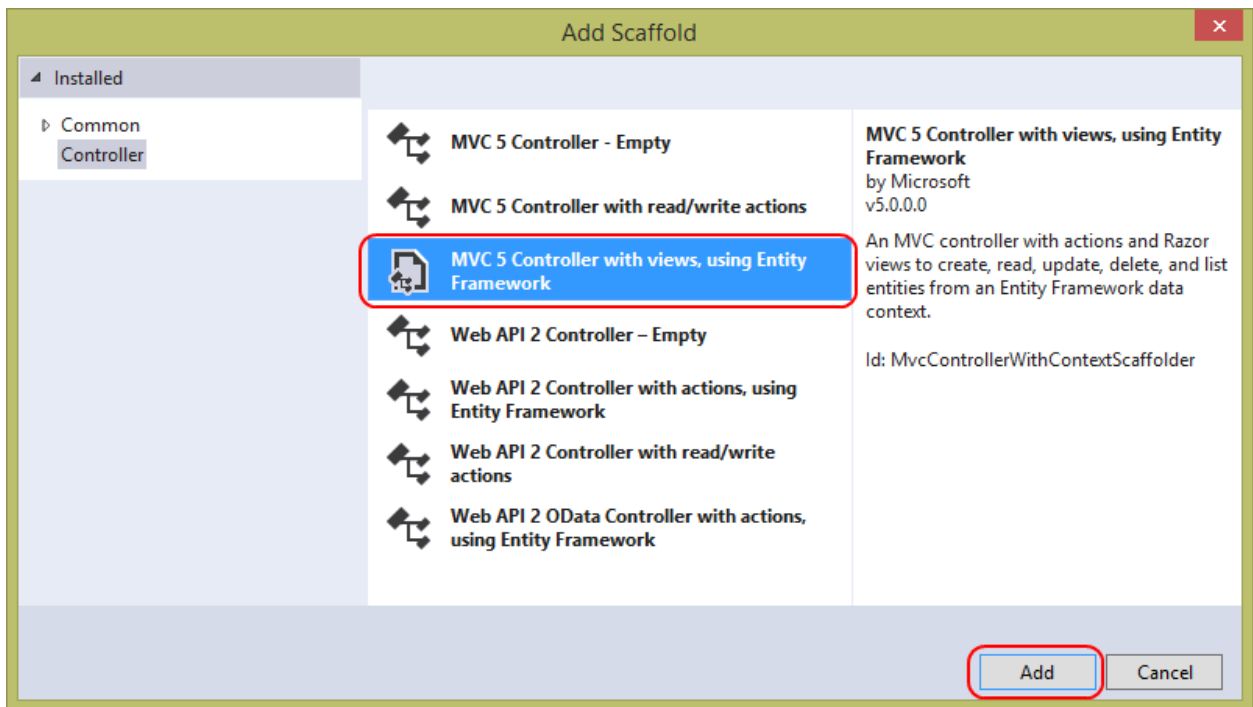
10. Add the definition of a **Category** partial class near the end of the same file, right inside the closing brace. Note that the **MetadataTypeAttribute** is used here as well.

```
C#  
[MetadataType(typeof(CategoryMetadata))]  
public partial class Category  
{  
}
```

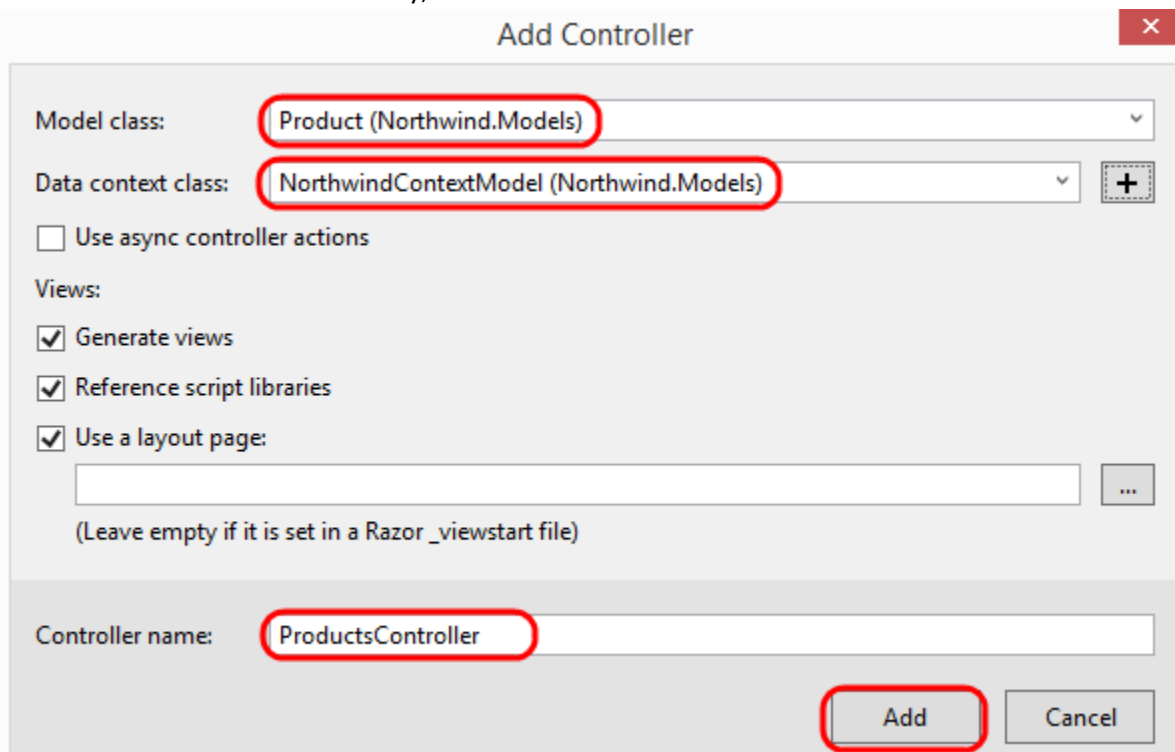
11. Select **Build | Build Solution** from the main menu to build the solution.
12. In **Solution Explorer**, right-click the **Controllers** folder and select **Add | Controller**.



13. In the **Add Scaffold** dialog select the **MVC 5 Controller with views, using Entity Framework** and click **Add**.

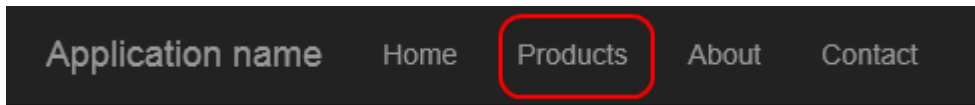


14. Set the **Controller name** to “**ProductsController**”. For the **Model class** field, begin to type “**Product**” and it will autocomplete for you. The same is true when setting the **Data context class** to “**NorthwindContextModel**”. Finally, click **Add**.



Task 3 – Verification

1. Press **F5** to run the application.
2. Click **Products**.



3. In the products page, click **Create New**.

Index

[Create New](#)

| Name | Price | Category |
|---------|--------|-------------|
| Cookies | \$3.00 | Confections |

4. In the **Create** view, type **“Cupcakes”** as the **Name**, select **“Confections”** as the **CategoryId**, and type **“4”** as the **Price**. Click **Create** to create the record, which will also return you to the products list.

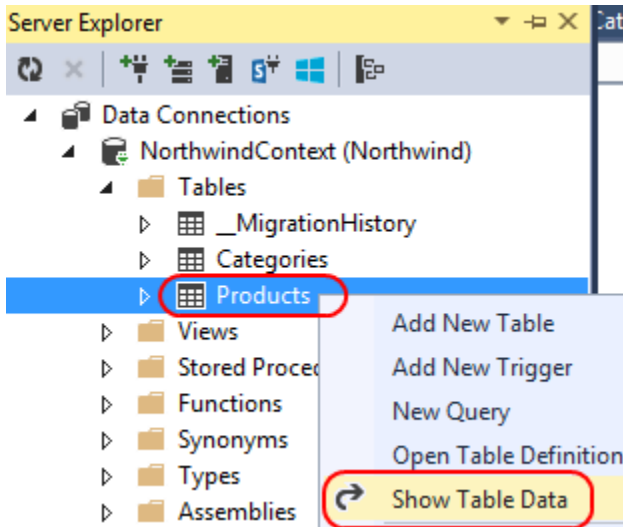
Create

Product

| | |
|------------|--|
| Name | <input type="text" value="Cupcakes"/> |
| CategoryId | <input type="text" value="Confections"/> |
| Price | <input type="text" value="4"/> |
| | <input type="button" value="Create"/> |

[Back to List](#)

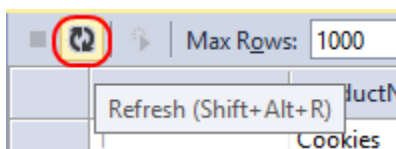
9. Close the browser, and go back to Visual Studio.
10. In **Server Explorer**, right-click on the **Products** table and select **Show Table Data**.



This will show the table data with the record we just added.

| | Id | ProductName | CategoryId | UnitPrice |
|----|------|-------------|------------|-----------|
| | 1 | Cookies | 3 | 3.00 |
| | 2 | Cupcakes | 3 | 4.00 |
| ▶* | NULL | NULL | NULL | NULL |

- If you still have the table open from the previous exercise, you'll need to click the **Refresh** button to refresh the data.

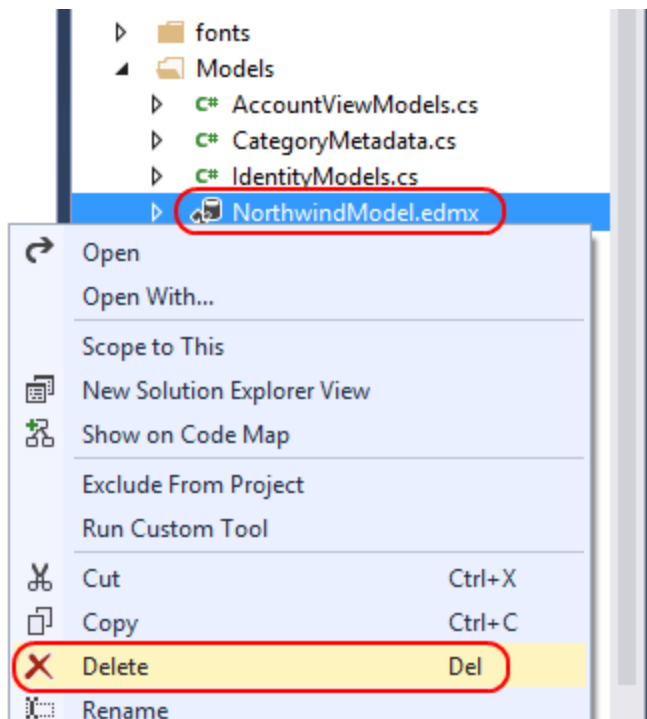


Exercise 3: Using Model First

In the **Model First** approach, Entity Framework starts from a conceptual model and generates the database structure, as well as the model entities. In Model First, you describe your entities in the Entity Framework Designer, which enables you to visually create and modify entities, associations, mappings, and inheritance relationships. This approach is ideal for those who are familiar with database design and the entity-relationship model and prefer to work in a visual designer. It can also be advantageous in the initial stages of development, as the **Entity Framework Designer** view facilitates business domain modeling.

Task 1 – Creating the Data Model

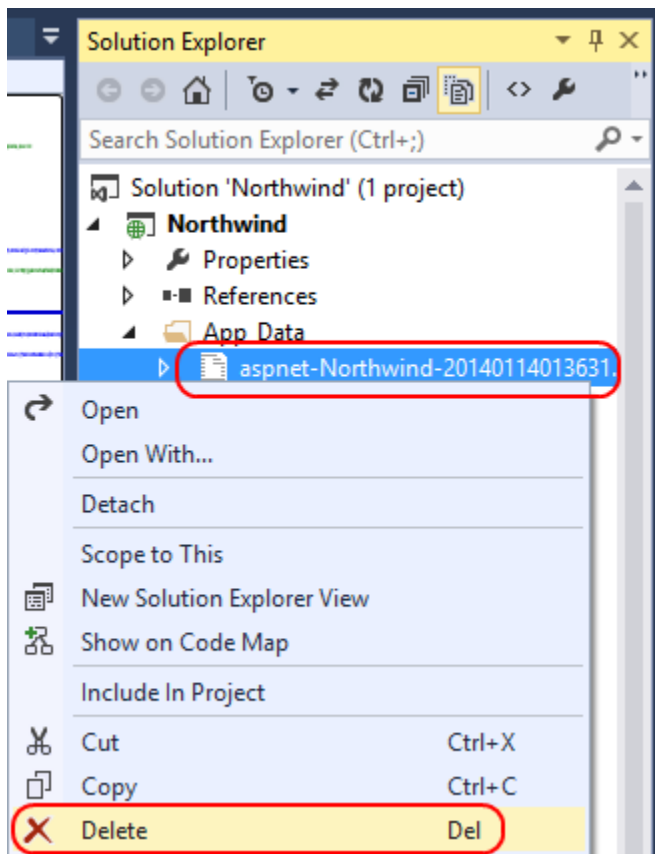
- In **Solution Explorer**, delete **NorthwindModel.edmx** from the **Models** folder.



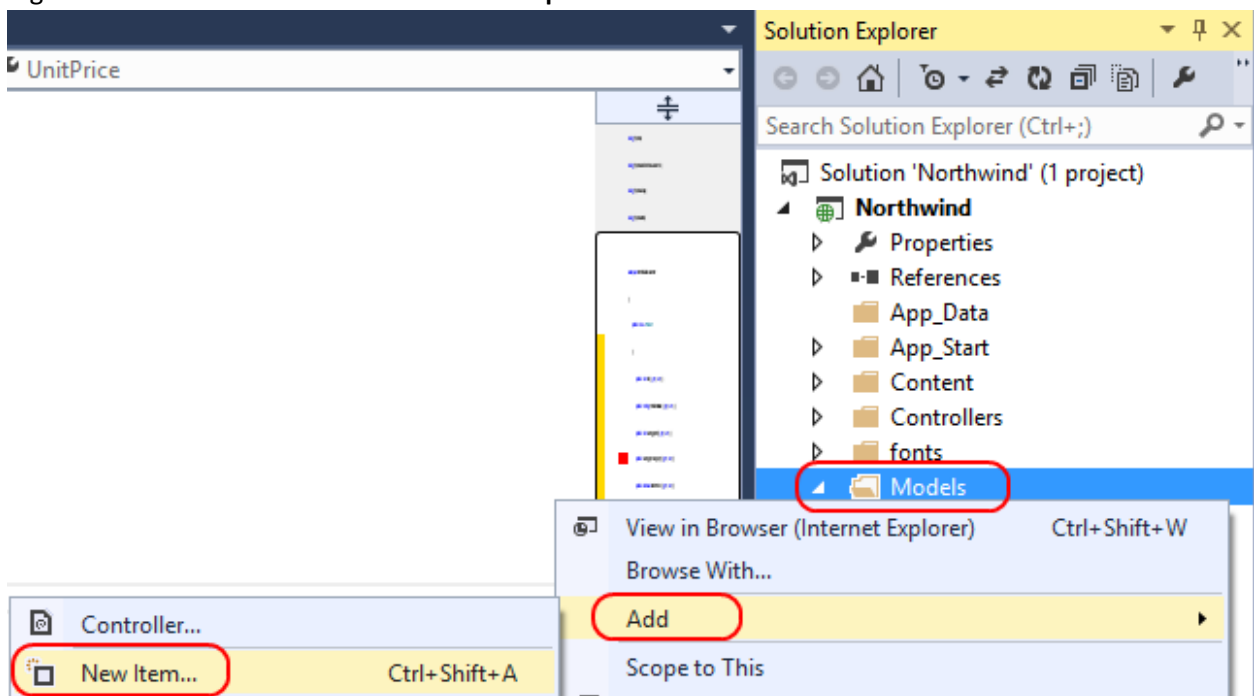
2. Open **Web.Config** from the project root and delete the **NorthwindContextModel** connection string, located in the **configuration\connectionStrings** section.

```
<configuration>
  <configSections>
    <section name="entityFramework" type="System.
    <!-- For more information on Entity Framework
  </configSections>
  <connectionStrings>
    <add name="NorthwindContext" connectionString=
    <add name="NorthwindContextModel" connectionString=
  </connectionStrings>
```

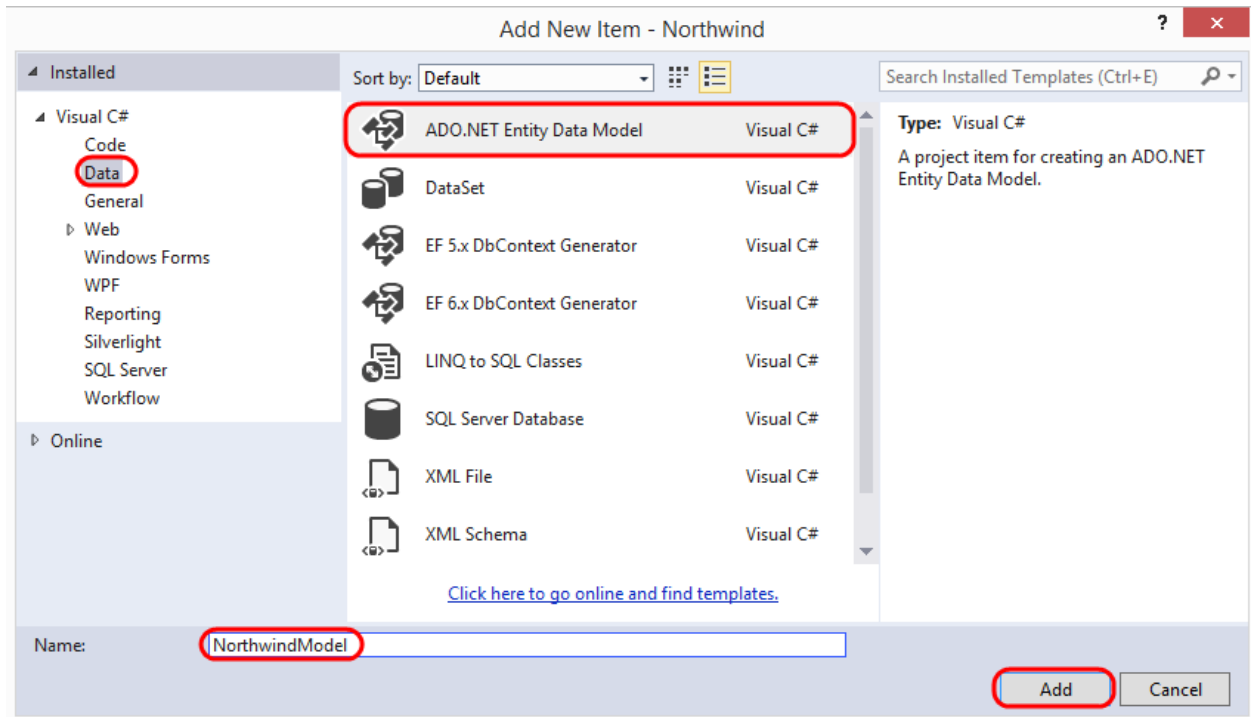
3. From the **App_Data** folder, delete the **MDF** database created earlier. You may need to click the **Show All Files** button if it's not visible.



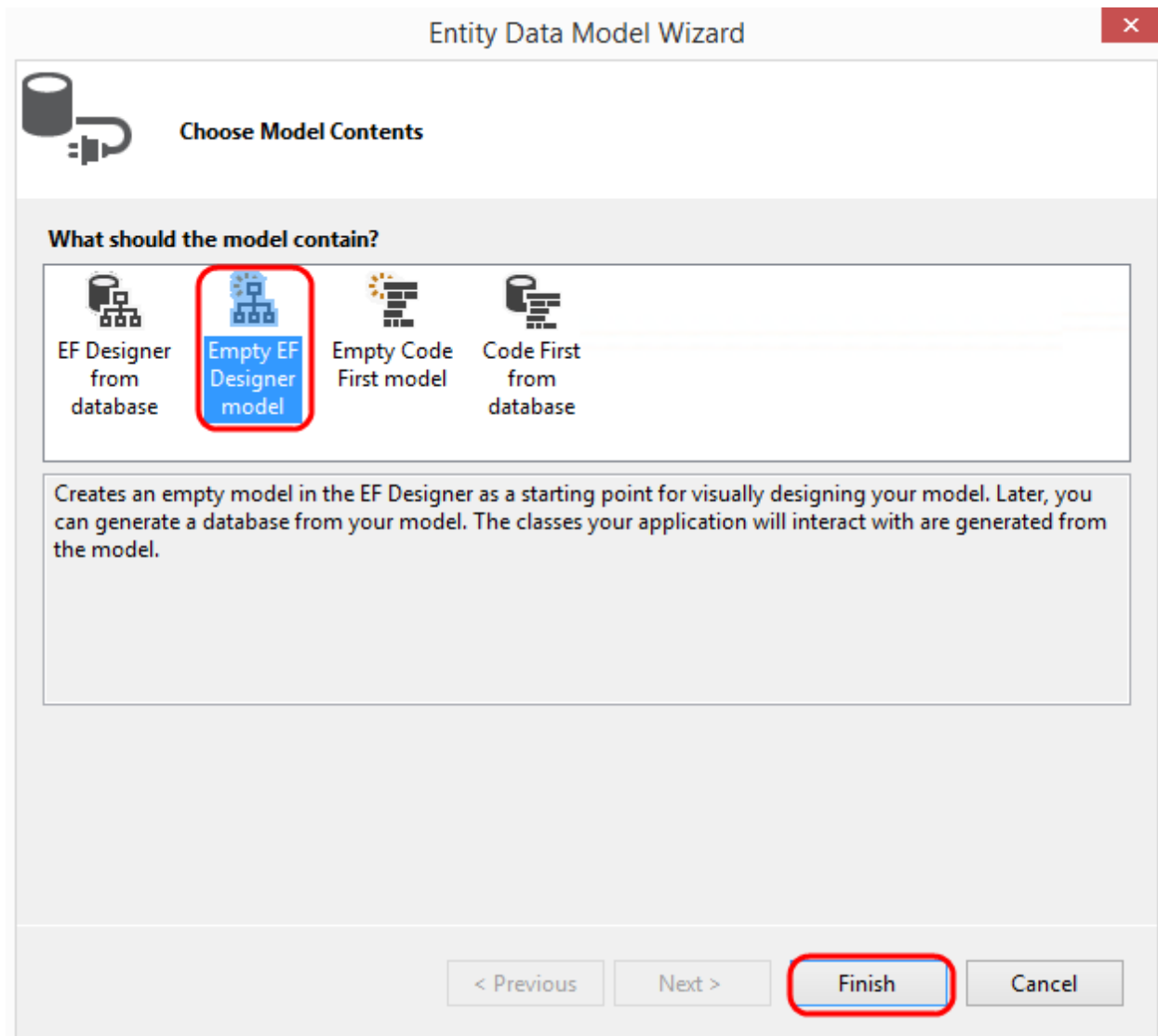
4. Right-click the **Models** folder and select **Add | New Item...**



5. In the **Add New Item** dialog, select the **Data** category and the **ADO.NET Entity Data Model** template. Type **"NorthwindModel"** as the **Name**, and click **Add** to open the **Entity Data Model Wizard**.

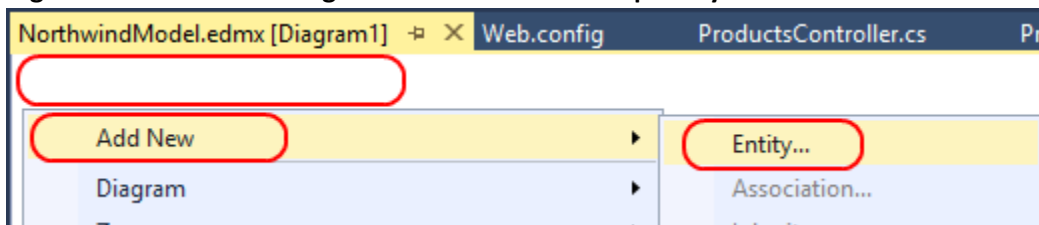


6. In the **Choose Model Contents** step, select **Empty EF Designer model** and click **Finish**.



You will now manually create the Northwind database model entities - **Product** and **Category** - in the **Entity Framework Designer**.

7. Right-click the **Model Designer** and select **Add New | Entity....**



8. Type "**Product**" in the **Entity name** field, and verify the value in the **Entity Set** field is set to **Products**. In the **Key Property** section, make sure that **Create key property** is checked, and the **Property name** is **Id**. Click **OK** to create.

Add Entity

Properties

Entity name: Product

Base type: (None)

Entity Set: Products

Key Property

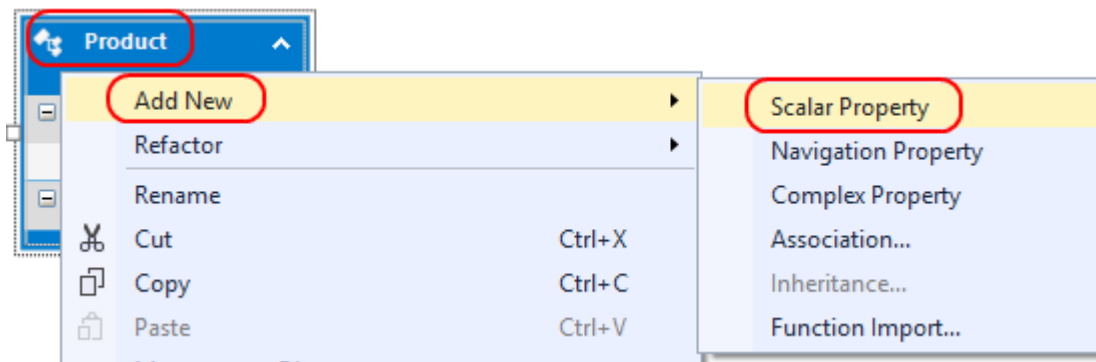
☒ Create key property

Property name: Id

Property type: Int32

OK Cancel

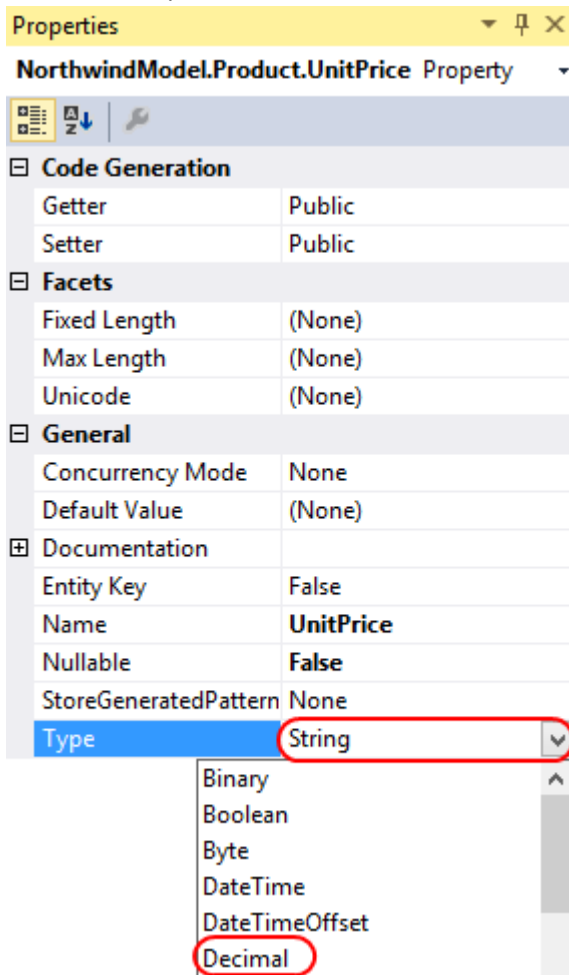
9. Right-click the **Product** entity and select **Add New | Scalar Property**.



Scalar properties are properties of an entity that map to a single field in the storage model.

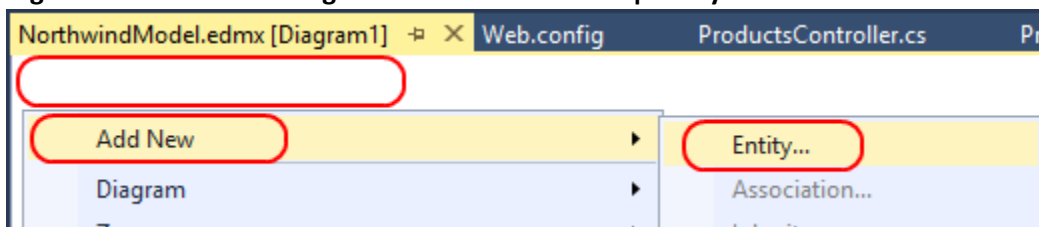
10. Type **"ProductName"** as the property name and press **Enter**.
11. Right-click the **Product** entity again, and again select **Add New | Scalar Property**.
12. Type **"UnitPrice"** as the property name and press **Enter**.

13. Notice that the default property type is **String**. Change the **Type** to **Decimal**. If the **Properties Panel** is not visible, press **F4** to show it.

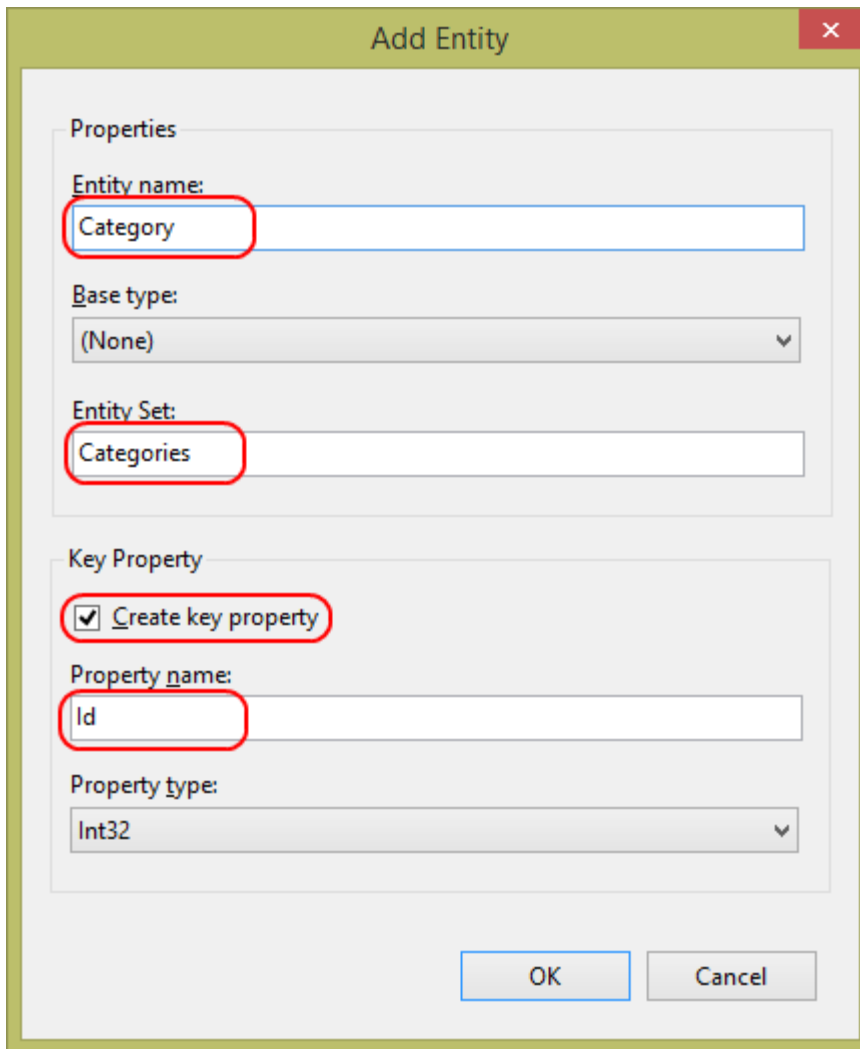


Another important property to notice is **Nullable**. In this case, the **Unit Price** is mandatory because the **Nullable** property is **false**.

14. Right-click the **Model Designer** and select **Add New | Entity....**



15. Type "**Category**" in the **Entity name** field, and verify the value in the **Entity Set** field is set to **Categories**. In the **Key Property** section, make sure that **Create key property** is checked, and the **Property name** is **Id**. Click **OK** to create.

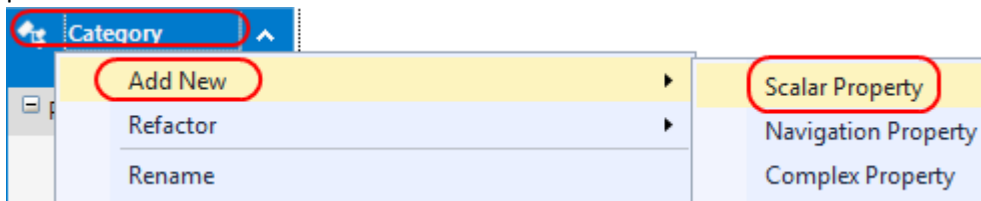


The 'Add Entity' dialog box is shown with the following fields and options:

- Entity name:** Category
- Base type:** (None)
- Entity Set:** Categories
- Key Property:**
 - ☒ Create key property
 - Property name:** Id
 - Property type:** Int32

Buttons: OK, Cancel

16. Right-click the **Category** entity, and select **Add New | Scalar Property**. Type “**CategoryName**”, and press **Enter**.



Now you will add an association between the **Category** and **Product** entities. In Entity Framework, an entity can be related to other entities through an association. Each relationship contains two ends that describe the entity type and the multiplicity of the type (one, zero-or-one, or many).

17. Right-click the **Category** entity, and select **Add New | Association**.



Note: You can alternatively right-click the **Product** entity, but remember to change the multiplicity. The **Entity Framework Designer** assumes that the entity that originates the new association has the multiplicity of **1**.

18. Make sure that the option **Add foreign key properties to the 'Product' Entity** is checked and click **OK**.

Add Association

Association Name:
CategoryProduct

End Entity: Category Multiplicity: 1 (One) ☒ Navigation Property: Products

End Entity: Product Multiplicity: * (Many) ☒ Navigation Property: Category

☒ Add foreign key properties to the 'Product' Entity

Category can have * (Many) instances of Product. Use Category.Products to access the Product instances.

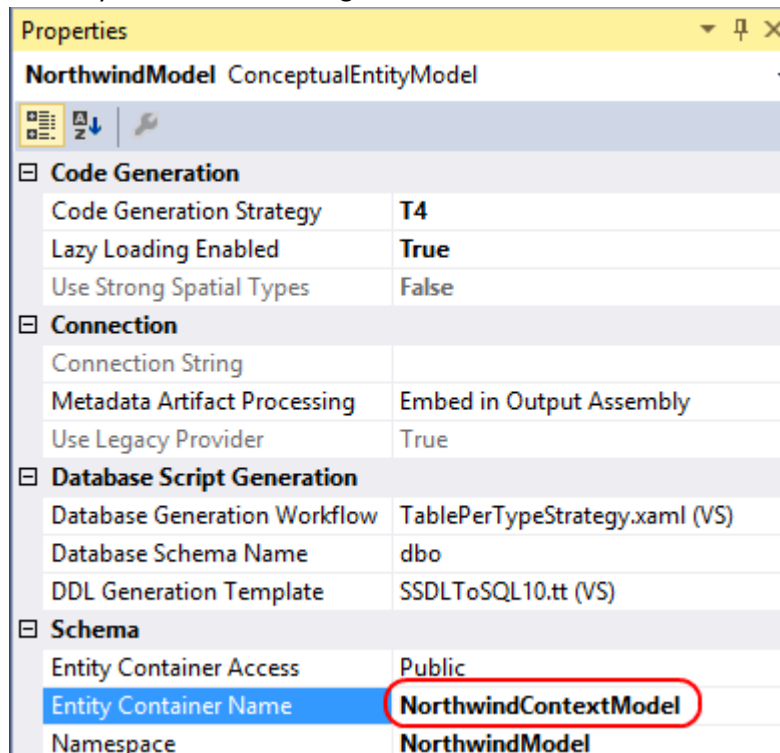
Product can have 1 (One) instance of Category. Use Product.Category to access the Category instance.

OK Cancel

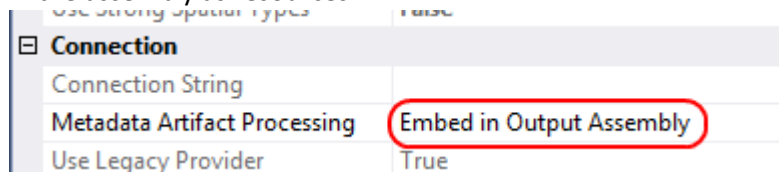
Note: The designer assigns singular names for the navigation properties by default. As one category can be related to many products, in this step you are also changing the navigation property of **Category** to the plural **Products**.

19. Click on any blank part of the **Model Designer**.

20. In the **Properties** panel, change the **Entity Container Name** to “**NorthwindContextModel**”. This will enable you to continue using the same views and controllers from the previous exercise.



21. Also, make sure that the **Metadata Artifact Processing** property is set to **Embed in Output Assembly**. This way the CSDL, MSL, and SSDL files are extracted from the EDMX file and embedded in the assembly as resources.



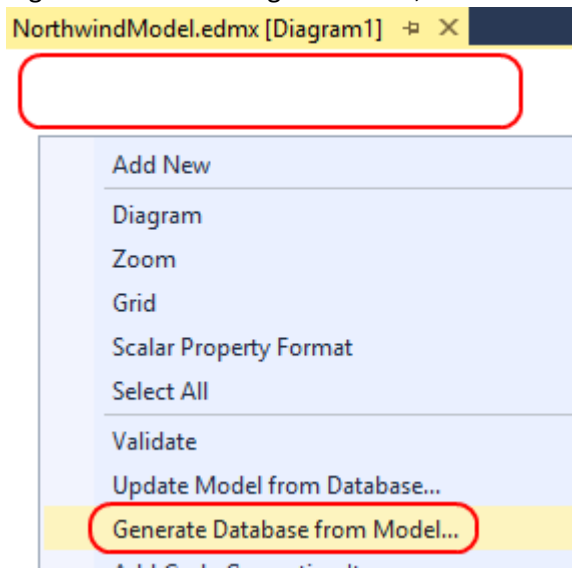
22. Press **Ctrl+S** to save the model.

Task 2 – Generating the Database

In this task, you will generate and execute the SQL script that creates the database and the tables.


In the Model First approach, Entity Framework generates a SQL script, rather than executing or directly modifying the database. Therefore, if you are using a data source that targets an empty database, you have to run the SQL script that creates the database tables manually.

1. Right-click on the designer surface, select **Generate Database from Model...**



2. In the **Generate Database Wizard**, click **New Connection**.

Generate Database 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.

Entity connection string:

☒ Save entity connection settings in Web.Config as:

< Previous Next > Finish Cancel

3. If the **Choose Data Source** dialog appears, select **Microsoft SQL Server Database File** and click **Continue**. Otherwise, click **Change...**, select **Microsoft SQL Server Database File** and click **OK**.

Connection Properties

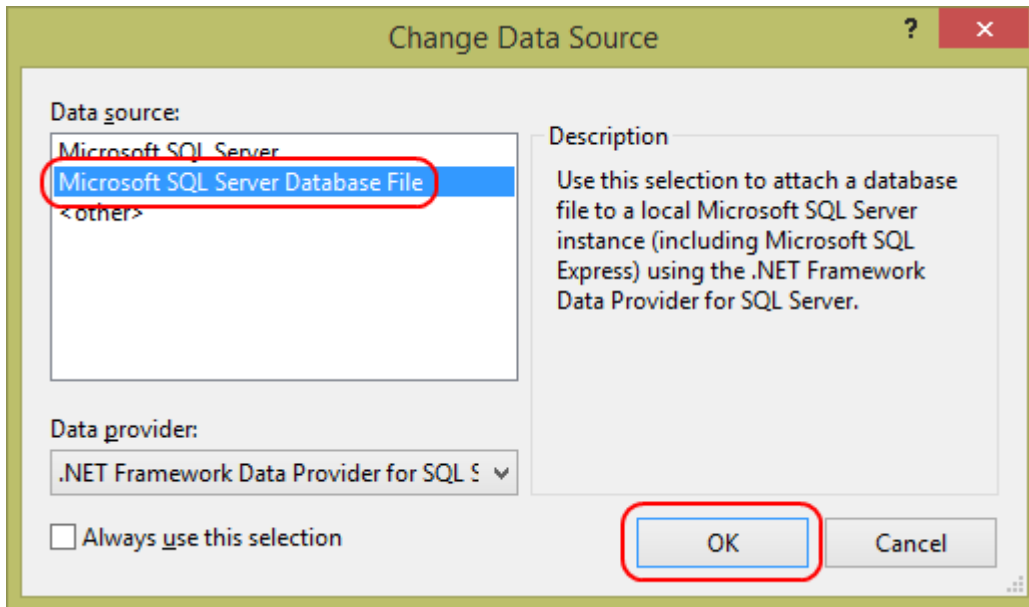
Enter information to connect to the selected data source or click "Change" to choose a different data source and/or provider.

Data source:

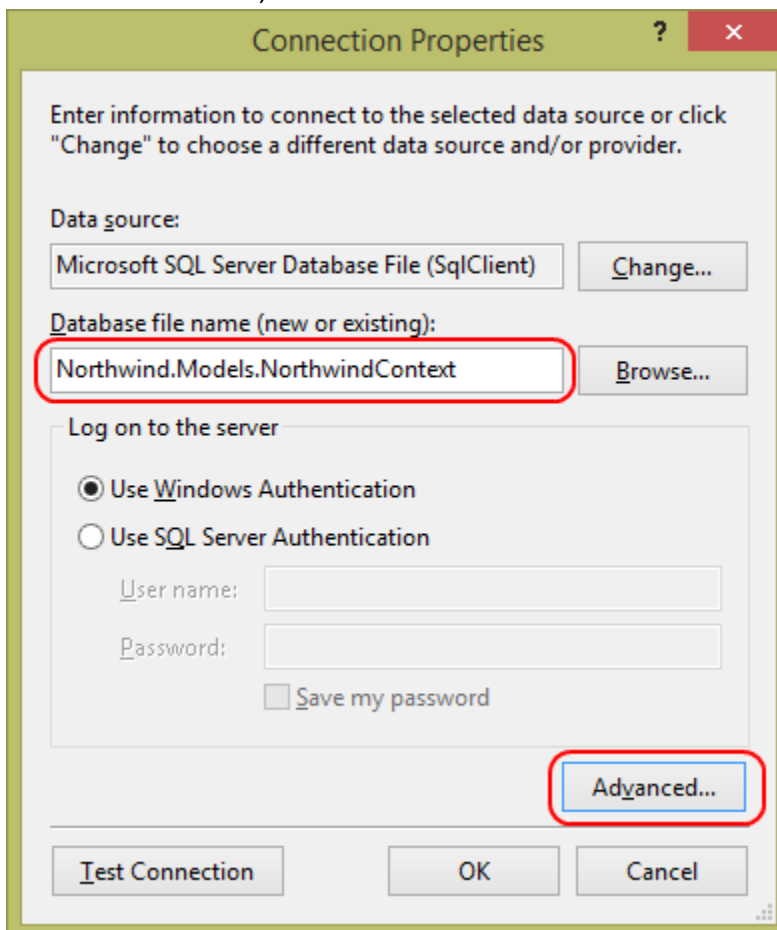
Microsoft SQL Server (SqlClient) **Change...**

Server name:

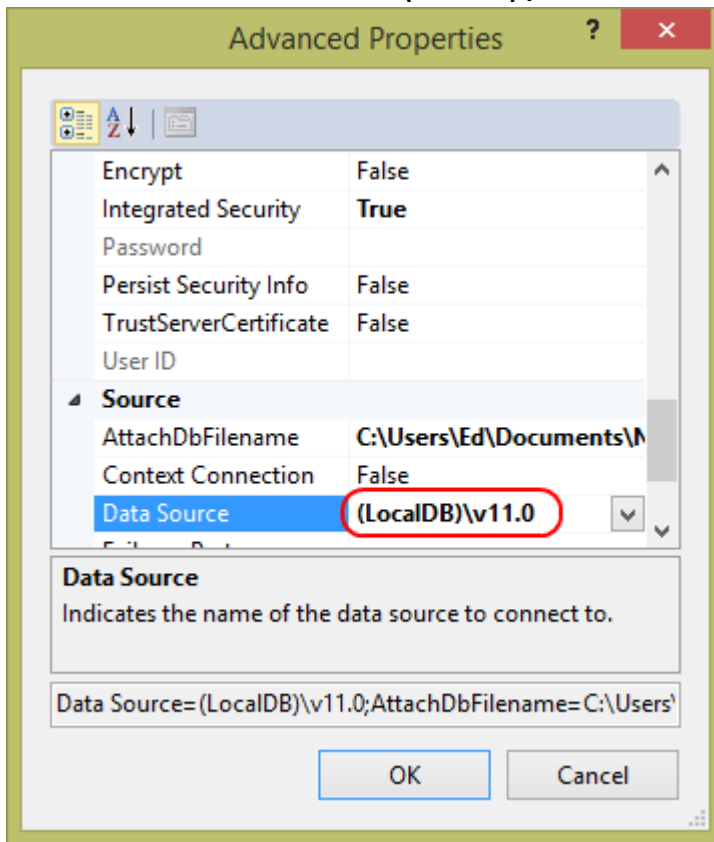
Refresh



4. In the **Connection Properties** window, type “**Northwind.Models.NorthwindContext**” as the database name. Next, click **Advanced**.



5. Ensure that **Data Source** is set to **(LocalDB)\v11.0** and click **OK**.



6. In the **Connection Properties** dialog, click **OK**. When prompted to create the database, click **Yes**.

Connection Properties ? X

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):
Northwind.Models.NorthwindContext.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

7. In the **Generate Database Wizard** window, wait until the data connection updates. Make sure that the property **Save entity connection settings in Web.Config** is checked. Click **Next** to continue.

Generate Database Wizard

 Choose Your Data Connection

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

Northwind.Models.NorthwindContext.mdf 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.

Entity connection string:

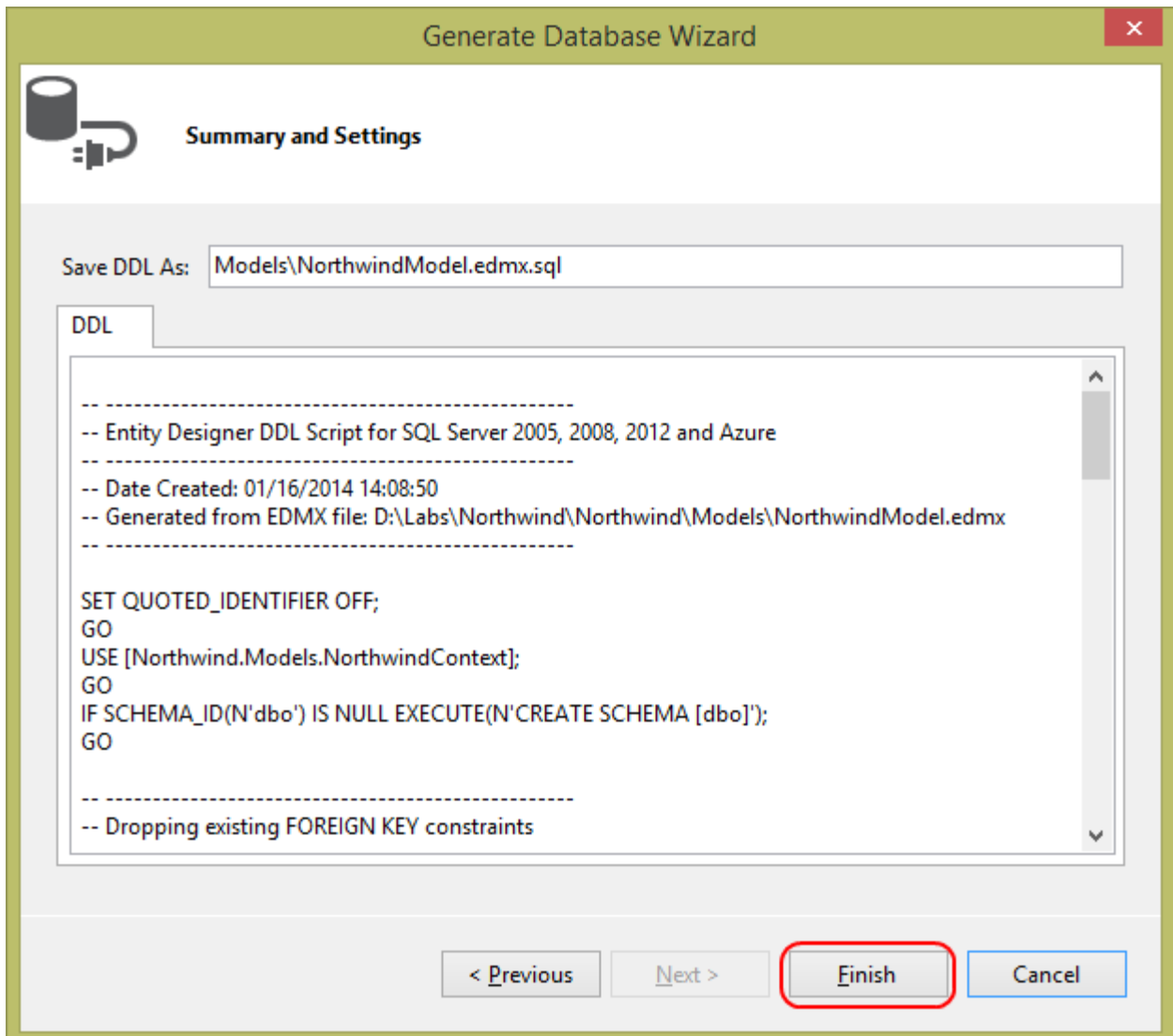
```
metadata=res://*/Models.NorthwindModel.csdl|res://*/Models.NorthwindModel.ssdl|
res://*/Models.NorthwindModel.msl;provider=System.Data.SqlClient;provider connection string="data
source=(LocalDB)\v11.0;attachdbfilename=C:\Users\Ed\Documents
\Northwind.Models.NorthwindContext.mdf;integrated security=True;connect
timeout=30;MultipleActiveResultSets=True;App=EntityFramework"
```

☒ Save entity connection settings in Web.Config as:

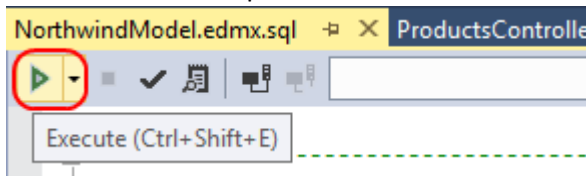
NorthwindContextModel

< Previous **Next >** Finish Cancel

8. In the **Summary and Settings** step, click **Finish**.



- When the **NorthwindModel.edmx.sql** file opens, click the **Execute** button or press **Ctrl+Shift+E** to execute the SQL script that creates the database tables.



- In the **Connect to Server** dialog, set the **Server name** to **"(localdb)\v11.0"** and click **Options**.

Connect to Server

Microsoft SQL Server 2012

Server type: Database Engine

Server name: (localdb)\v11.0

Authentication: Windows Authentication

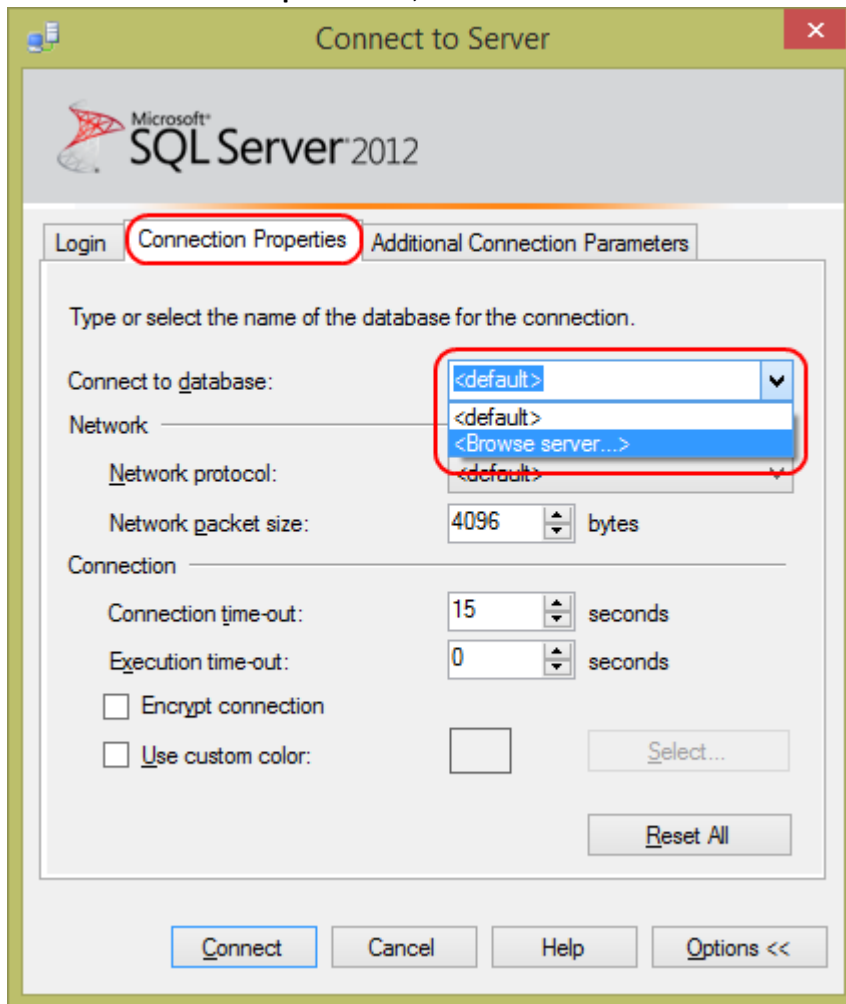
User name: ED\Ed

Password:

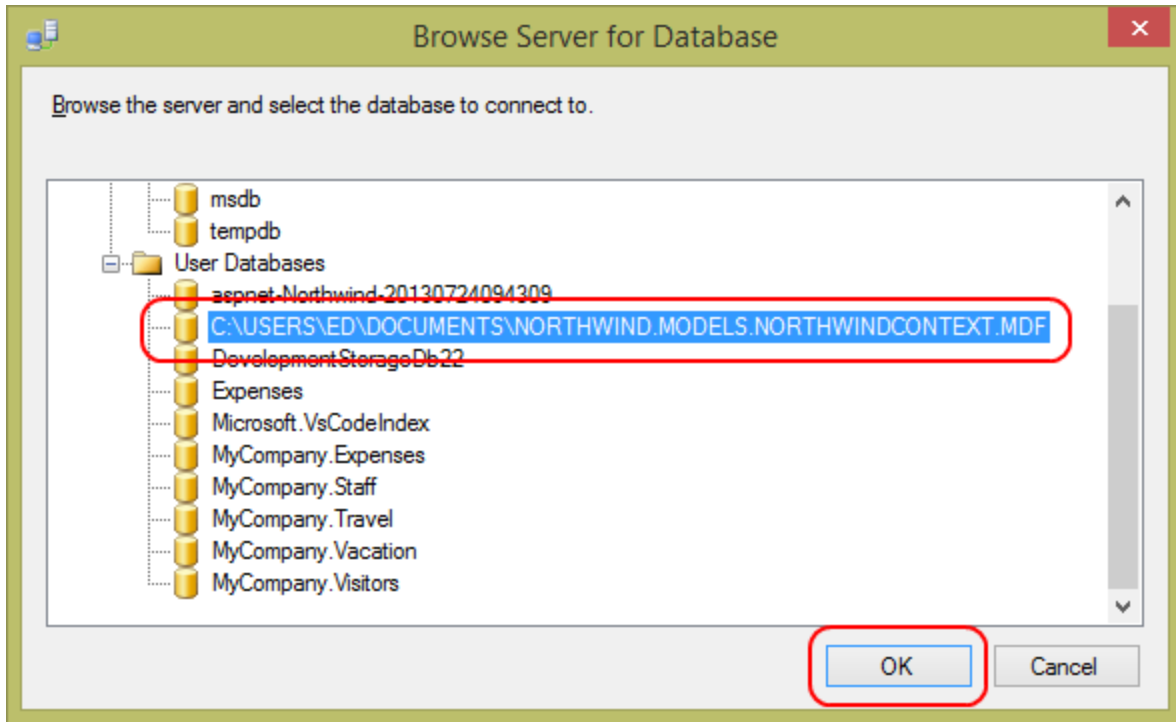
☐ Remember password

Connect Cancel Help Options >>

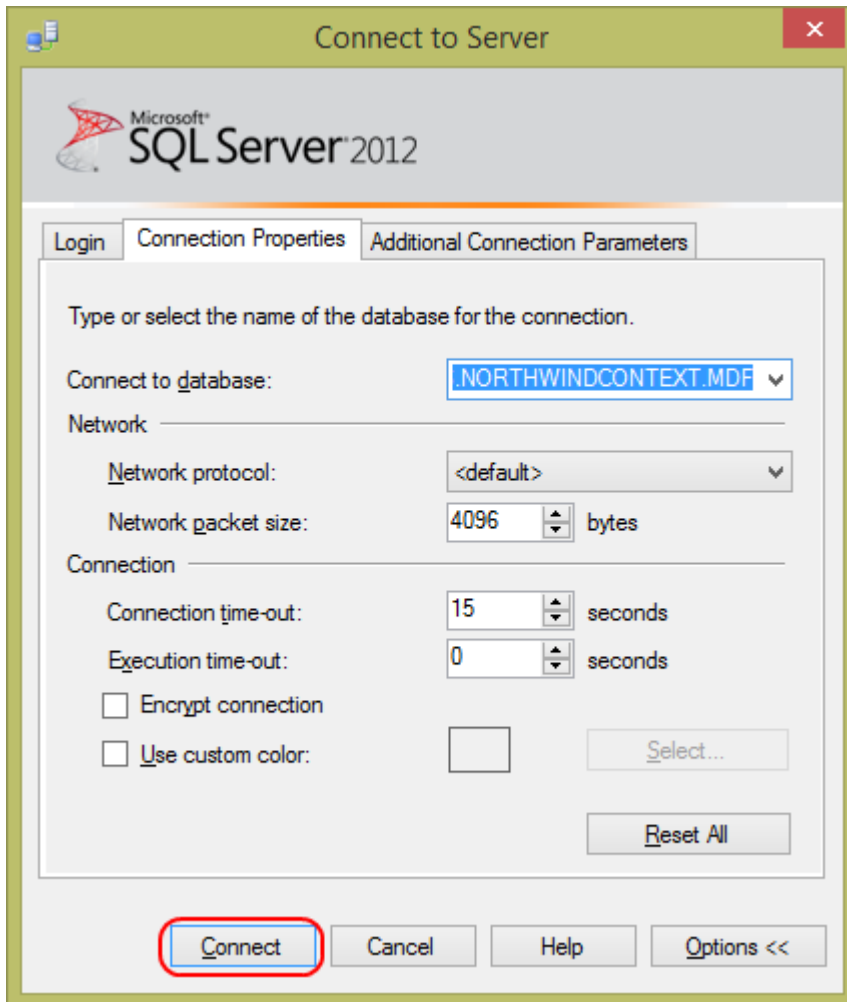
11. On the **Connection Properties** tab, select **<Browse server...>** in the **Connect to database** option.



12. When asked about connecting to the server, click **Yes**.
13. Under **User Databases**, select the database file that you created in the **Connection Wizard** ([your Documents directory path]\Northwind.Models.NorthwindContext.mdf). Click **OK**.



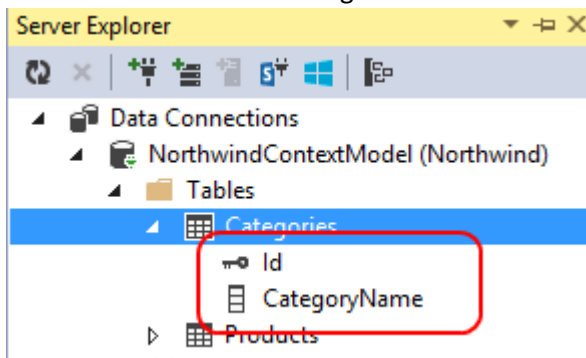
14. Click **Connect** to complete the script execution.



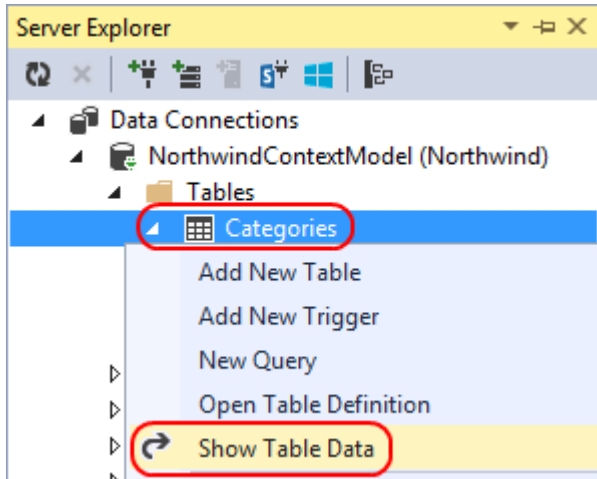
15. To confirm the database, open **Windows Explorer** and navigate to the location it was created at. By default, it should be in the root of your **Documents** folder (C:\Users\{user name}\Documents).
16. Select **Build | Build Solution** from the main menu to build the solution.

Task 3 – Verification

1. In **Server Explorer**, expand **NorthwindContextModel | Tables | Categories** to confirm it has the two fields we defined in the designer.



2. Right-click the **Categories** table and select **Show Table Data**.

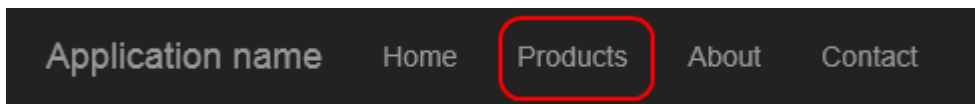


3. The table should be empty since we just created it. Type **"Confections"** as the first **CategoryName** and press **Enter**. The database will assign in an ID. Optionally, you can add more categories using the same process.

The screenshot shows the 'dbo.Categories [Data]' table. The table has two columns: 'Id' and 'CategoryName'. The first row has '1' in the 'Id' column and 'Confections' in the 'CategoryName' column. The 'Confections' value is highlighted with a red circle. Below the first row, there is a row with 'NULL' in both columns, preceded by an asterisk (*).

| | Id | CategoryName |
|---|------|--------------|
| ▶ | 1 | Confections |
| * | NULL | NULL |

6. Press **F5** to run the application.
7. Click **Products**.



8. In the products page, click **Create New**.

Index

Create New

| Name | Price | Category |
|------|-------|----------|
|------|-------|----------|

9. In the **Create** view, type **"Cupcakes"** as the **Name**, select **"Confections"** as the **CategoryId**, and type **"4"** as the **Price**. Click **Create** to create the record, which will also return you to the products list.

Create

Product

| | |
|---------------------------------------|--|
| Name | <input type="text" value="Cupcakes"/> |
| CategoryId | <input type="text" value="Confections"/> |
| Price | <input type="text" value="4"/> |
| <input type="button" value="Create"/> | |

[Back to List](#)

11. Close the browser to end the debugging session.

Exercise 4: WCF Data Services and OData

WCF Data Services (formerly known as "**ADO.NET Data Services**") is a component of the .NET Framework that enables you to create services that use the **Open Data Protocol (OData)** to expose and consume data on the Web. OData is a web protocol for querying and updating data that enables you to expose your data as resources that are addressable by URIs.

In this exercise, you will learn how to create an OData-based service by using an Entity Framework data model.

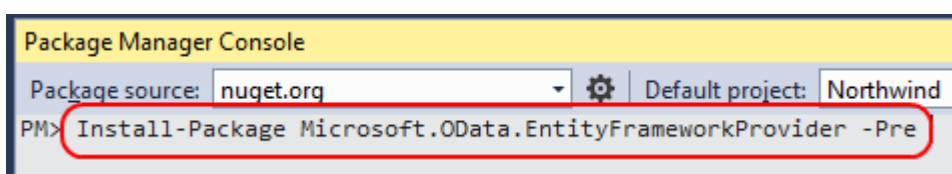
Task 1 – Creating the WCF Data Service

In this task, you will create a data service that uses WCF Data Services to expose an Open Data Protocol (OData) feed that is based on the Northwind database you have using throughout the lab.

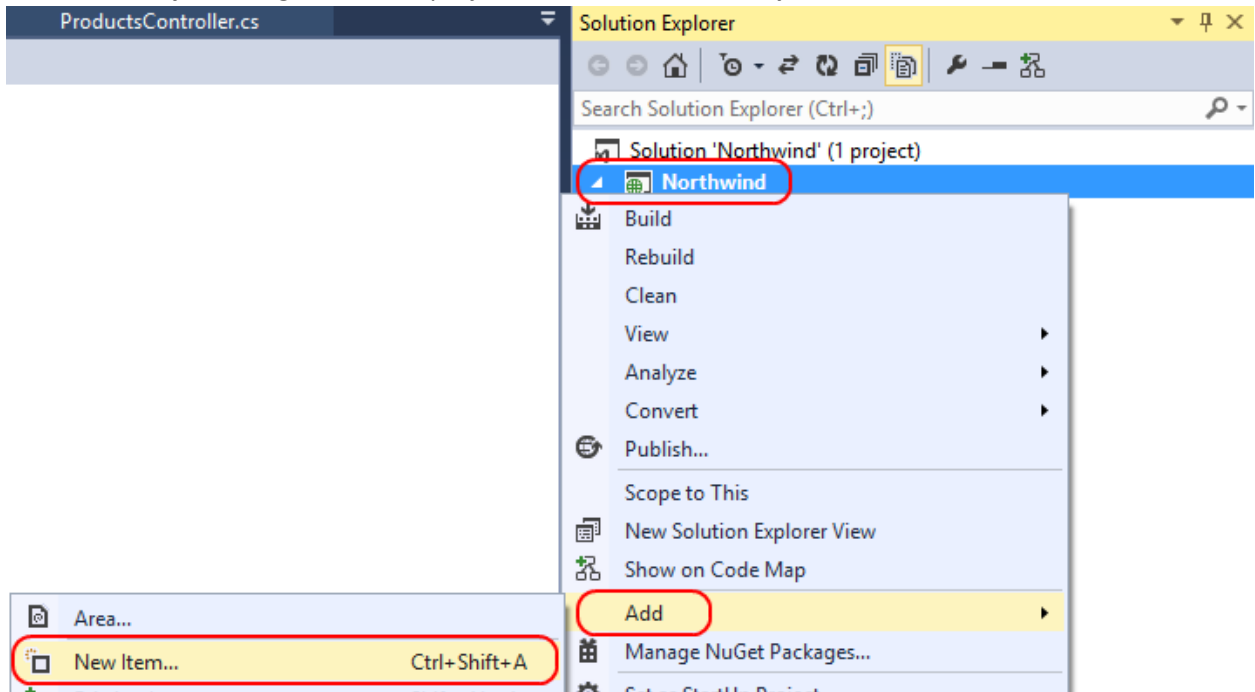
1. Navigate to the **Northwind Solution** folder provided with this lab and open **Northwind.sln**. This project is what you ended exercise 1 with.
2. From the main menu, select **Tools | NuGet Package Manager | Package Manager Console**.
3. Paste the following line to manually install a preview version of the **Entity Framework Provider for OData** and press **Enter**.

text

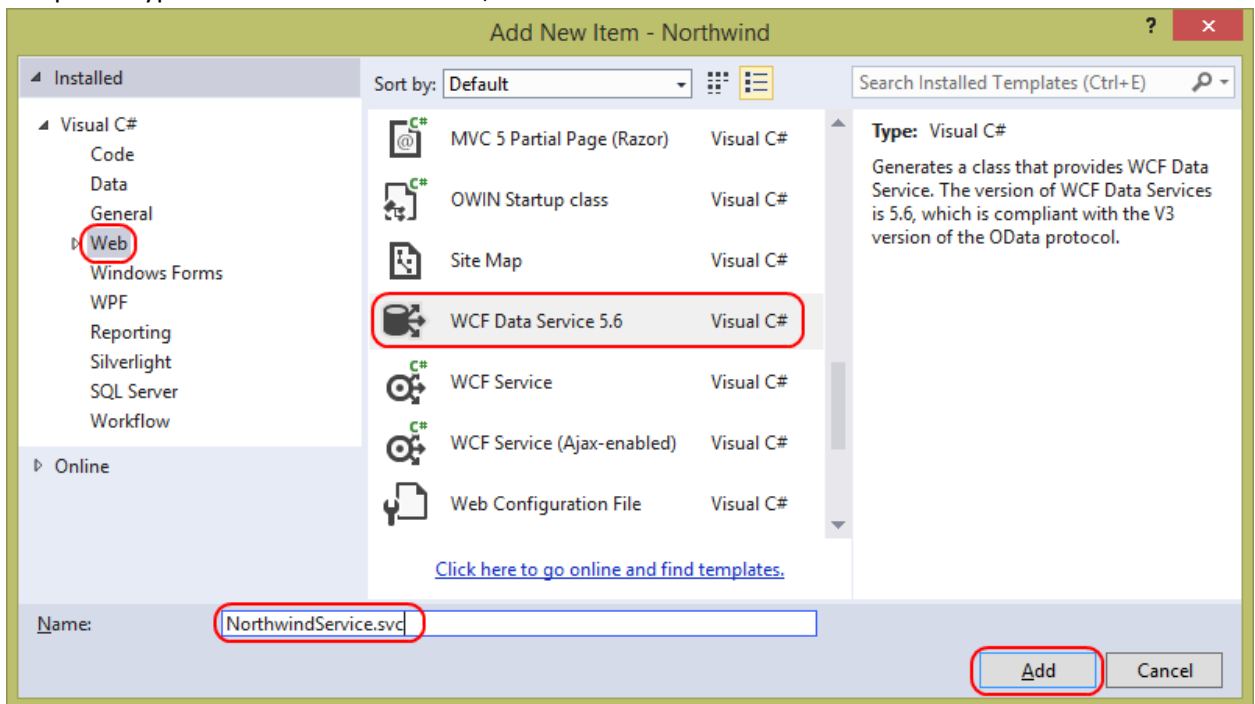
```
Install-Package Microsoft.OData.EntityFrameworkProvider -Pre
```



4. In **Solution Explorer**, right-click the project node and select **Add | New Item....**



5. In the **Add New Item** dialog, select the **Web** category from the left pane and the **WCF Data Service** template. Type "**NorthwindService.svc**", and then click **Add**.



6. Add the following using statement to the **NorthwindService.cs** file.

C#

```
using Northwind.Models;
using System.Data.Services.Providers;
```

- Update the class to derive from **EntityFrameworkDataService<NorthwindContext>**.

C#

```
public class NorthwindService : EntityFrameworkDataService<NorthwindContext>
```

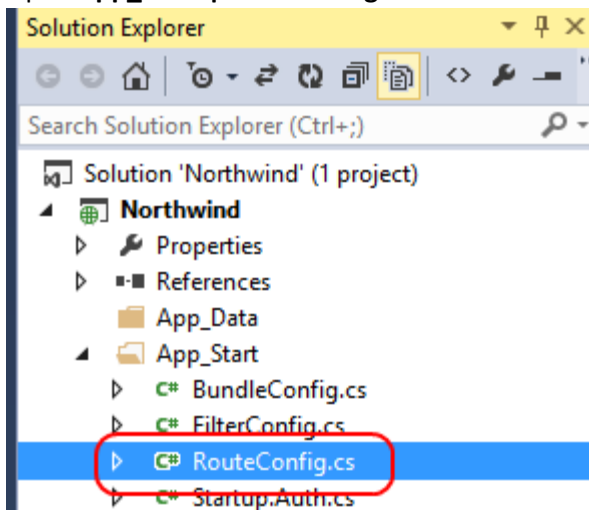
- Grant read and write access for **Products** and only read for **Categories** by adding the following code to the beginning of the **InitializeService** method.

C#

```
config.SetEntitySetAccessRule("Products", EntitySetRights.All);
config.SetEntitySetAccessRule("Categories", EntitySetRights.AllRead);
```

Note: By default, a data service does not expose any resources. Access to resources needs to be explicitly enabled before any resources or associations can be consumed.

- Since this is an ASP.NET MVC project, you need to override routing to enable access to the service. Open **App_Start | RouteConfig.cs**.

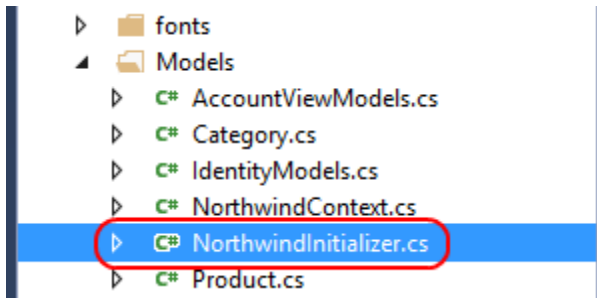


- Add an **IgnoreRoute** entry for **NorthwindService.svc** and any sub-paths by inserting the following code at the beginning of the **RegisterRoutes** method.

C#

```
routes.IgnoreRoute("NorthwindService.svc/{*pathInfo}");
```

- Finally, you need to update **NorthwindInitializer** to seed a few products. Open the **NorthwindInitializer.cs** class located in the **Models** folder.



12. Replace the **AddCategory** method with the code below to return an instance of **Category** after it is created.

```
C#
private Category AddCategory(NorthwindContext context, string categoryName)
{
    var category = context.Categories.Add(
        new Category()
        {
            CategoryName = categoryName
        });

    return category;
}
```

13. Implement an **AddBeverage** method to add products to the data context.

```
C#
private void AddBeverage (NorthwindContext context, string productName, int
categoryId, decimal unitPrice)
{
    context.Products.Add(
        new Product()
        {
            ProductName = productName,
            CategoryId = categoryId,
            UnitPrice = unitPrice
        });
}
```

14. Replace the existing **Seed** method with the code below. It will retrieve the **Beverage** category after it is created and pre-populate the **Products** set with some sample data. Notice the call to **context.SaveChanges()**. This persists the changes to the underlying database, which is required before a created **Category's Id** property can be retrieved.

```
C#
protected override void Seed(NorthwindContext context)
{
    var beverages = this.AddCategory(context, "Beverages");
    this.AddCategory(context, "Condiments");
}
```

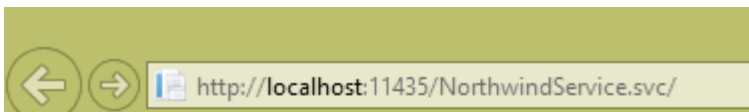
```

        this.AddCategory(context, "Confections");
        this.AddCategory(context, "Dairy Products");
        this.AddCategory(context, "Grains/Cereals");
        this.AddCategory(context, "Meat/Poultry");
        this.AddCategory(context, "Produce");
        this.AddCategory(context, "Seafood");
        context.SaveChanges();

        this.AddBeverage(context, "Red Wine", beverages.Id, 10);
        this.AddBeverage(context, "White Wine", beverages.Id, 10);
        this.AddBeverage(context, "Diet Coke", beverages.Id, 1);
        context.SaveChanges();
    }

```

15. Press **F5** to run the web site and go to **/NorthwindService.svc**. Notice that there are two collections available.



```

<?xml version="1.0" encoding="UTF-8"?>
- <service xmlns:atom="http://www.w3.org/2005/
  - <workspace>
    <atom:title>Default</atom:title>
    - <collection href="Categories">
      <atom:title>Categories</atom:title>
    </collection>
    - <collection href="Products">
      <atom:title>Products</atom:title>
    </collection>
  </workspace>
</service>

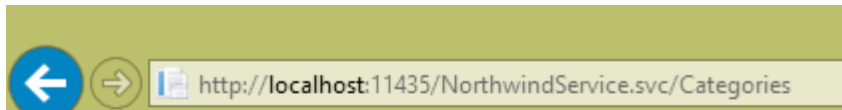
```

Note: The XML document returned by default is an Atom service document because the default serialization used by the data service is Atom. If the browser displays a message saying it cannot display the feed, turn off the feed reading view. In **Internet Explorer**, navigate to **Tools | Internet Options | Content Tab | Feeds Section | Settings** and clear the **Turn on feed reading view** check box. Then close and reopen IE.

Task 2 – Querying the OData Service

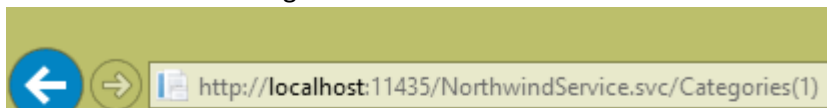
In OData, you address any data exposed by the data model by using a URI. OData defines a set of system query options that you can use to perform traditional query operations against resources, such as filtering, sorting, and paging. In this task, you will access the OData service using a web browser and you will query the service resources by using OData query options.

1. Retrieve all the categories. To do this, go to **/NorthwindService.svc/Categories**. The following feed should appear.



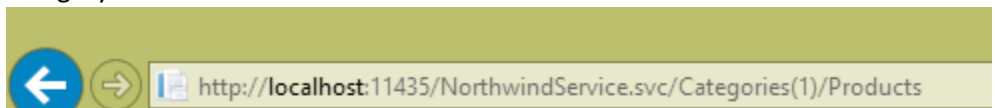
```
<?xml version="1.0" encoding="utf-8" ?>
- <feed xml:base="http://localhost:11435/NorthwindService.svc/" xmlns
  <id>http://localhost:11435/NorthwindService.svc/Categories</id>
  <title type="text">Categories</title>
  <updated>2014-01-17T19:20:10Z</updated>
  <link rel="self" title="Categories" href="Categories" />
- <entry>
  <id>http://localhost:11435/NorthwindService.svc/Categories(1)<
  <category term="Northwind.Models.Category" scheme="http://schen
  <link rel="edit" title="Category" href="Categories(1)" />
  <link rel="http://schemas.microsoft.com/ado/2007/08/dataservic
  <title />
  <updated>2014-01-17T19:20:10Z</updated>
- <author>
```

2. Retrieve the details of category 1. To do this, go to `/NorthwindService.svc/Categories(1)`. You should see the following feed.




```
<?xml version="1.0" encoding="utf-8" ?>
- <entry xml:base="http://localhost:11435/NorthwindService.svc/" xml
  <id>http://localhost:11435/NorthwindService.svc/Categories(1)<
  <category term="Northwind.Models.Category" scheme="http://schen
  <link rel="edit" title="Category" href="Categories(1)" />
  <link rel="http://schemas.microsoft.com/ado/2007/08/dataservic
  <title />
  <updated>2014-01-17T19:20:47Z</updated>
- <author>
  <name />
</author>
```

3. Go to `/NorthwindService.svc/Categories(1)/Products`. You should see all the products within the category with ID 1.



```
<?xml version="1.0" encoding="utf-8" ?>
- <feed xml:base="http://localhost:11435/NorthwindService.svc/" xmlns="http://ww
  <id>http://localhost:11435/NorthwindService.svc/Categories%281%29/Product
  <title type="text">Products</title>
  <updated>2014-01-17T19:34:18Z</updated>
  <link rel="self" title="Products" href="Products" />
- <entry m:etag="W/"1"">
  <id>http://localhost:11435/NorthwindService.svc/Products(1)</id>
  <category term="Northwind.Models.Product" scheme="http://schemas.microsoft.
  <link rel="edit" title="Product" href="Products(1)" />
  <link rel="http://schemas.microsoft.com/ado/2007/08/dataservices/related/C
  <title />
```

4. Go to `/NorthwindService.svc/Products?$filter=UnitPrice ge 10`. You should see the details of the products with unit price greater or equal than 10.



```
<?xml version="1.0" encoding="utf-8" ?>
- <feed xml:base="http://localhost:11435/NorthwindService.svc/" xmlns="http://www.w3.org/2005/
  <id>http://localhost:11435/NorthwindService.svc/Products</id>
  <title type="text">Products</title>
  <updated>2014-01-17T19:34:49Z</updated>
  <link rel="self" title="Products" href="Products" />
- <entry m:etag="W/"1"">
  <id>http://localhost:11435/NorthwindService.svc/Products(1)</id>
  <category term="Northwind.Models.Product" scheme="http://schemas.microsoft.com/ado/2007/1
  <link rel="edit" title="Product" href="Products(1)" />
  <link rel="http://schemas.microsoft.com/ado/2007/08/dataservices/related/Category" type="a
  <title />
  <updated>2014-01-17T19:34:49Z</updated>
- <author>
  <name />
</author>
- <content type="application/xml">
  - <m:properties>
    <d:Id m:type="Edm.Int32">1</d:Id>
```

5. Close the browser.

Note: To learn more about how to perform queries using OData URI conventions go to:

<http://www.odata.org/developers/protocols/uri-conventions>.

Exercise 5: Using Code First Fluent API

Code First enables you to describe a model by using C# or Visual Basic .NET classes. The basic shape of the model is detected by using conventions. Conventions are sets of rules that automatically configure a conceptual model based on class definitions when working with Code First. After your classes are defined you can optionally specify additional mapping configuration that will override the default Code First conventions.

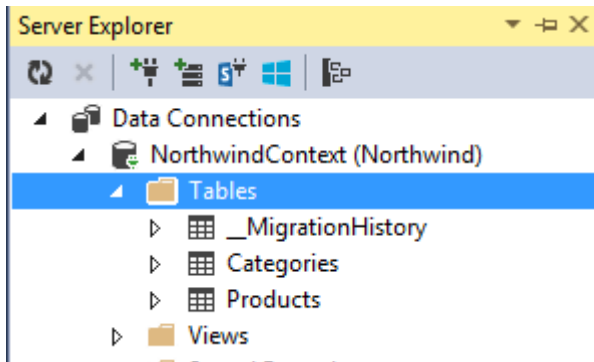
Code First gives you two ways to add these configurations to your classes. One is using simple attributes called Data Annotations (see Exercise 1) and the other is using Code First's fluent API, which provides you with a way to describe configurations imperatively in code.

This exercise will show you how to use the Fluent API to override default Entity Framework conventions affecting the shape of a generated schema.

Task 1 – Exploring the Conventions Used by Entity Framework

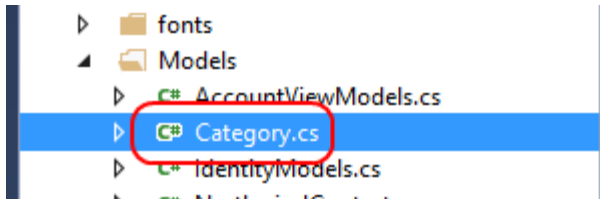
In this task, you will learn some conventions used by Entity Framework by exploring the schema from the database used in the previous exercise, which is the same generated in Exercise 1.

1. In **Server Explorer**, expand **NorthwindContext | Tables**.

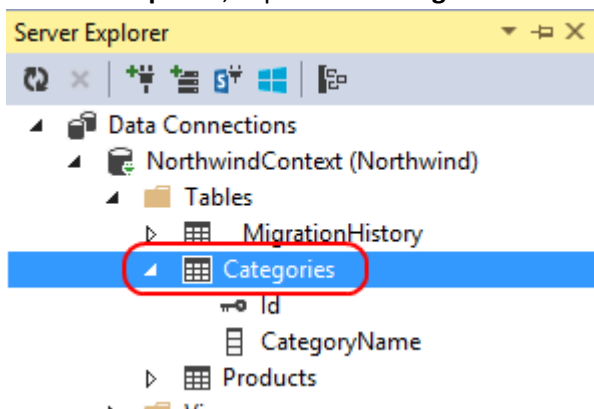


Notice the table names match the entities you are using in your model but they are pluralized. This is the first convention you will notice Entity Framework uses.

2. Open **Model | Category.cs**.



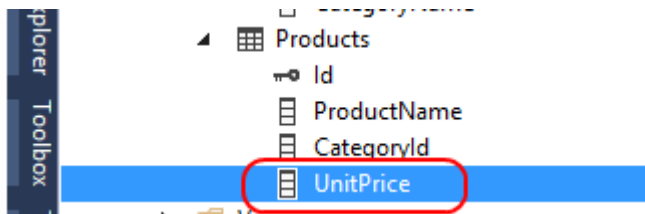
3. In **Server Explorer**, expand the **Categories** table.



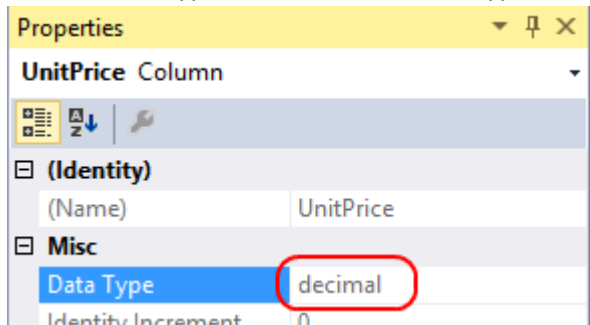
4. Notice the **Id** property in the **Category** class does not have any metadata nor attributes indicating that it is a primary key.

Code First infers that a property is a primary key if a property in a class contains the "ID" or "Id" string, or the class name is followed by Id (Id can be any combination of uppercase and lowercase). The primary key properties are mapped to identity columns in the database. By convention, the integer key properties are configured with **DatabaseGeneratedOption.Identity** mapping when working with Code First (if you check the **Id** column properties you will notice the **AutoIncrement** property is set to **True**).

5. Expand the **Products** table and select the **UnitPrice** column.



- If you take a look at the **Properties** panel, you'll see that the **DataType** is set to **decimal**. Entity Framework uses the data type that better fits the type used in the code, and in this case, it uses the decimal data type for the decimal .NET type.

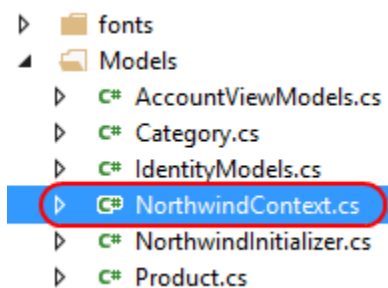


Task 2 – Disabling a Convention

The basic shape of the model is detected by using conventions. Conventions are sets of rules that are used to automatically configure a conceptual model based on class definitions when working with Code First. The conventions are defined in the **System.Data.Entity.ModelConfiguration.Conventions** namespace.

In this task, you will learn how to use the **Fluent API** (exposed through the **DbModelBuilder** object) to disable the convention used by Entity Framework to pluralize the table names.

- Open the **NorthwindContext.cs** file.



- Add the following using statement at the top.

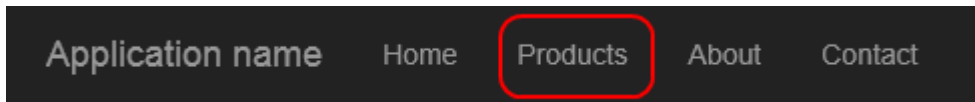
```
C#
using System.Data.Entity.ModelConfiguration.Conventions;
```

- In the **NorthwindContext** class, override the **OnModelCreating** method by adding the following code that removes the pluralizing convention.

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

    base.OnModelCreating(modelBuilder);
}
```

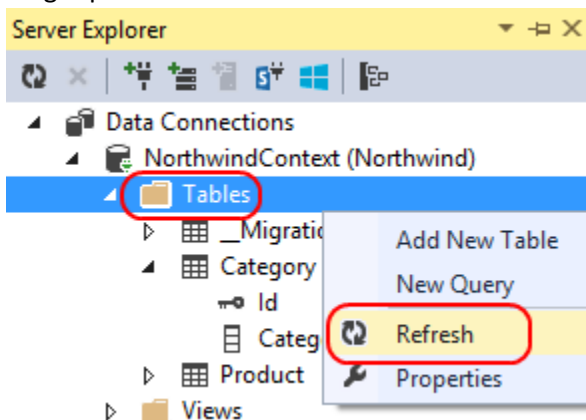
4. Press **F5** to run the application.
5. Click the **Products** link in the menu bar.



The Products index view will use Entity Framework to list the products, since Entity Framework is configured to use the **DropCreateDatabaseIfModelChanges** strategy and you have changed the model using the model builder (removed pluralizing), the database will be dropped and created again using the updated model.

Note: If you receive a `SqlException` during the DROP process, it is likely the result of having an open handle to the database, most likely from tools like the Server Explorer. To work around this, close all panes that have an open reference to the database and run the app again. If this fails, restart Visual Studio and try again.

6. Close the browser and go back to Visual Studio.
7. In **Server Explorer**, right-click the **Tables** folder and select **Refresh**. Note that the table names are no longer pluralized.



Note: To learn more about Entity Framework conventions go to <http://msdn.microsoft.com/en-us/library/hh161541.aspx>.

Task 3 – Customizing the Database Generation

You can further configure your model by using data annotations or the Fluent API. Precedence is given to configuration through the Fluent API followed by data annotations and then conventions.

The Fluent API is exposed through the **DbModelBuilder** type and is most commonly accessed by overriding the **OnModelCreating** method on **DbContext**.

In this task, you will learn how to use the Fluent API to customize how your model classes will map to the database schema.

1. In the **NorthwindContext** class, add the following code to the beginning of the **OnModelCreating** method to use the **CategoryCode** as the primary key of the Category entity.

C#

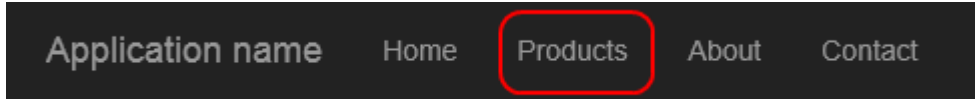
```
modelBuilder.Entity<Category>()  
    .Property(c => c.Id)  
    .HasColumnName("CategoryCode");
```

2. Add the following code to the beginning of the **OnModelCreating** method to change the **UnitPrice** column type from the **Product** entity to a more appropriate one, the **"money"** type.

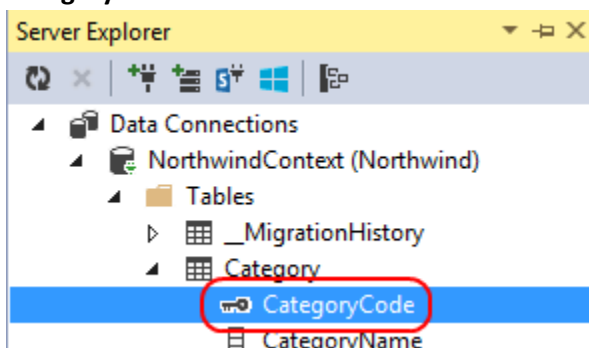
C#

```
modelBuilder.Entity<Product>()  
    .Property(p => p.UnitPrice)  
    .HasColumnType("money");
```

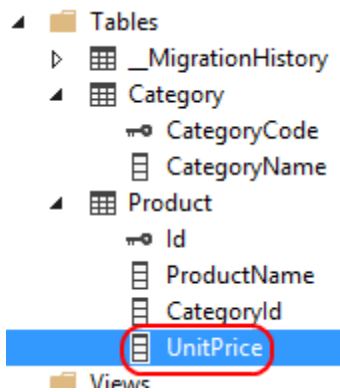
3. Press **F5** to run the application.
4. Click the **Products** link in the menu bar. This will regenerate the database.



5. In **Server Explorer**, expand the **Category** table and notice that the primary key column name is **CategoryCode** instead of **Id**.



6. Expand the **Product** table and select the of the **UnitPrice** column.



7. In the **Properties** pane you can see that the **DataType** property is using the **money** type.

| Properties | |
|--------------------|-----------|
| UnitPrice Column | |
| (Identity) | |
| (Name) | UnitPrice |
| Misc | |
| Data Type | money |
| Identity Increment | 0 |
| Identity Seed | 0 |
| Is Identity | False |
| Length | 8 |
| Nullable | False |
| Precision | 19 |
| Scale | 4 |