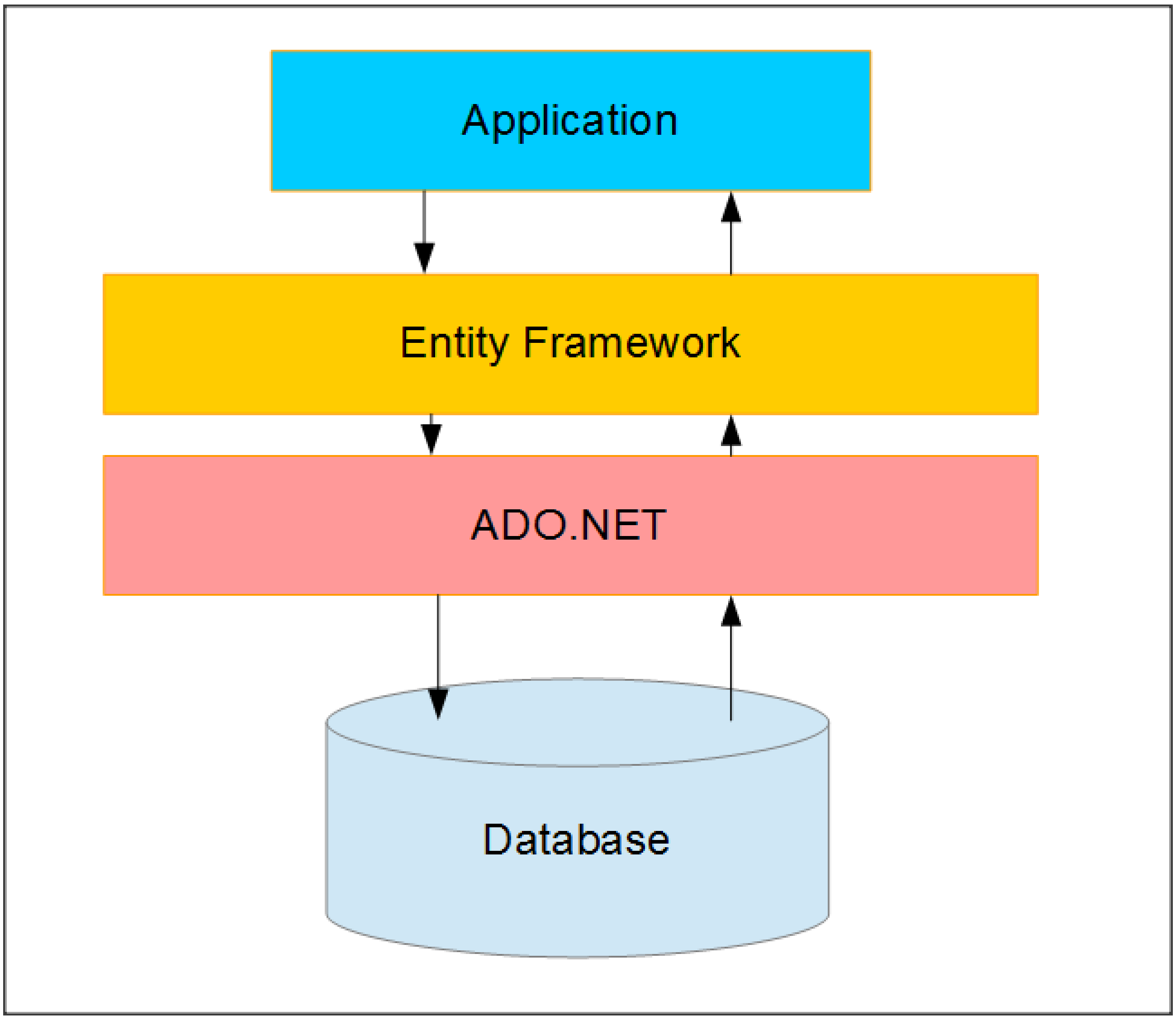
2015 Mastering Entity Framework

Chapter 1 Introduction to Entity Framework

- ✓ Entity Framework is an Object Relational Mapper (ORM) from Microsoft that lets the application's developers work with relational data as business models.
- ✓ Entity Framework eases the task of creating a data access layer by enabling the access of data, by representing the data as a conceptual model, that is, a set of entities and relationships.
- ✓ The application can perform the basic CRUD (create, read, update, and delete) operations and easily manage one-to-one, one-to-many, and many-to-many relationships between the entities.
- ✓ Here are a few benefits of using Entity Framework:
  - ● The development time is reduced since the developers don't have to write all the ADO.NET plumbing code needed for data access
  - ● We can have all the data access logic written in a higher-level language such as C# rather than writing SQL queries and stored procedures
  1. Since the database tables cannot have advanced relationships (inheritance) as the domain entities can, the business model, that is, the conceptual model can be used to suit the application domain using relationships among the entities.
  2. The underlying data store can be replaced relatively easily if we use an ORM since all the data access logic is present in our application instead of the data layer. If an ORM is not being used, it would be comparatively difficult to do so.

✓ The Entity Framework architecture



✓ **Understanding the Entity Data Model**

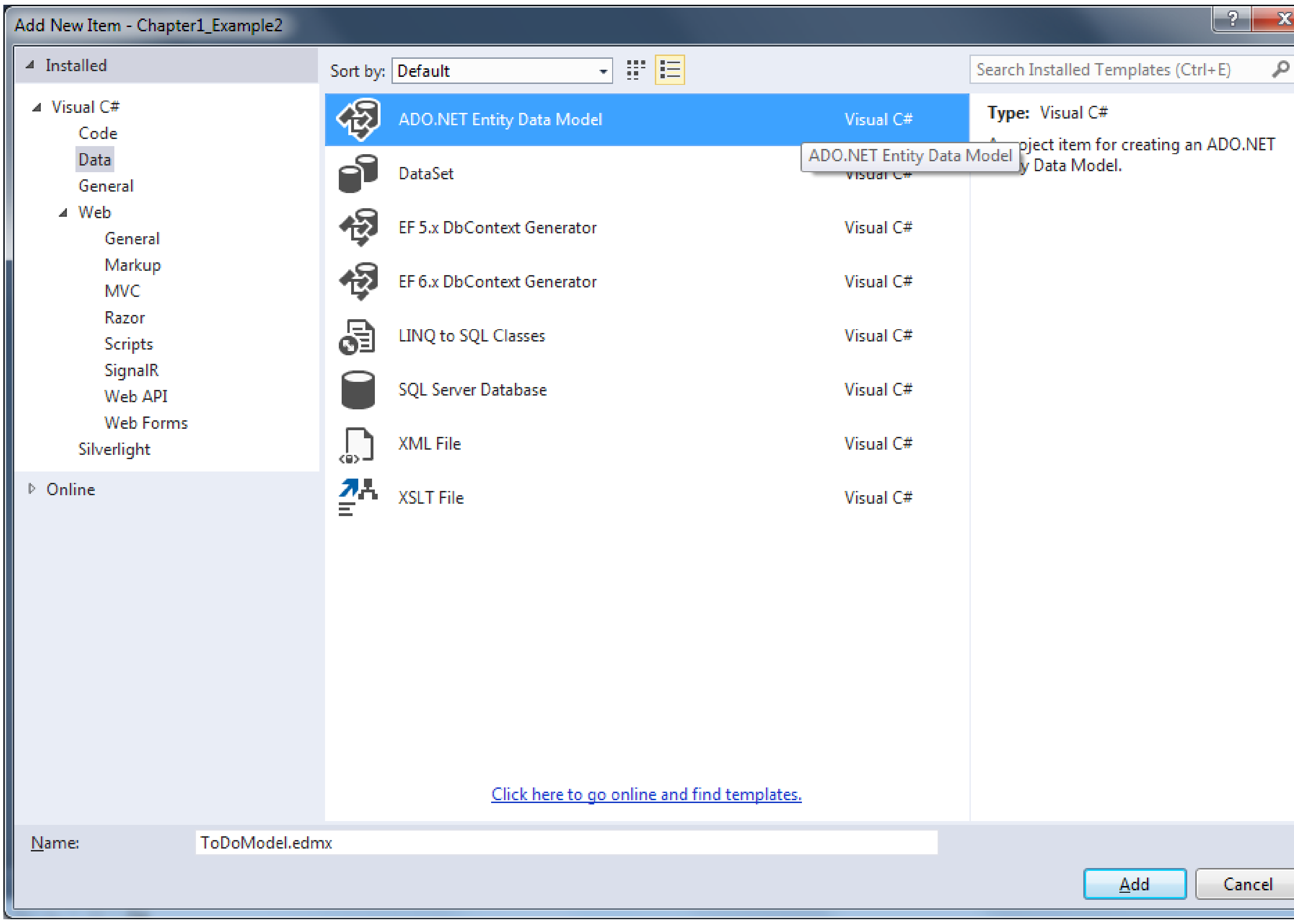
1. To use Entity Framework, we have to create the conceptual data model, that is, the Entity Data Model (EDM).
2. The EDM defines our conceptual model classes, the relationships between those classes, and the mapping of those models to the database schema.
3. Once our EDM is created, we can perform all the CRUD operations (create, retrieve, update, and delete) on our conceptual model, and Entity Framework will translate all these object queries to database queries (SQL).
4. Once these queries are executed, the results will again be converted to conceptual model object instances by Entity Framework.
5. Entity Framework will use the mapping information stored in the EDM to perform this translation of object queries to SQL queries, and the relational data to conceptual models.

✓ Understanding the ObjectContext class

1. The ObjectContext class is the main object in Entity Framework.
2. It is the class that is responsible for:
   - ◆ Managing database connection
   - ◆ Providing support to perform CRUD operations

        - ◆ Keeping track of changes in the models so that the models can be updated in the database

- ✓ Development styles and different Entity Framework approaches
    1. Database First: This is the approach that will be used with an existing database schema.
        - ◆ In this approach, the EDM will be created from the database schema.
        - ◆ This approach is best suited for applications that use an already existing database.
    2. Code First: This is the approach where all the domain models are written in the form of classes.
        - ◆ These classes will constitute our EDM.
        - ◆ The database schema will be created from these models.
        - ◆ This approach is best suited for applications that are highly domain-centric and will have the domain model classes created first.
        - ◆ The database here is needed only as a persistence mechanism for these domain models.
    3. Model First: This approach is very similar to the Code First approach, but in this case, we use a visual EDM designer to design our models.
        - ◆ The database schema and the classes will be generated by this conceptual model.
        - ◆ The model will give us the SQL statements needed to create the database, and we can use it to create our database and connect up with our application.
        - ◆ For all practical purposes, it's the same as the Code First approach, but in this approach, we have the visual EDM designer available.
- ✓ Comparing the development styles
    1. The Database First approach
        - ◆ When we are working with a legacy database.
        - ◆ When we are working in a scenario where the database design is being done by another team of DBAs, and once the database is ready, the application development starts.
        - ◆ When we are working on a data centric application, that is, the application domain model is the database itself, and it is being changed frequently to cater to new requirements. For instance, when the tables are being updated regularly and new columns are being added to it frequently then we can simply use this approach, and the application code will not break. We simply have to write the code to cater to the newly added columns.
    2. The Model First approach
        - ◆ The only reason to choose the Model First approach is that we really want to use the Visual Entity Designer.
    3. The Code First approach
        - ◆ The Code First approach is usually helpful where all the business logic is implemented in terms of classes, and the database is simply being used as a persistence mechanism for these models
        - ◆ The database is simply a persistence mechanism for the models, that is, no logic is in the database.
        - ◆ Full control over the code, that is, there is no auto-generated model and context code.
        - ◆ The database will not be changed manually. The model classes will always change and based on this, the database should change itself.
- ✓ Entity Framework Database First approach
    1. Add a new ADO.NET EDM



Add New Item - Chapter1_Example2

Installed

- Visual C#
  - Code
  - Data
  - General
  - Web
    - General
    - Markup
    - MVC
    - Razor
    - Scripts
    - SignalR
    - Web API
    - Web Forms
  - Silverlight
- Online

Sort by: Default

ADO.NET Entity Data Model	Visual C#
DataSet	Visual C#
EF 5.x DbContext Generator	Visual C#
EF 6.x DbContext Generator	Visual C#
LINQ to SQL Classes	Visual C#
SQL Server Database	Visual C#
XML File	Visual C#
XSLT File	Visual C#

Search Installed Templates (Ctrl+E)

Type: Visual C#

A project item for creating an ADO.NET Entity Data Model.

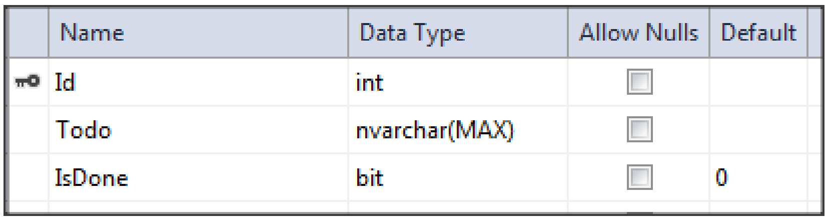
ADO.NET Entity Data Model

Click here to go online and find templates.

Name: ToDoModel.edmx

Add Cancel

2. A table schema for the sample ToDo application

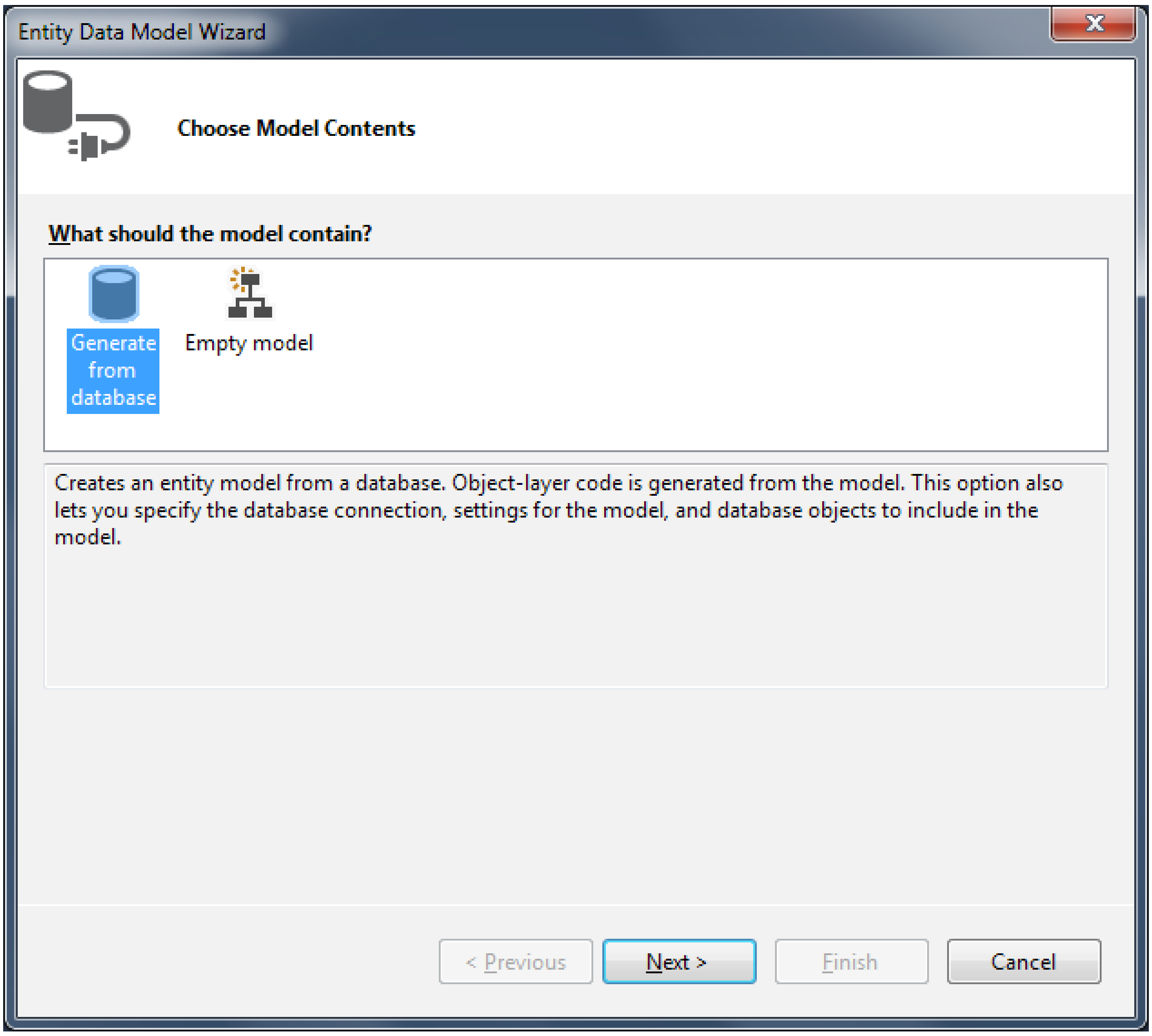


	Name	Data Type	Allow Nulls	Default
🔑	Id	int	☐	
	Todo	nvarchar(MAX)	☐	
	IsDone	bit	☐	0

3. The application requirements state that it should be possible to perform the following activities on the ToDos:

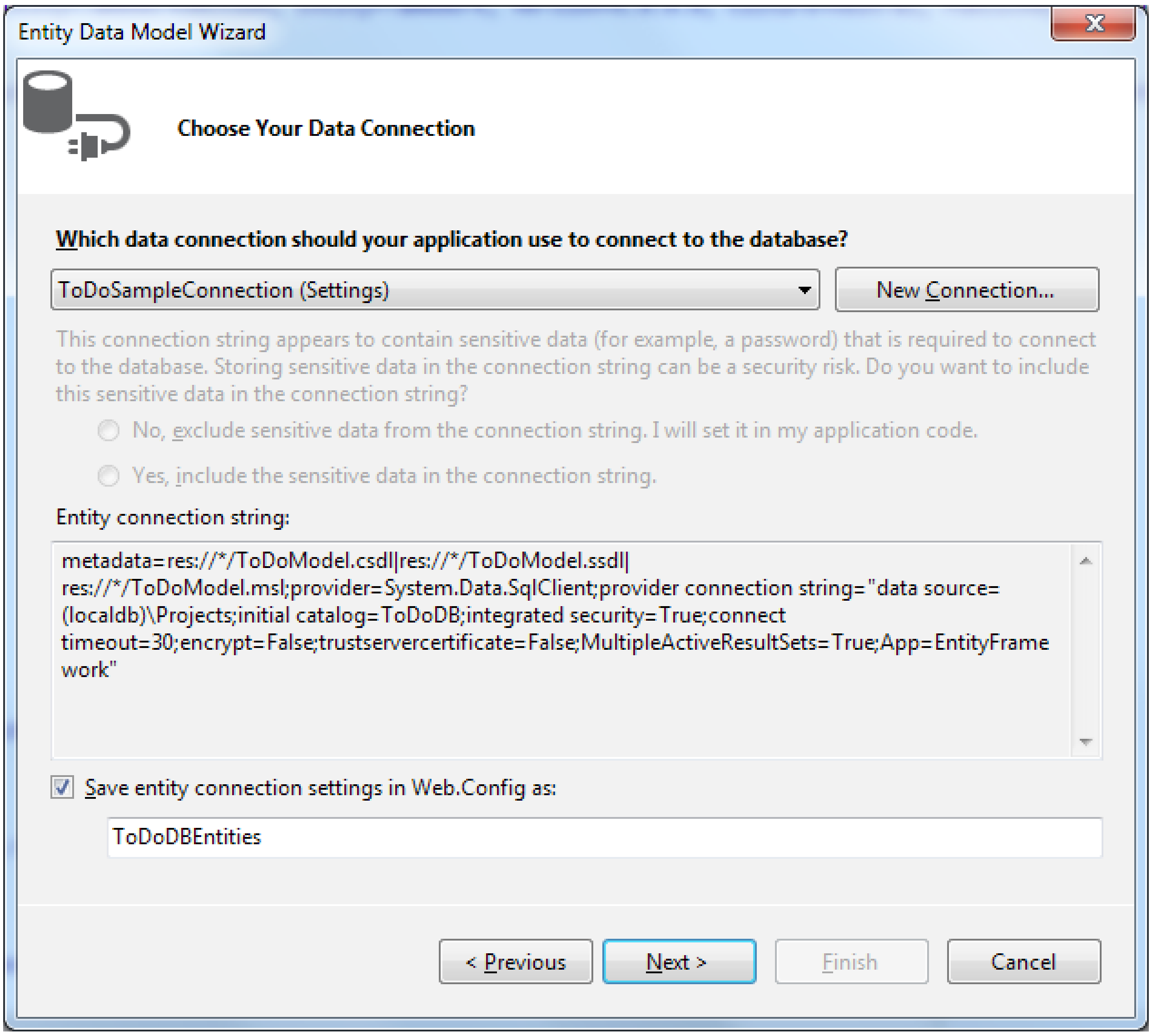
- Read the list of ToDos
- Create a ToDo item
- Read a specific ToDo item
- Update the status of a ToDo item
- Delete a ToDo item

4. ADO.NET EDM wizard step 1—generate from the database



Creates an entity model from a database. Object-layer code is generated from the model. This option also lets you specify the database connection, settings for the model, and database objects to include in the model.

5. ADO.NET EDM wizard step 2—select the database



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

ToDoSampleConnection (Settings)

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.

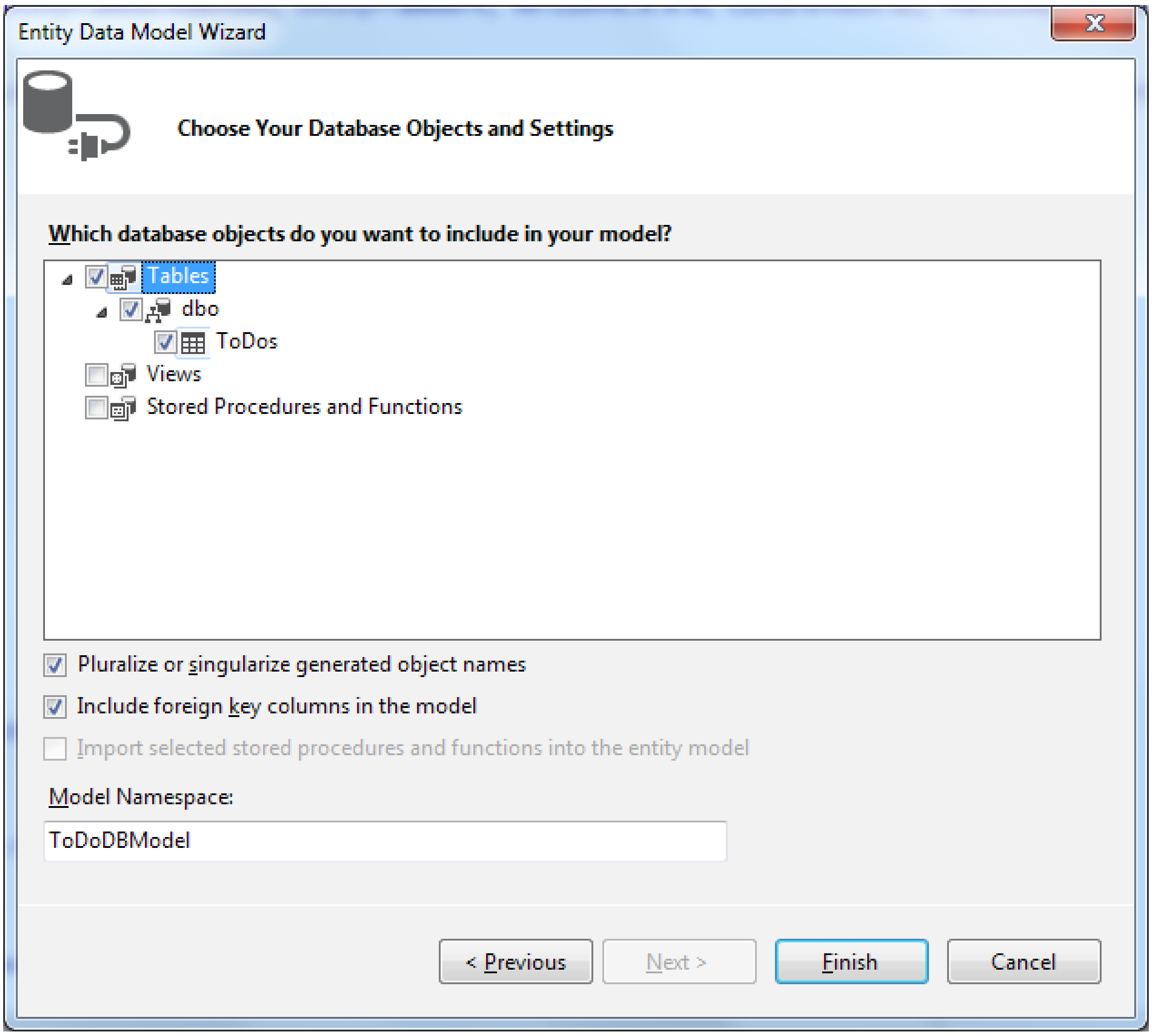
Entity connection string:

```
metadata=res://*/ToDoModel.csdl|res://*/ToDoModel.ssdl|
res://*/ToDoModel.msl;provider=System.Data.SqlClient;provider connection string="data source=
(localdb)\Projects;initial catalog=ToDoDB;integrated security=True;connect
timeout=30;encrypt=False;trustservercertificate=False;MultipleActiveResultSets=True;App=EntityFrame
work"
```

[✓] Save entity connection settings in Web.Config as:

ToDoDBEntities

6. ADO.NET EDM wizard step 3—select the schema elements to be included in the EDM



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

- [✓] Tables
  - [✓] dbo
    - [✓] ToDos
- [ ] Views
- [ ] Stored Procedures and Functions

[✓] Pluralize or singularize generated object names

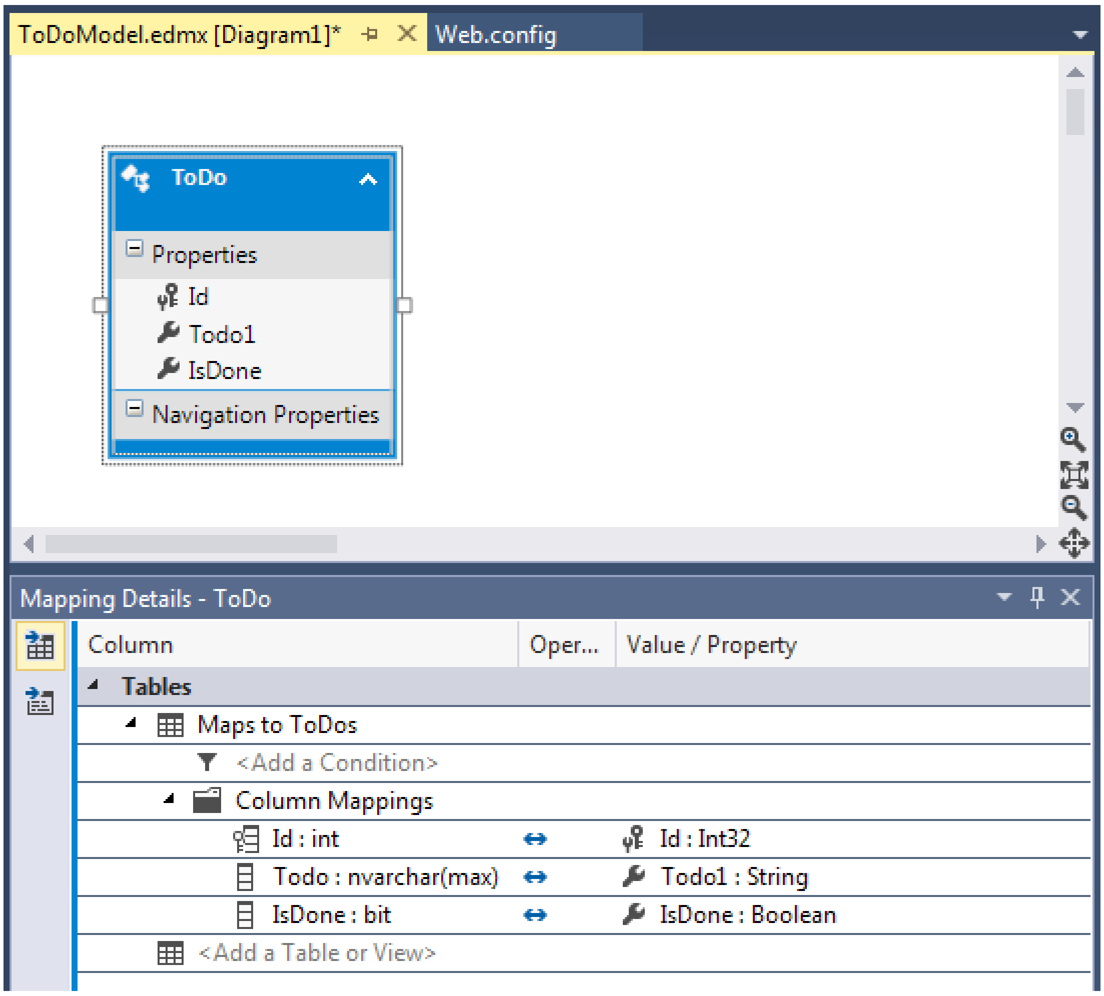
[✓] Include foreign key columns in the model

[ ] Import selected stored procedures and functions into the entity model

Model Namespace:

ToDoDBModel

7. Visual Entity Designer showing the default model


ToDoModel.edmx [Diagram1]* Web.config

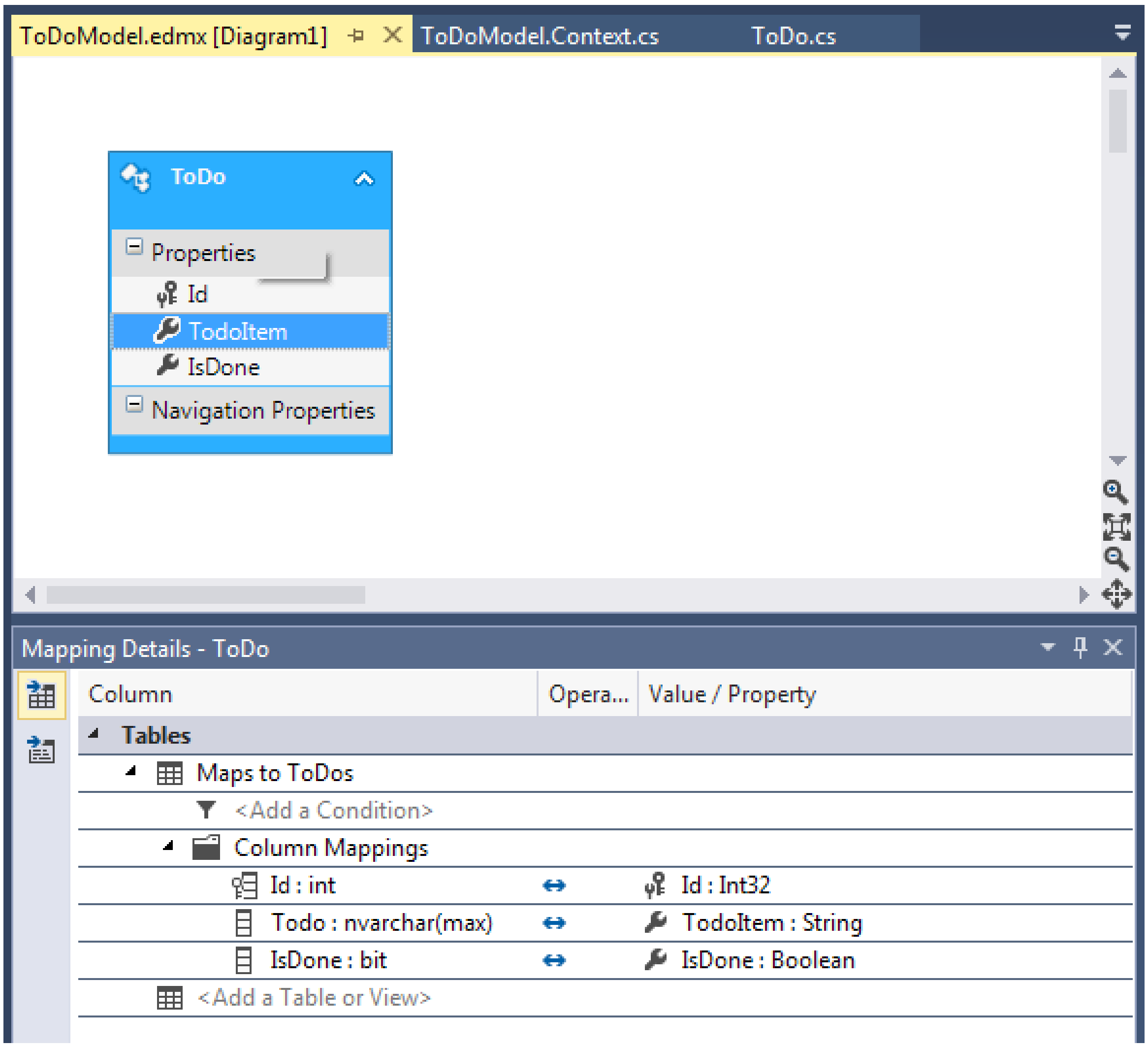
ToDo
Properties
- Id
- Todo1
- IsDone

Navigation Properties

Mapping Details - ToDo

Column	Oper...	Value / Property
Tables		
Maps to ToDos		
<Add a Condition>		
Column Mappings		
Id : int	↔	Id : Int32
Todo : nvarchar(max)	↔	Todo1 : String
IsDone : bit	↔	IsDone : Boolean
<Add a Table or View>		

8. Visual Entity Designer showing the customized model



Column	Opera...	Value / Property
Tables		
Maps to ToDos		
<Add a Condition>		
Column Mappings		
Id : int	↔	Id : Int32
Todo : nvarchar(max)	↔	TodoItem : String
IsDone : bit	↔	IsDone : Boolean
<Add a Table or View>		

9. EDMX file's three sections:

◆ Conceptual schema definition: This specifies how a strongly typed model for our conceptual model will be created:

```
<edmx:ConceptualModels>
  <Schema Namespace="ToDoDBModel" Alias="Self"
  annotation:UseStrongSpatialTypes="false"
  xmlns:annotation="http://schemas.microsoft.com/ado/2009/02/edm/annotation"
  xmlns="http://schemas.microsoft.com/ado/2009/11/edm">
    <EntityType Name="ToDo">
        <Key>
            <PropertyRef Name="Id" />
        </Key>
        <Property Name="Id" Type="Int32" Nullable="false" />
        <Property Name="TodoItem" Type="String" MaxLength="Max"
  FixedLength="false" Unicode="true" Nullable="false" />
        <Property Name="IsDone" Type="Boolean" Nullable="false" />
    </EntityType>
    <EntityContainer Name="ToDoDBEntities" annotation:LazyLoadingEnabled="true">
        <EntitySet Name="ToDos" EntityType="Self.ToDo" />
        </EntityContainer>
    </Schema>
</edmx:ConceptualModels>
```

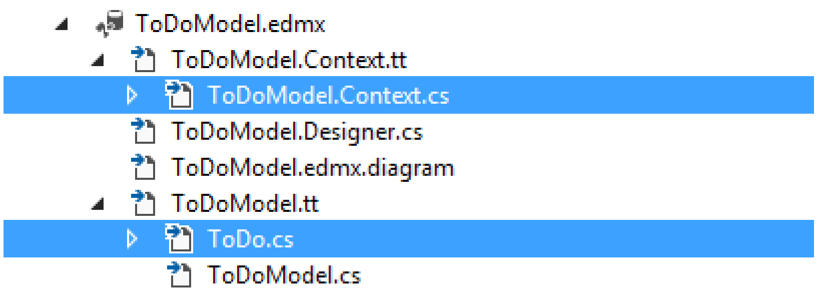
- Storage schema definition: This specifies how the storage model is created, that is, how the values are stored in the database:

```
<edmx:StorageModels>
    <Schema Namespace="ToDoDBModel.Store" Provider="System.Data.SqlClient"
    ProviderManifestToken="2012" Alias="Self"
    xmlns:store="http://schemas.microsoft.com/ado/2007/12/edm/EntityStoreSchemaGener
    ator" xmlns="http://schemas.microsoft.com/ado/2009/11/edm/ssdl">
        <EntityType Name="ToDos">
            <Key>
                <PropertyRef Name="Id" />
            </Key>
            <Property Name="Id" Type="int" Nullable="false" />
            <Property Name="Todo" Type="nvarchar(max)" Nullable="false" />
            <Property Name="IsDone" Type="bit" Nullable="false" />
        </EntityType>
        <EntityContainer Name="ToDoDBModelStoreContainer">
            <EntitySet Name="ToDos" EntityType="Self.ToDos" Schema="dbo"
            store:Type="Tables" />
        </EntityContainer>
    </Schema>
</edmx:StorageModels>
```

- Mapping: This specifies the mapping between the conceptual model and the storage model:

```
<edmx:Mappings>
    <Mapping Space="C-S"
    xmlns="http://schemas.microsoft.com/ado/2009/11/mapping/cs">
        <EntityContainerMapping StorageEntityContainer="ToDoDBModelStoreContainer"
        CdmEntityContainer="ToDoDBEntities">
            <EntitySetMapping Name="ToDos">
                <EntityTypeMapping TypeName="ToDoDBModel.ToDo">
                    <MappingFragment StoreEntitySet="ToDos">
                        <ScalarProperty Name="Id" ColumnName="Id" />
                        <ScalarProperty Name="TodoItem" ColumnName="Todo" />
                        <ScalarProperty Name="IsDone" ColumnName="IsDone" />
                    </MappingFragment>
                </EntityTypeMapping>
            </EntitySetMapping>
        </EntityContainerMapping>
    </Mapping>
</edmx:Mappings>
```

- Solution Explorer showing the generated DbContext and Model class



```
◢ ToDoModel.edmx
    ◢ ToDoModel.Context.tt
        ▷ ToDoModel.Context.cs
      ToDoModel.Designer.cs
      ToDoModel.edmx.diagram
    ◢ ToDoModel.tt
        ▷ ToDo.cs
          ToDoModel.cs
```

- The generated DBContext class will look like this:

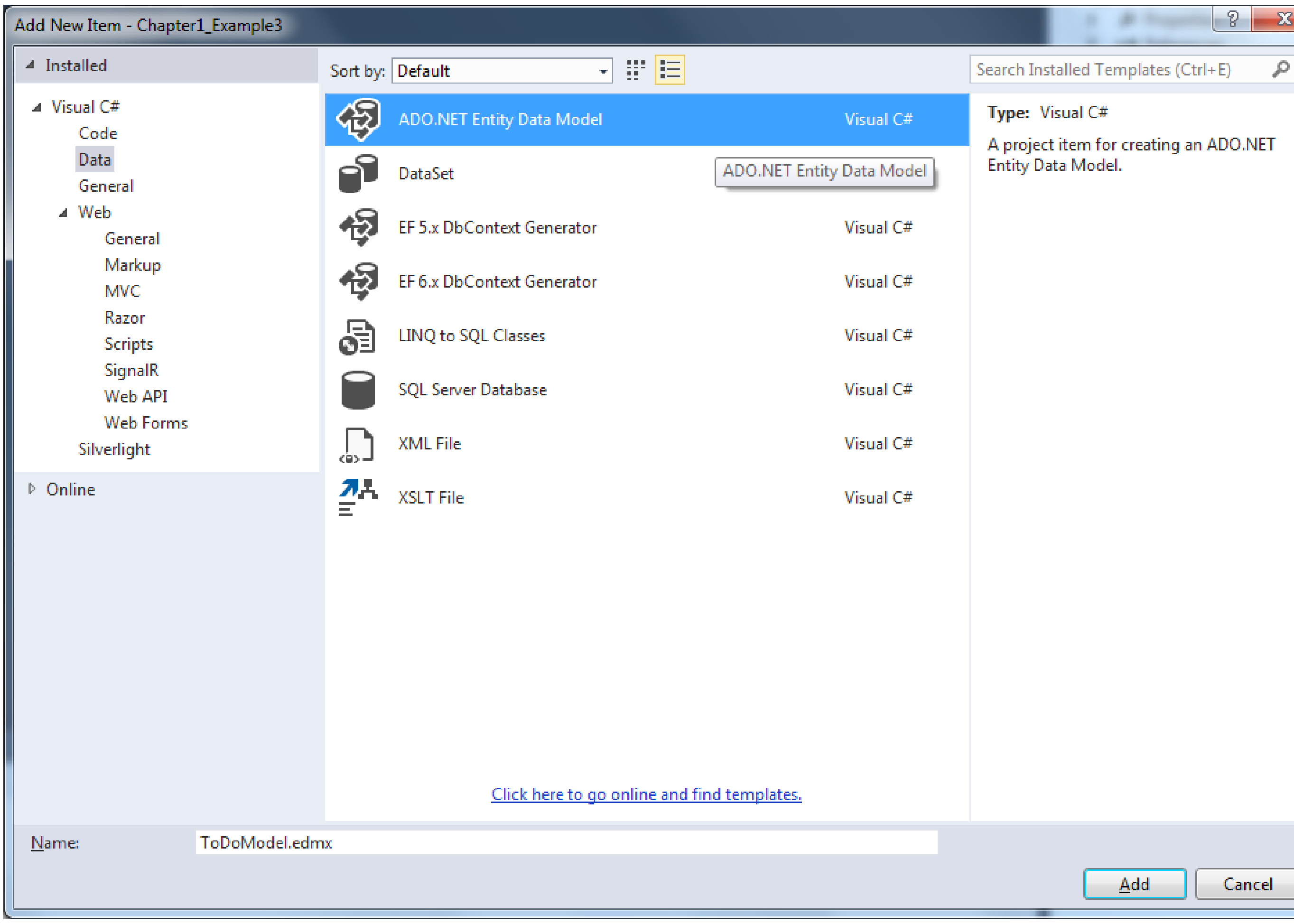
```
public partial class ToDoDBEntities : DbContext
{
    public ToDoDBEntities()
        : base("name=ToDoDBEntities")
    {
    }
    public virtual DbSet<ToDo> ToDos { get; set; }
}
```

- The generated model class will look like this:

```
public partial class ToDo
{
    public int Id { get; set; }
    public string TodoItem { get; set; }
    public bool IsDone { get; set; }
}
```

✓ Entity Framework Model First approach

1. Add a new ADO.NET EDM


Add New Item - Chapter1_Example3

Installed
- Visual C#
  - Code
  - Data
  - General
  - Web
    - General
    - Markup
    - MVC
    - Razor
    - Scripts
    - SignalR
    - Web API
    - Web Forms
  - Silverlight
- Online

Sort by: Default

Search Installed Templates (Ctrl+E)

ADO.NET Entity Data Model	Visual C#
DataSet	ADO.NET Entity Data Model
EF 5.x DbContext Generator	Visual C#
EF 6.x DbContext Generator	Visual C#
LINQ to SQL Classes	Visual C#
SQL Server Database	Visual C#
XML File	Visual C#
XSLT File	Visual C#

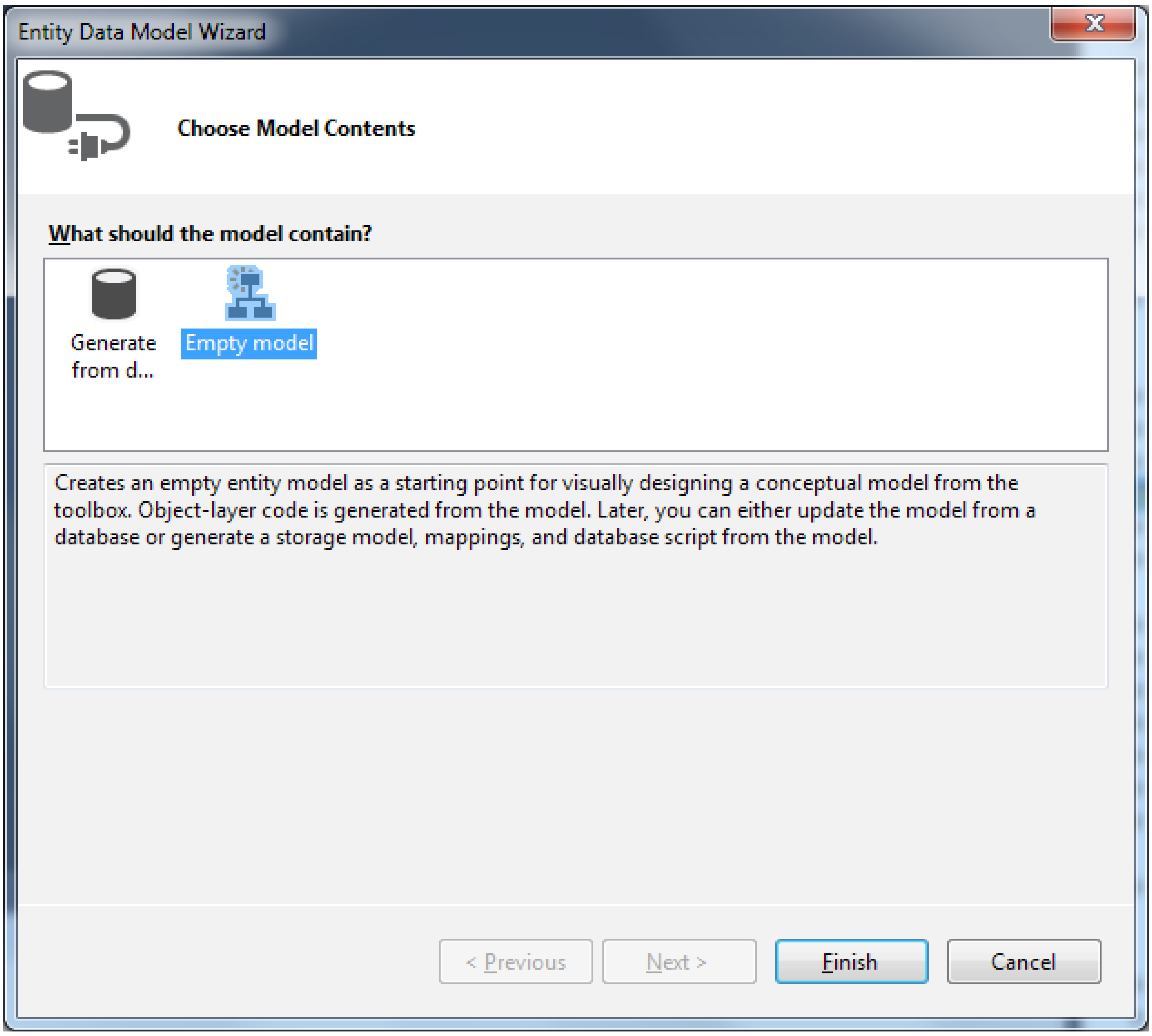
Type: Visual C#

A project item for creating an ADO.NET Entity Data Model.

Click here to go online and find templates.

Name: ToDoModel.edmx

2. ADO.NET EDM wizard step 1—select the Model First approach

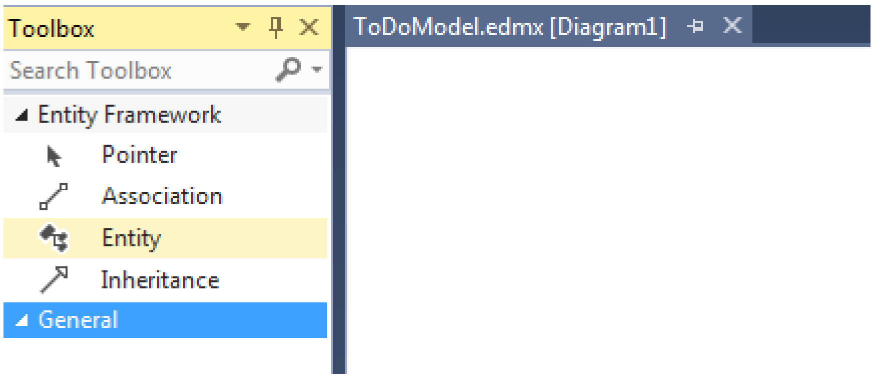

Entity Data Model Wizard

Choose Model Contents

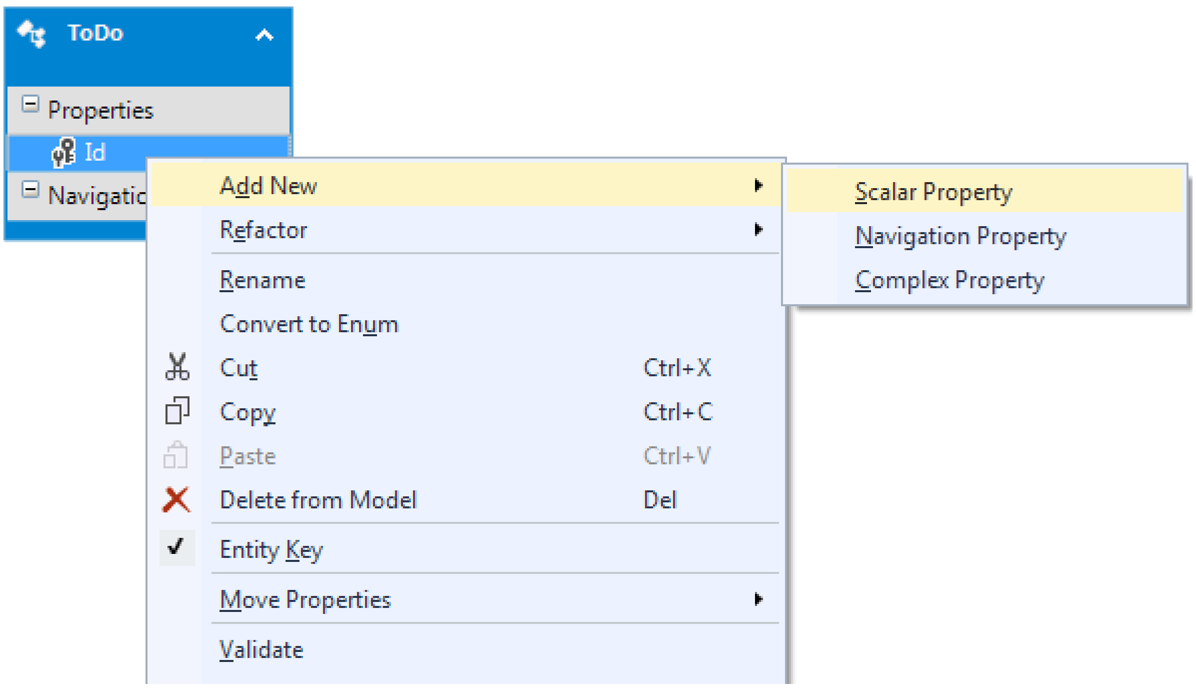
What should the model contain?

Creates an empty entity model as a starting point for visually designing a conceptual model from the toolbox. Object-layer code is generated from the model. Later, you can either update the model from a database or generate a storage model, mappings, and database script from the model.

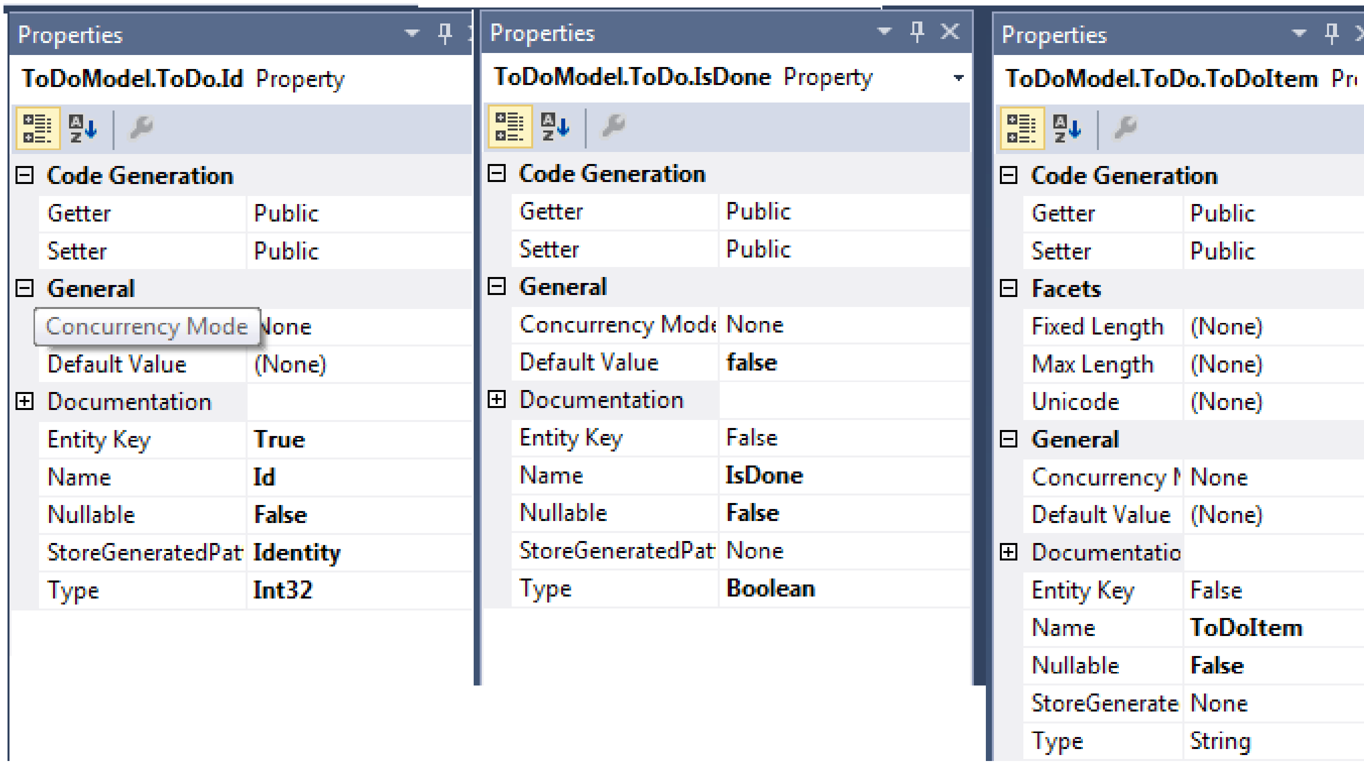
3. An empty Visual Entity Designer after selecting the Model First approach



4. Adding a new property from Visual Entity Designer



5. Visual Studio's Property panes showing the properties of a conceptual model



Properties

ToDoModel.ToDo.Id Property

Code Generation	
Getter	Public
Setter	Public
General	
Concurrency Mode	None
Default Value	(None)
Documentation	
Entity Key	**True**
Name	**Id**
Nullable	**False**
StoreGeneratedPat	**Identity**
Type	**Int32**

Properties

ToDoModel.ToDo.IsDone Property

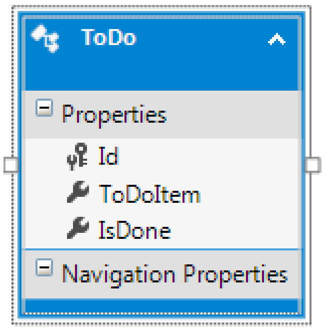
Code Generation	
Getter	Public
Setter	Public
General	
Concurrency Mode	None
Default Value	**false**
Documentation	
Entity Key	False
Name	**IsDone**
Nullable	**False**
StoreGeneratedPat	None
Type	**Boolean**

Properties

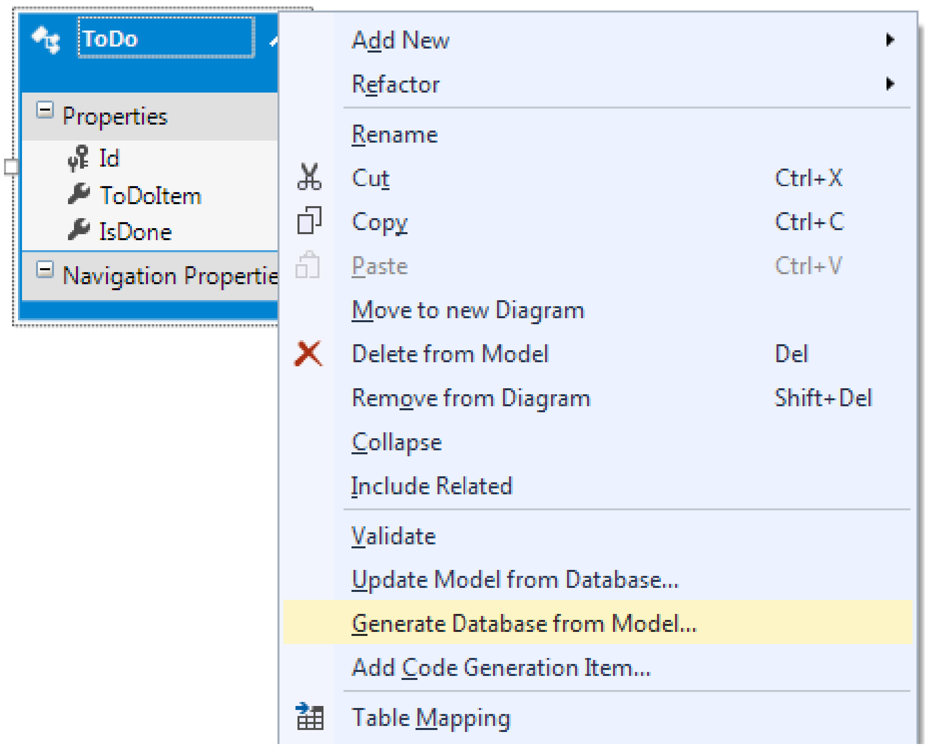
ToDoModel.ToDo.ToDoItem Pr

Code Generation	
Getter	Public
Setter	Public
Facets	
Fixed Length	(None)
Max Length	(None)
Unicode	(None)
General	
Concurrency M	None
Default Value	(None)
Documentatio	
Entity Key	False
Name	**ToDoItem**
Nullable	**False**
StoreGenerate	None
Type	String

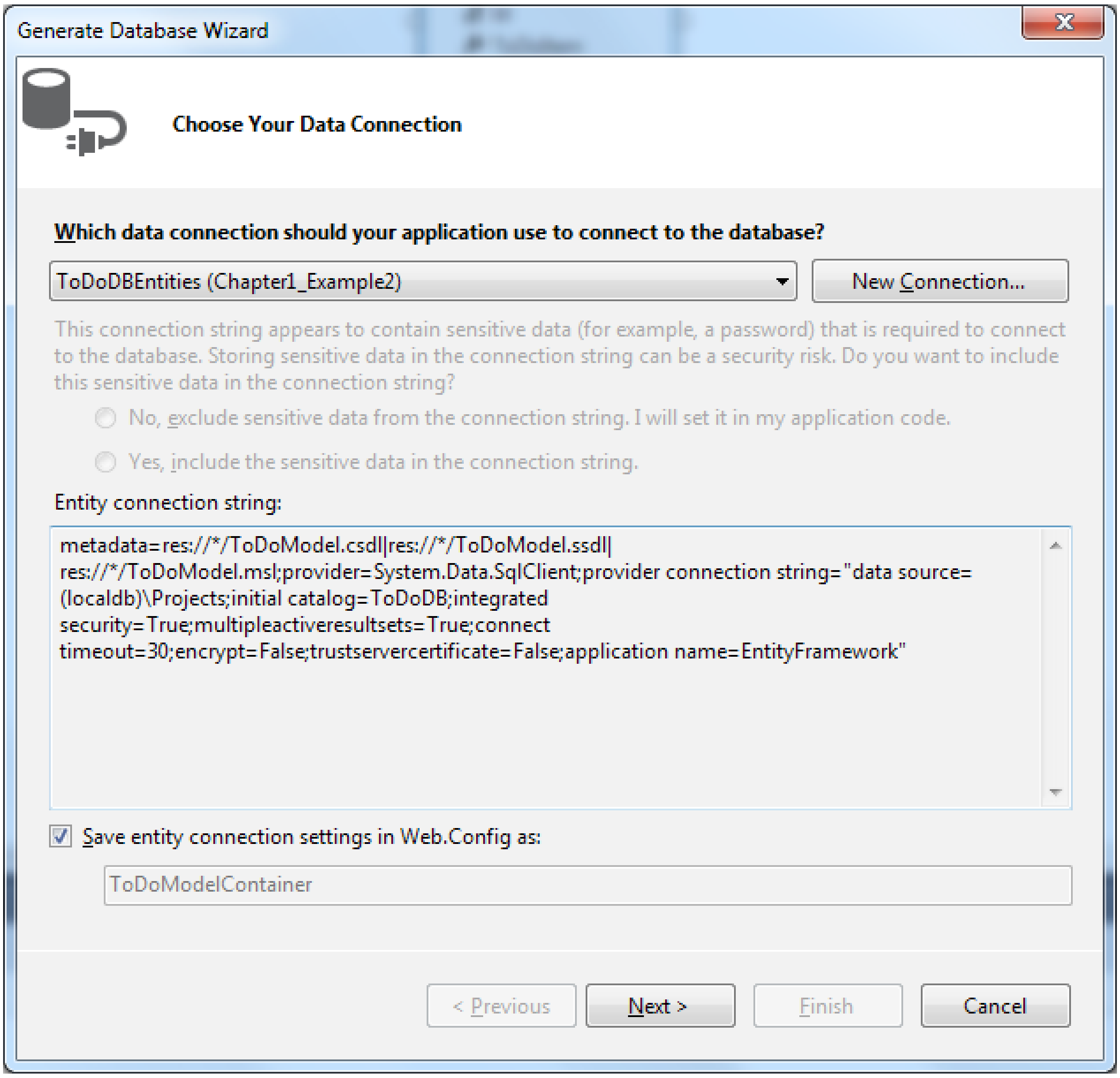
6. The final model created from the Visual Entity Designer



7. Generating database scripts from the Visual Entity Designer



8. Wizard step to select the connection string to be used for a generated database script



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

ToDoDBEntities (Chapter1_Example2)

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.

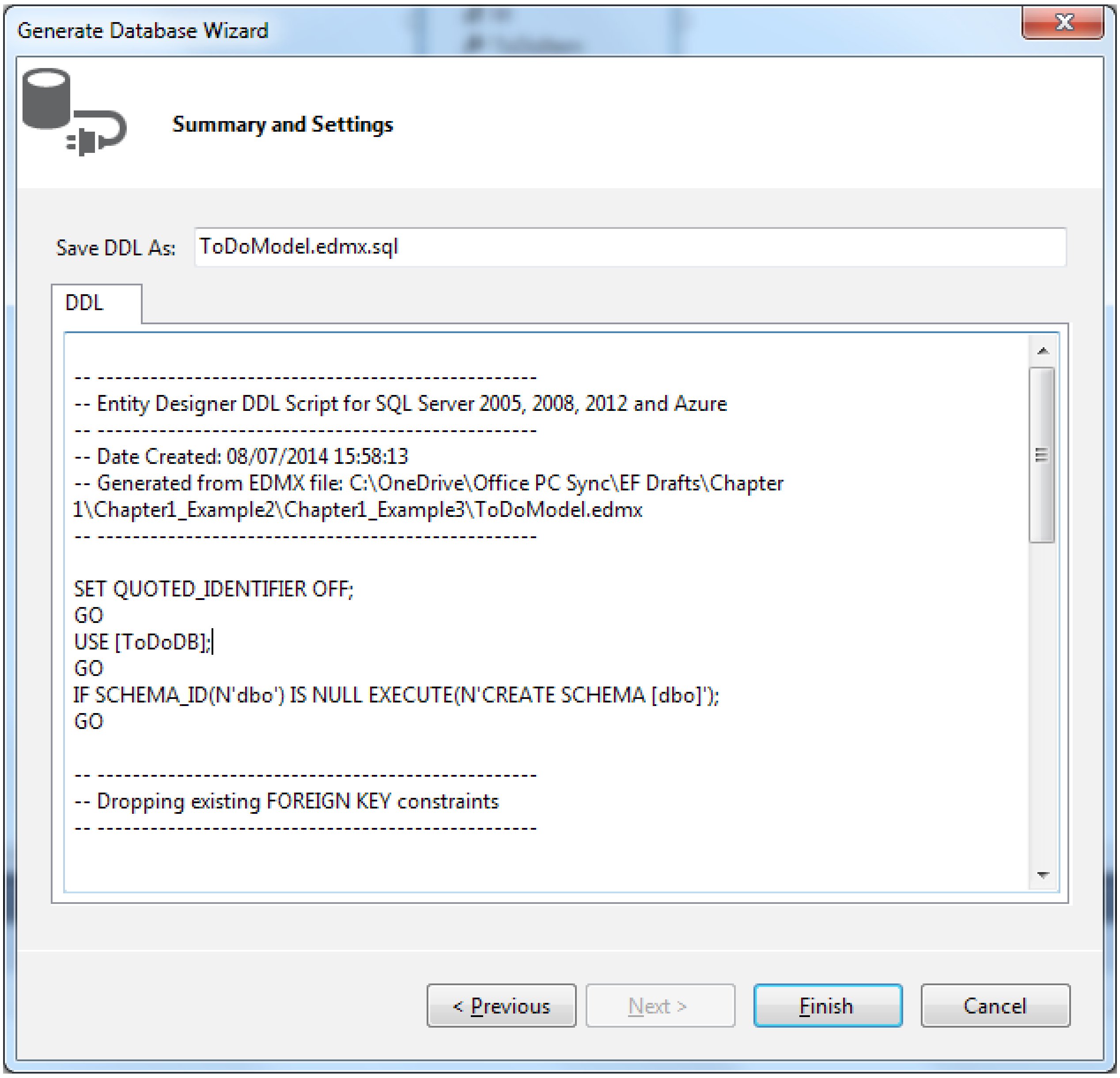
Entity connection string:

```
metadata=res://*/ToDoModel.csdl|res://*/ToDoModel.ssdl|
res://*/ToDoModel.msl;provider=System.Data.SqlClient;provider connection string="data source=
(localdb)\Projects;initial catalog=ToDoDB;integrated
security=True;multipleactiveresultsets=True;connect
timeout=30;encrypt=False;trustservercertificate=False;application name=EntityFramework"
```

☑ Save entity connection settings in Web.Config as:

ToDoModelContainer

9. Wizard step showing the create database script



Save DDL As: ToDoModel.edmx.sql

```
-- --------------------------------------------------
-- Entity Designer DDL Script for SQL Server 2005, 2008, 2012 and Azure
-- --------------------------------------------------
-- Date Created: 08/07/2014 15:58:13
-- Generated from EDMX file: C:\OneDrive\Office PC Sync\EF Drafts\Chapter
1\Chapter1_Example2\Chapter1_Example3\ToDoModel.edmx
-- --------------------------------------------------

SET QUOTED_IDENTIFIER OFF;
GO
USE [ToDoDB];
GO
IF SCHEMA_ID(N'dbo') IS NULL EXECUTE(N'CREATE SCHEMA [dbo]');
GO

-- --------------------------------------------------
-- Dropping existing FOREIGN KEY constraints
-- --------------------------------------------------
```

10. The generated DbContext class:

```
public partial class ToDoModelContainer : DbContext
{
    public ToDoModelContainer() : base("name=ToDoModelContainer")
    {
    }
    public virtual DbSet<ToDo> ToDoes { get; set; }
}
```

11. The generated ToDo class:

```
public partial class ToDo
{
    public ToDo()
    {
        this.IsDone = false;
    }
    public int Id { get; set; }
    public string ToDoItem { get; set; }
    public bool IsDone { get; set; }
}
```

✓ Entity Framework Code First approach

1. Create a simple ToDo class that will keep track of our ToDo items:

```
public class ToDo
{
    public int Id { get; set; }
    public string ToDoItem { get; set; }
    public bool IsDone { get; set; }
}
```

2. Resultant ToDo model will look like this:

```
[Table("ToDo")] // Table name
public class ToDo
{
    [Key] // Primary key
        public int Id { get; set; }
    [Column("ToDoItem", TypeName="ntext")]
        public string ToDoItem { get; set; }
    [Column("IsDone", TypeName="bit")]
        public bool IsDone { get; set; }
}
```

3. DBContext class:

```
public partial class ToDoDBContext : DbContext
{
    public ToDoDBContext() : base("name=ToDoConnectionString")
    {
    }
    public DbSet<ToDo> ToDoes { get; set; }
}
```

✓ Performing CRUD operations using Entity Framework

1. Reading a list of items

```
using(ToDoDBEntities db = new ToDoDBEntities())
{
    IEnumerable<ToDo> todoItems = db.ToDos;
}
```

(1) We created an object of the DbContext class:
ToDoDBEntities db = new ToDoDBEntities();

(2) We then used the DbContext object to fetch the ToDo entities:
IEnumerable<ToDo> todoItems = db.ToDos;
After this call, the list of ToDo items will be available in the ToDo items collection if we

enumerate ToDo items. Entity Framework will internally perform the following activities for us to fetch the result:

a. Parsed our request for the data.
b. Generated the needed SQL query to perform this action.
c. Used the context class to wire up the SQL to the database.
d. Fetched the results from the database.
e. Created the strongly typed models from the retrieved results and return them to the user.

2. **Reading a specific item**

```
using (ToDoDBEntities db = new ToDoDBEntities())
{
   ToDo todo1 = db.ToDos.Find(id);
   ToDo todo3 = db.ToDos.FirstOrDefault(item => item.TodoItem == "Test item");
   ToDo todo2 = db.ToDos.SingleOrDefault(item => item.TodoItem == "Test item");
}
```

(1) We created an object of the DbContext class:
ToDoDBEntities db = new ToDoDBEntities()
(2) Retrieved an item by passing the key values, that is, using the Find function:
ToDo todo1 = db.ToDos.Find(id);
(3) Retrieved an item by passing a non-key value using the FirstOrDefault function:
ToDo todo2 = db.ToDos.SingleOrDefault(item => item.TodoItem == "Test item");
(4) Retrieved an item by passing a non-key value using the SingleOrDefault function:
ToDo todo2 = db.ToDos.SingleOrDefault(item => item.TodoItem == "Test item");

3. **Creating a new item**

```
using (ToDoDBEntities db = new ToDoDBEntities())
{
   ToDo todoItem = new ToDo();
   todoItem.TodoItem = "This is a test item.";
   todoItem.IsDone = false;
   db.ToDos.Add(todoItem);
   db.SaveChanges();
}
```

(1) We created an object of the DbContext class:
ToDoDBEntities db = new ToDoDBEntities()
(2) Created a new ToDo item:
ToDo todoItem = new ToDo();
(3) Populated the ToDo item properties with the desired values:
todoItem.TodoItem = "This is a test item.";
todoItem.IsDone = false;
(4) Added the item to the context class:
db.ToDos.Add(todoItem);
(5) Called Entity Framework's SaveChanges method to persist these changes in the database:

```
db.SaveChanges();
```

4. **Updating an existing item**

```
using (ToDoDBEntities db = new ToDoDBEntities())
{
   int id = 3;
   ToDo todo = db.ToDos.Find(id);
   todo.TodoItem = "This has been updated";
   todo.IsDone = false;
   db.SaveChanges();
}
```

(1) We created an object of the DbContext class:
```
ToDoDBEntities db = new ToDoDBEntities()
```
(2) Fetched the item with id = 3:
```
int id = 3;
ToDo todo = db.ToDos.Find(id);
```
(3) Updated the model properties:
```
todo.TodoItem = "This has been updated";
todo.IsDone = false;
```
(4) Called Entity Framework's SaveChanges method to persist these changes in the database:
```
db.SaveChanges();
```
(5) Attaching the entity back to the DbContext class, if it is being passed from a different tier:
```
using (ToDoDBEntities db = new ToDoDBEntities())
{
   db.Entry(todo).State = EntityState.Modified;
   db.SaveChanges();
}
```
①、We created an object of the DbContext class:
```
ToDoDBEntities db = new ToDoDBEntities()
```
②、Attached the model to the DbContext class and marked this model as modified:
```
db.Entry(todo).State = EntityState.Modified;
```
③、Called Entity Framework's SaveChanges method to persist these changes in the database:
```
db.SaveChanges();
```

5. **Deleting an item**

```
using (ToDoDBEntities db = new ToDoDBEntities())
{
   int id = 3;
   ToDo todo = db.ToDos.Find(id);
   db.ToDos.Remove(todo);
   db.SaveChanges();
}
```

(1) We created an object of the DbContext class:

```
ToDoDBEntities db = new ToDoDBEntities()
```

(2) Fetched the item with id = 3:

```
int id = 3;
ToDo todo = db.ToDos.Find(id);
```

(3) Removed the item from the collection:

```
db.ToDos.Remove(todo);
```

(4) Called Entity Framework's SaveChanges method to persist these changes in the database:

```
db.SaveChanges();
```

6. Choosing persistence approaches

Question	**Approach**
Is there a legacy database or does the database already exist?	Database First
Will we be getting a database created by the DBAs before starting the development?	Database First
Will there be frequent database changes, based on which our application should change?	Database First
Do we want to use the Visual Entity Designer to generate the database and model classes?	Model First
Do we have the model classes already and we need the database to save the data only?	Code First
Do we want to write all the model classes, implement them, and then think about the database storage later?	Code First
We don't want to deal with the auto-generated classes and would prefer to write them ourselves	Code First
Is the answer to all the preceding questions "no"?	Database First

Chapter 2 Entity Framework DB First – Managing Entity Relationships

✓