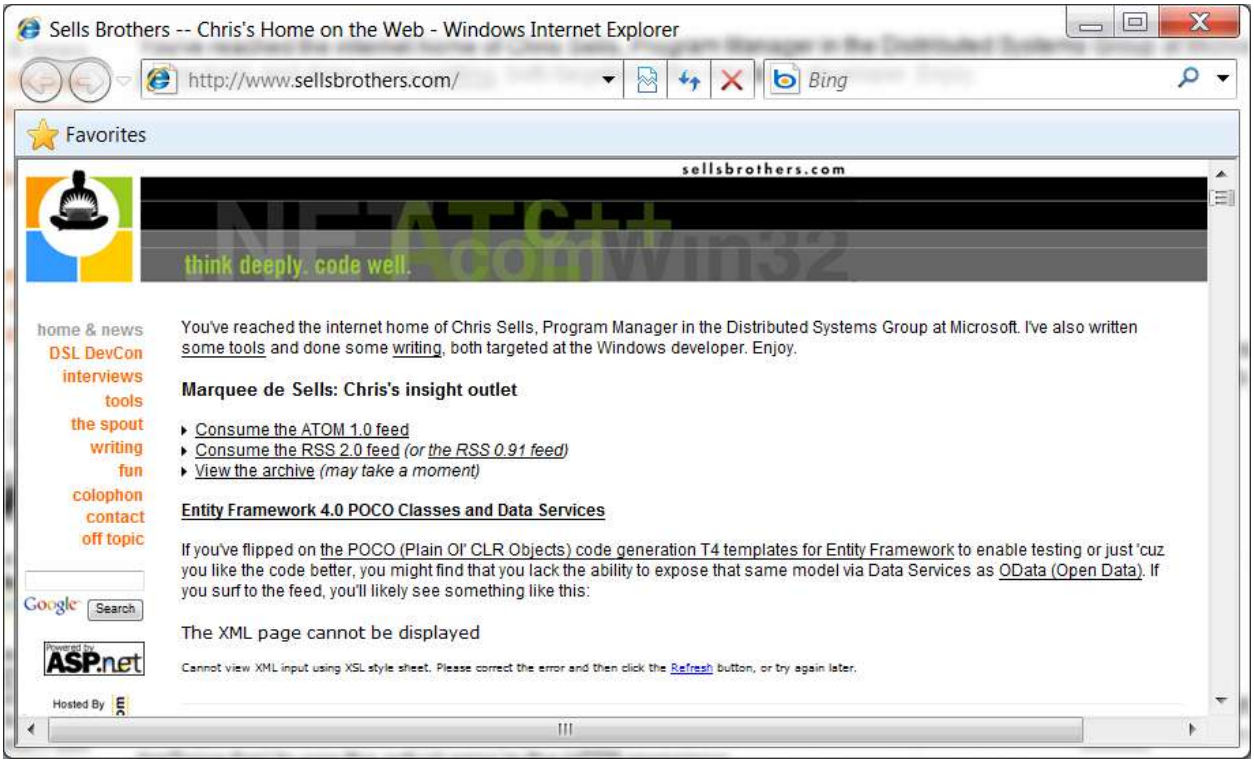
Hello, Data

In this chapter, we'll take a look at the beginnings of a real application as a way to explore the overall set of data access and data management technologies that come out of the box in Visual Studio 2010, including model-first design, the Entity Data Model, the Entity Framework, Database Projects and the Open Data Protocol.

Modeling Our Data with the Entity Data Model

My web site, sellsbrothers.com, was first built as a single static web page in 1995 and has grown haphazardly to encompass nearly 7,000 code and content files since. Unfortunately, it's accumulated several dead-ends and rough corners that make it very difficult to accommodate new features. After nearly 15 years of patching by me and my friends, it's time to start again, this time using the latest in custom web and data-centric application development technologies provided with Microsoft's Visual Studio 2010[1].

While the old sellsbrothers.com site is a creaky amalgam of static and dynamic pages, duplicate code and data spread hither and yon, the functionality on the home page is largely what I'd like it to be, as shown in Figure 1.



[1] If you're building a content-driven site like mine, there are several free and commercial content management systems (CMS) you should consider before building one from scratch as I'm doing in this example. For coverage of ASP.NET, I recommend "Professional ASP.NET MVC 2.0," by Jon Galloway, et al. (http://tinysells.com/124)

Figure 1: The old, busted sellsbrothers.com

As you can see, it's a very typical web site with a header, menu items and the main content in the center. Also, if you scroll down, you'll see the same set of menu items across the bottom. The content itself is split into "posts," rendered as either HTML or in a feed (like RSS, ATOM[2] or OData[3]) for consumption by external tools. Each post consists of a title, a creation date, the HTML of the content itself and a list of zero or more associated comments (which, in turn, each have an author, content and a creation date of their own). The schema associated with this little bit of data is easy to write down in a number of textual tools, but I prefer the graphical Entity Designer available in Visual Studio 2010.

The Entity Designer is for use modeling your data schema in the context of an application, so let's get started building new a sellsbrothers.com with a new ASP.NET MVC 2 Web Application, as shown in Figure 2.

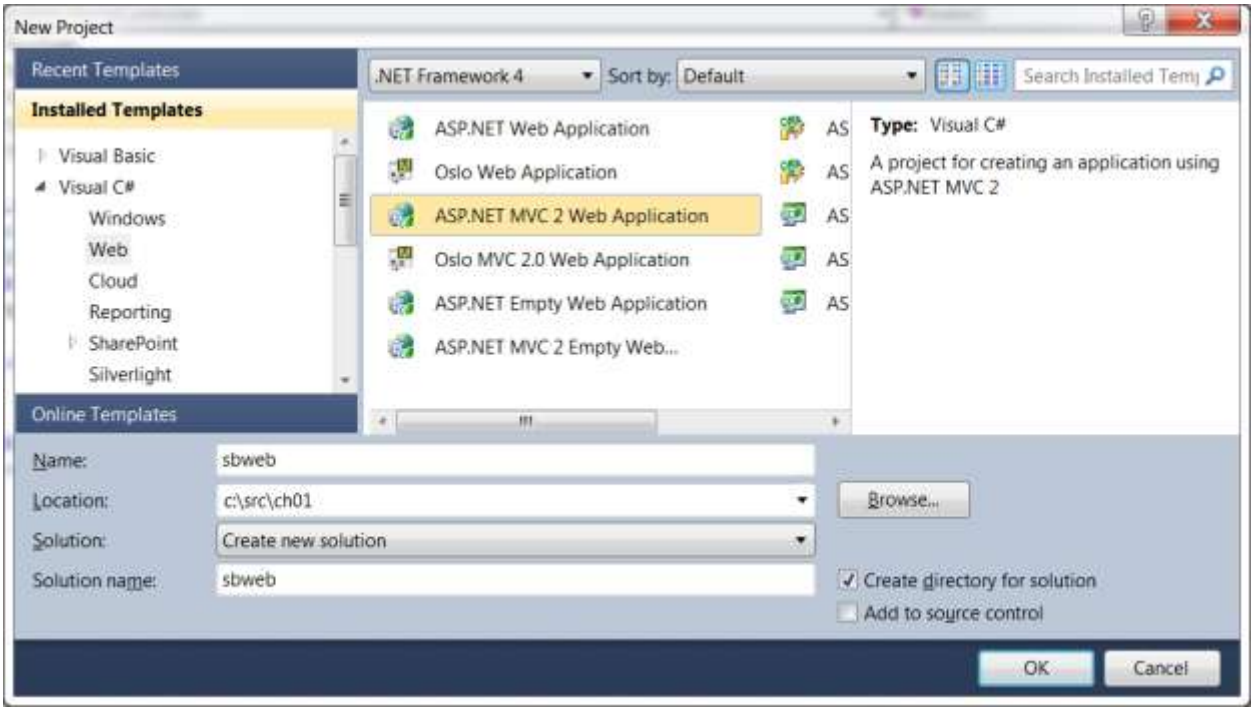


Figure 2: The Visual Studio 2010 New Project dialog

Before we can build any reasonable kind of web site, we'll need the data, so let's add a data model. Do that by right-clicking on the Models folder of our MVC project[4], selecting Add | New Item, choosing the

[2] The ATOM protocol, defined in IETF RFC 5023 (http://ietf.org/rfc/rfc5023.txt), is for exposing blog-style data via XML in a standard way so that people can write programs around the data, e.g. RSS Bandit, Google Reader or the feed support in Microsoft Internet Explorer.

[3] OData, or the "Open Data Protocol", is defined at http://odata.org and allows structured data to be passed using the ATOM protocol. It supports any kind of data to be exposes, not just blog data, which supports a a much larger set of programs to be written. We'll see more about it later.

[4] The M in MVC stands for Model, which means that data that drives our application, which is why we're dropping our data model in that folder. You don't have to, but it's an ASP.NET MVC convention. In case you're curious, the V and the C stand for View and Controller respectively.

ADO.NET Entity Data Model item from the Data category, entering the name of your model file (.edmx file) and pressing Add, as shown in Figure 3.

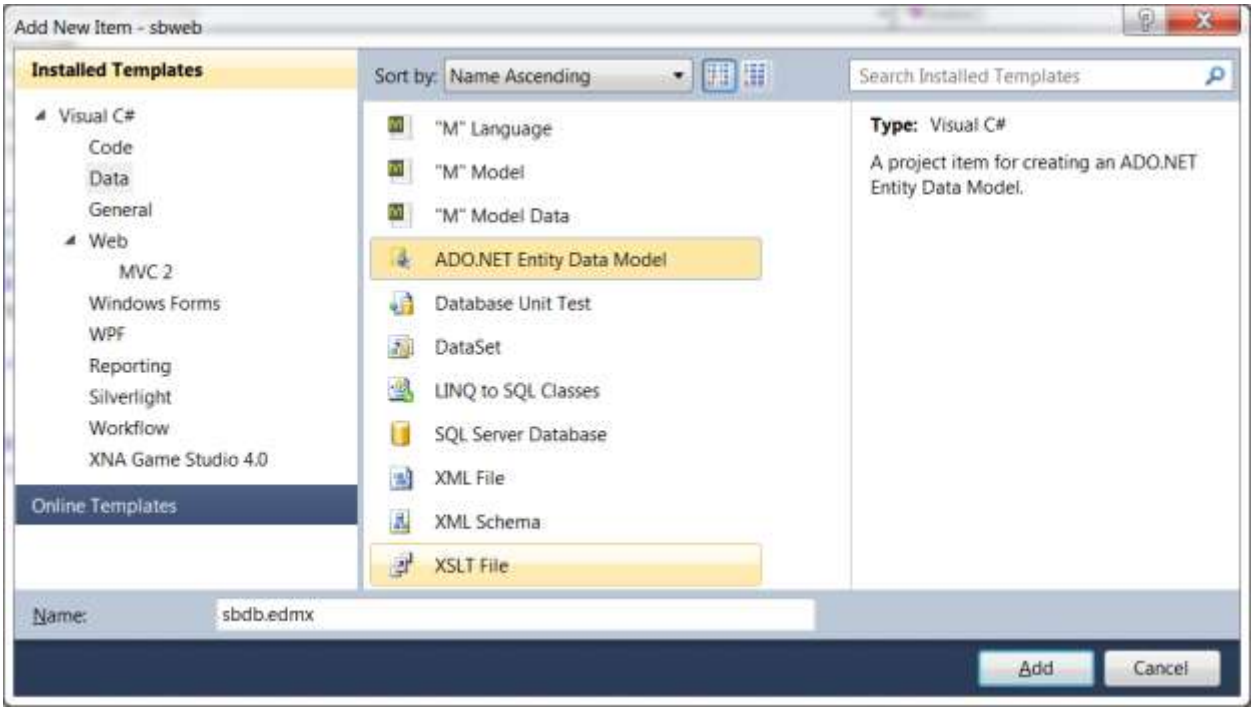


Figure 3: Adding a new ADO.NET Entity Data Model

When you press Add, you'll get to choose whether you'd like to generate the data model from a database or build it from scratch yourself (Figure 4).

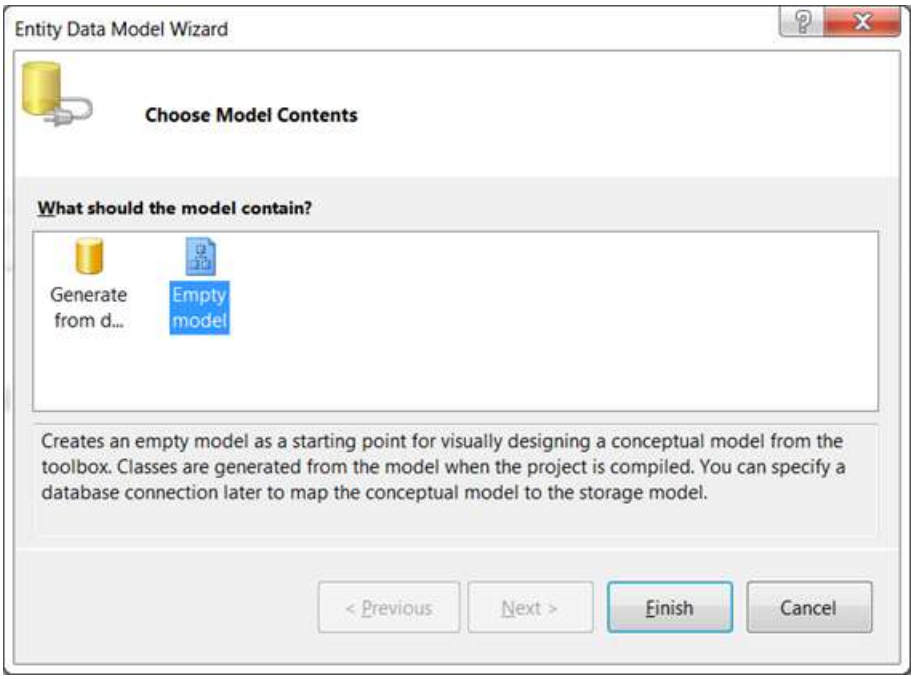


Figure 4: The Entity Data Model Wizard dialog

If you want to use a "database-first" style of development, where the schema is already defined in an existing SQL database, then you'd choose "Generate from database." This option lets you pick the tables, views and stored procedures you'd like to map into your .NET application. Since we're modeling our data anew and we plan on deriving the database from the model, we'll choose "Empty model". This is a "model-first" style of development[5] and we'll start with an empty space onto which to drag our entities and associations.

"Entities" and "associations" are the core of the Entity Data Model (EDM), which is what the design surface will let us manipulate to create our web site data model. An "entity" is simply an aggregate type that represents the grouping of our data. For example, we're going to have a Post type and a Comment type. Each entity has properties and associations. A "property" is a typed name value pair, e.g. the Post entity is going to have a CreationDate property of type DateTime. On the other hand, an "association" is a named relationship of one entity to another. For example, each Post entity will have an association to zero or more Comment entities. Likewise, each Comment will have an association with exactly one Post.

A little bit of drag 'n' drop and some naming and we get the start of our data model as shown in Figure 5.

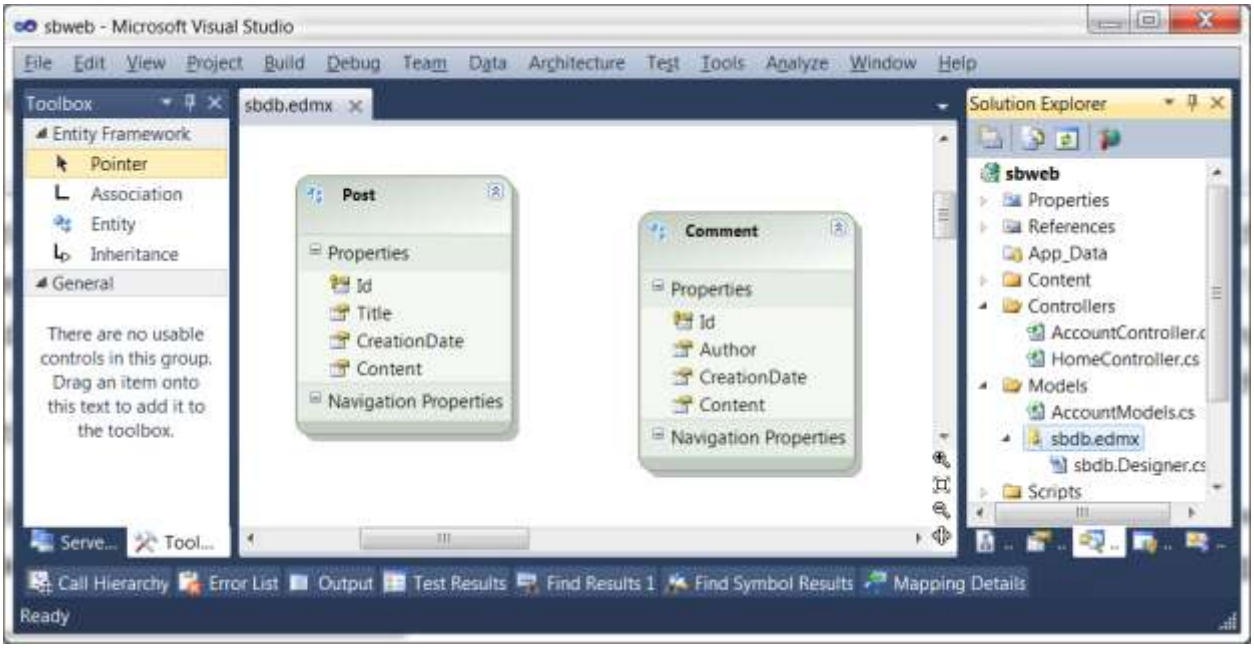


Figure 5: Building a data model in the Entity Designer

Besides the properties we've already talked about, you'll notice that both Post and Comment each have an Id property, which are unique identifiers for the instances of our entities. You'll notice the little key on each of them because they've got properties set to make it clear in the EDM that those properties

[5] The third type of data-based development you'll hear about is "code-first," which we'll talk about in Chapter 2: Entity Framework.

are special. You can see the properties of any property by right-clicking the property and choosing "Properties"[6]. Figure 6 shows the properties for the Id property of the Post entity.

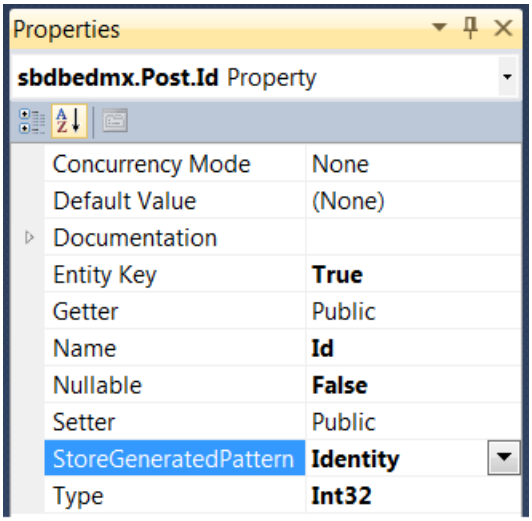


Figure 6: The Post.Id entity property Properties dialog

Notice that the Id property is marked as an Entity Key, which is what we've been talking about in terms of it being "special" as far as EDM is concerned. Notice also that it has a "StoreGeneratedPattern" of "Identity". This is just a fancy way of saying that we're gonna let the database generate unique identifiers for us (we'll see how that works later). The other important things to notice are that the Id property has no default value, that the property is public for getting and setting, it must be set (it can't be "null") and it's of type 32-bit integer[7].

By setting the properties on the rest of the entity properties, we can make sure the Title and Author strings don't go beyond a maximum length and that both CreationDate properties are DateTime instead of String, which is the default. However, we're not quite done yet because we don't have a way to associate posts with comments, which is a pretty important part of our data model. We can add an association by right-clicking on the Post entity and choosing Add | Association, as shown in Figure 7.

[6] I know – that's a whole lot of real estate.

[7] A 32-bit integer means that I can create a new blog post every minute of every day for 8,000 years. Hopefully I'll find something better to do with my time before I run out of unique post identifiers.

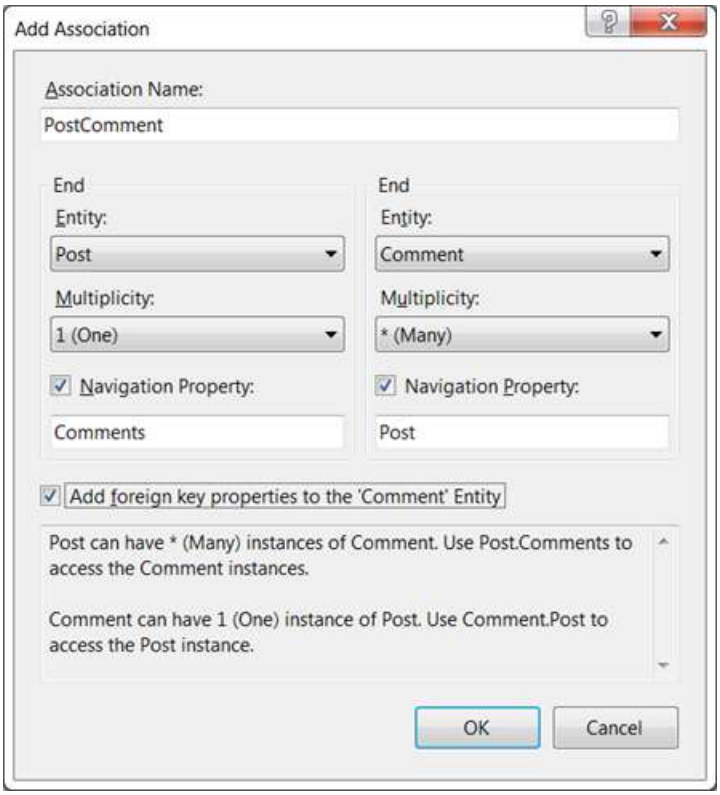


Figure 7: The Add Association dialog

What we're saying in Figure 7 is that we're associating the Post and the Comment entities and that for every post, we'll have zero or more comments. By leaving the "Navigation Property" checkbox checked (the default), you'll be adding properties that let you follow the link from each post to the associated comments and vice versa. And by leaving the "Add foreign key properties to the 'Comment' entity" option checked, you'll get a new property in the Comment entity which is the unique identifier of the associated post. When we map these entities to a relational database, you'll get a foreign key anyway so that the storage layer can do the mapping – we're just choosing whether we'd like to see it in our data model.

Pressing OK gives us an updated design surface showing the association, the navigation properties and the new PostId foreign key property in the Comment entity (Figure 8).

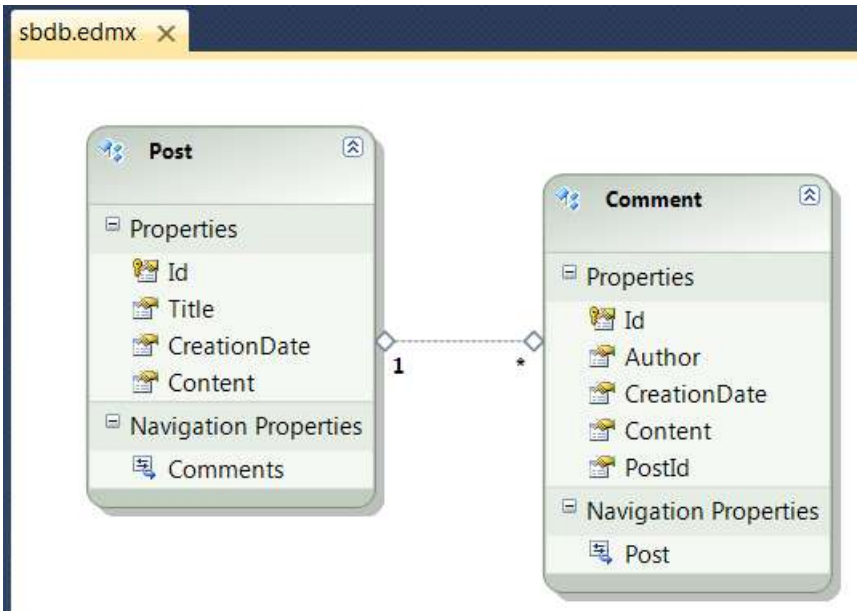


Figure 8:Adding an association in the Entity Designer

At this point, we've got enough of a data model to be dangerous. Let's get crazy.

Storing Our Data with SQL Server

One of things that makes our data model handy is that you can use it to generate a database to store values that conform to the schema for each entity. In fact, if you right-click on the Entity Designer, you'll see that one of the options is called "Generate Database from Model." Choosing that option gives you the Generate Database Wizard (Figure 9).

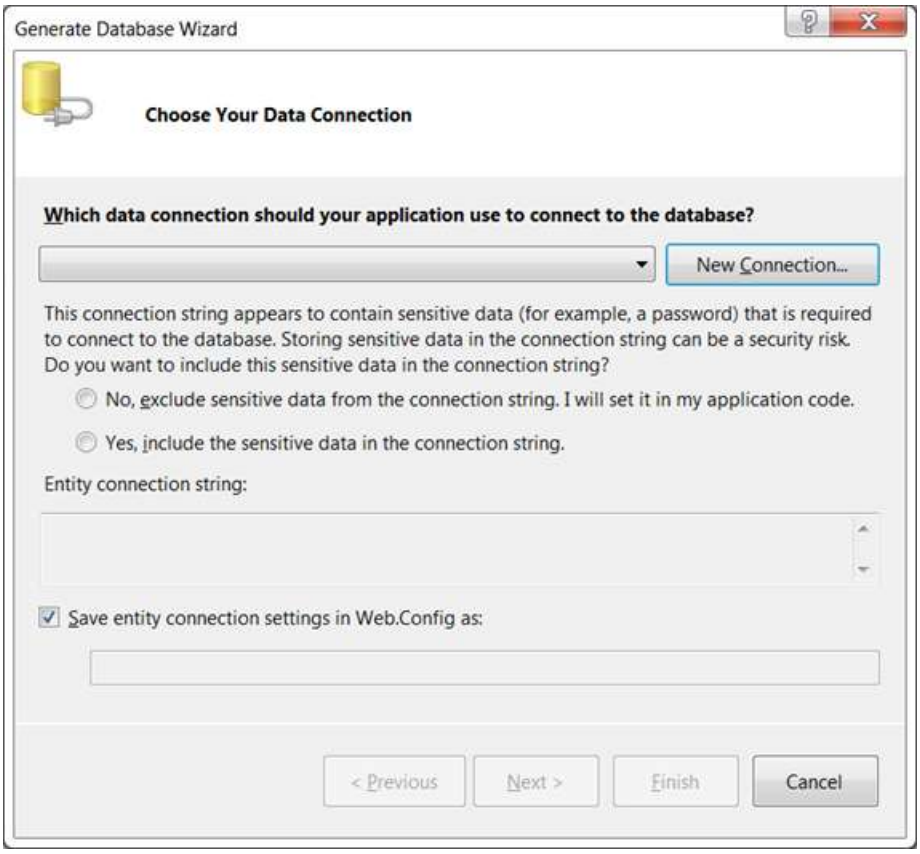


Figure 9: Generate Data Wizard dialog

The first thing you'll want to do is to press the New Connection button to create and connect to the database as shown in Figure 10.

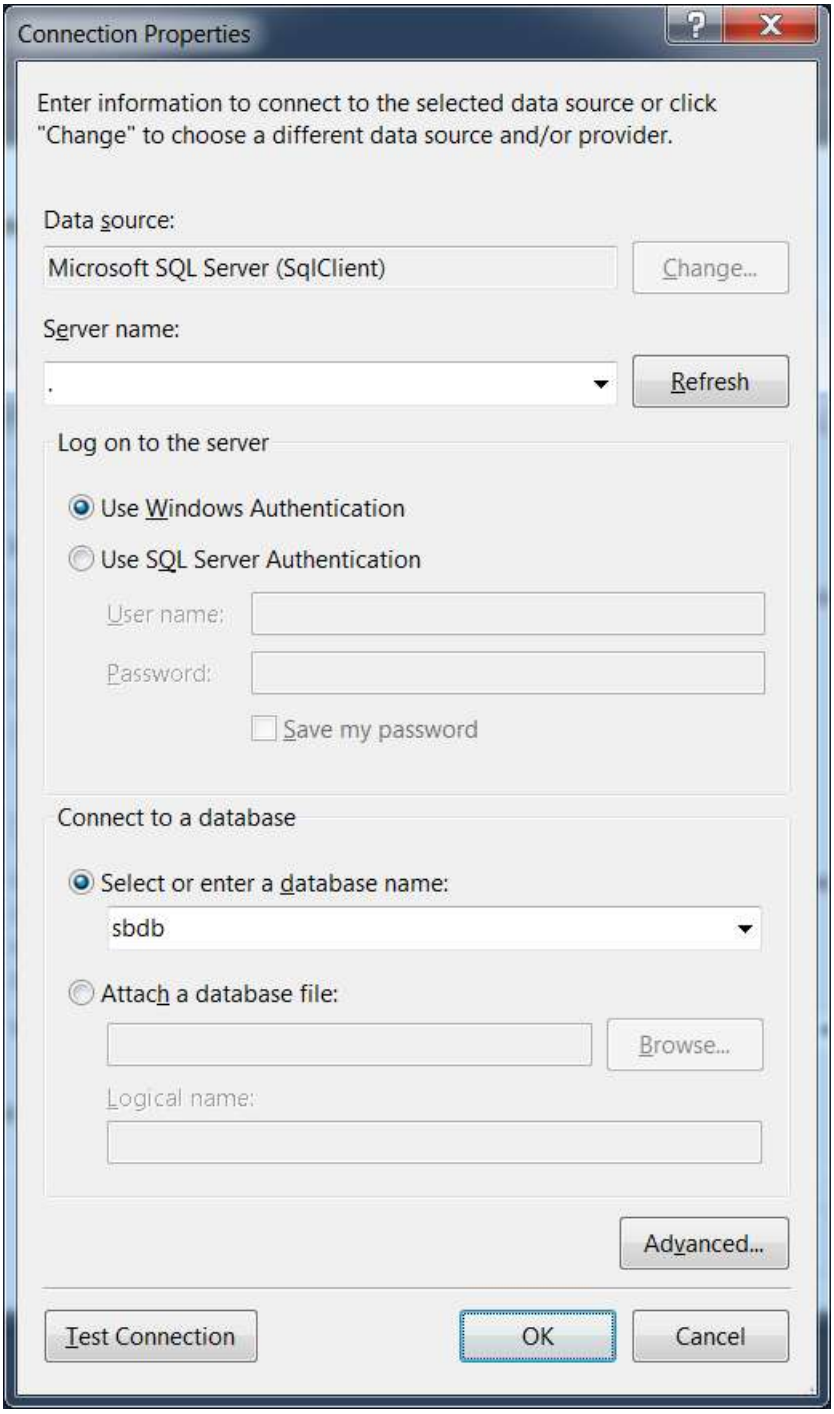


Figure 10: Connection Properties dialog

When you enter a database name for a database that doesn't yet exist, like we're doing in Figure 10, you're prompted to create that database. Here, the database we're creating is called "sbdb" and it's on the local machine (that's what the "." in the Server name field means). Pressing OK fills in the Generate Database Wizard with the right data and pressing Next actually shows you the SQL that's needed to generate the database ala Figure 11.

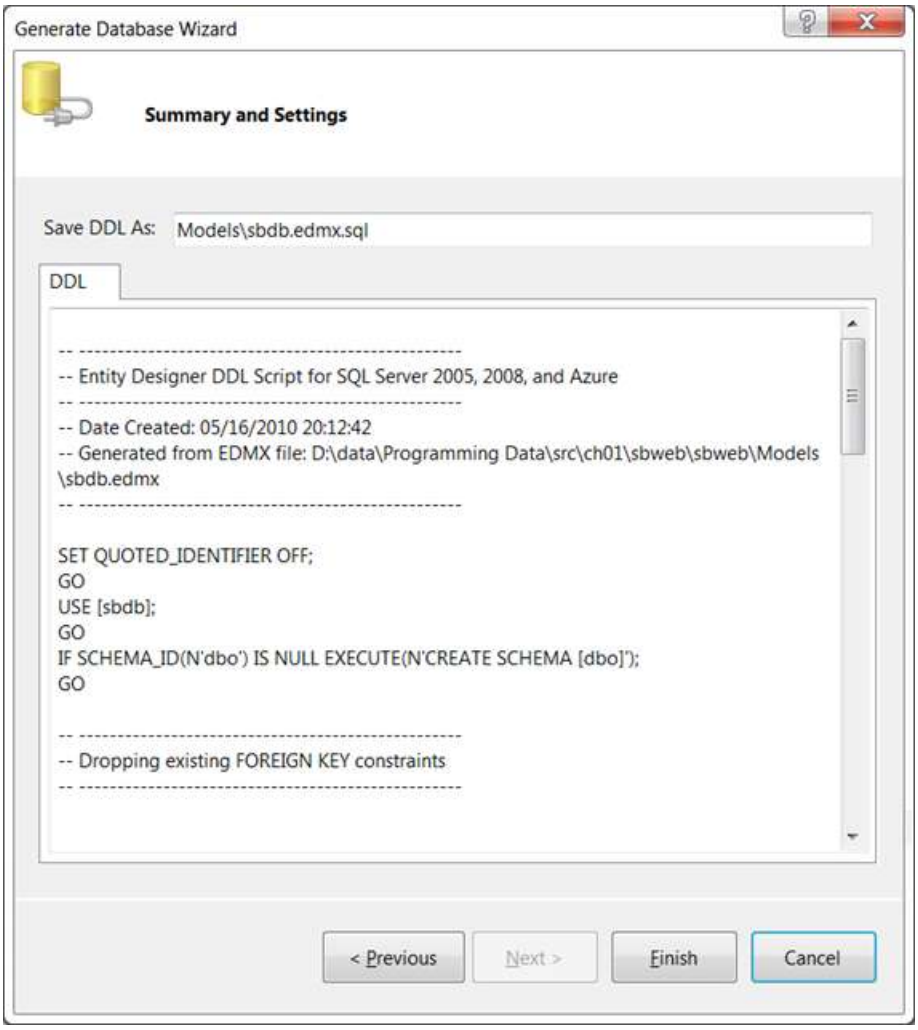


Figure 11: Generate Database Wizard summary dialog

Pressing the Finish button creates a .sql file and opens it. It should be pretty clear how the settings we made in the Entity Designer are represented in the generated SQL:

```
...
-- Creating table 'Posts'
CREATE TABLE [dbo].[Posts] (
    [Id] int IDENTITY(1,1) NOT NULL,
    [Title] nvarchar(128)  NOT NULL,
    [CreationDate] datetime  NOT NULL,
    [Content] nvarchar(max)  NOT NULL
);
GO

-- Creating table 'Comments'
CREATE TABLE [dbo].[Comments] (
    [Id] int IDENTITY(1,1) NOT NULL,
    [Author] nvarchar(128)  NOT NULL,
    [CreationDate] datetime  NOT NULL,
    [Content] nvarchar(max)  NOT NULL,
    [PostId] int  NOT NULL
```

```
);
GO
...
-- Creating primary key on [Id] in table 'Posts'
ALTER TABLE [dbo].[Posts]
ADD CONSTRAINT [PK_Posts] PRIMARY KEY CLUSTERED ([Id] ASC);
GO

-- Creating primary key on [Id] in table 'Comments'
ALTER TABLE [dbo].[Comments]
ADD CONSTRAINT [PK_Comments] PRIMARY KEY CLUSTERED ([Id] ASC);
GO
...
-- Creating foreign key on [PostId] in table 'Comments'
ALTER TABLE [dbo].[Comments]
ADD CONSTRAINT [FK_PostComment] FOREIGN KEY ([PostId]) REFERENCES [dbo].[Posts] ([Id])
    ON DELETE NO ACTION ON UPDATE NO ACTION;
...
```

For those that aren't familiar with SQL (Structured Query Language)[8], it's a standard for managing relational schemas and data. Here, we're creating two tables, Posts and Comments, both in the dbo[9] "schema" (which is roughly equivalent to a .NET namespace). The square brackets around all the names are to allow you to put special characters into them like spaces. Notice that the table names have been pluralized; that's a feature of the Entity Framework (EF), which is the set of .NET classes and tools that takes our conceptual Entity Data Model and maps it to a relational store (in this case, SQL Server 2008 R2 is what I'm running on my machine[10]). Notice also that the generated SQL knows about our Id properties by creating the primary key constraint and marking each one with the SQL "identity" keyword (which indicates that we're like inserts to create unique Id values for us). Also, notice that our association has turned into a foreign key constraint.

Given this generated SQL, we can execute it from within Visual Studio 2010 by right-clicking on it and choosing the Execute SQL option, which yields the Connect to Database Engine dialog, as shown in Figure 12.

[8] If you're not familiar with SQL, you should be. I recommend "Instant SQL Programming" by Joe Celko to get started. (http://tinysells.com/122)

[9] In the event that you cannot afford a SQL schema name, one will be provided for you. It will be called "dbo", which stands for "database objects." In general, it's better to pick one yourself.

[10] EF supports any store that comes with an EF provider. We'll be focusing on SQL Server in this book, but a sufficiently sophisticated EF provider should work as well.

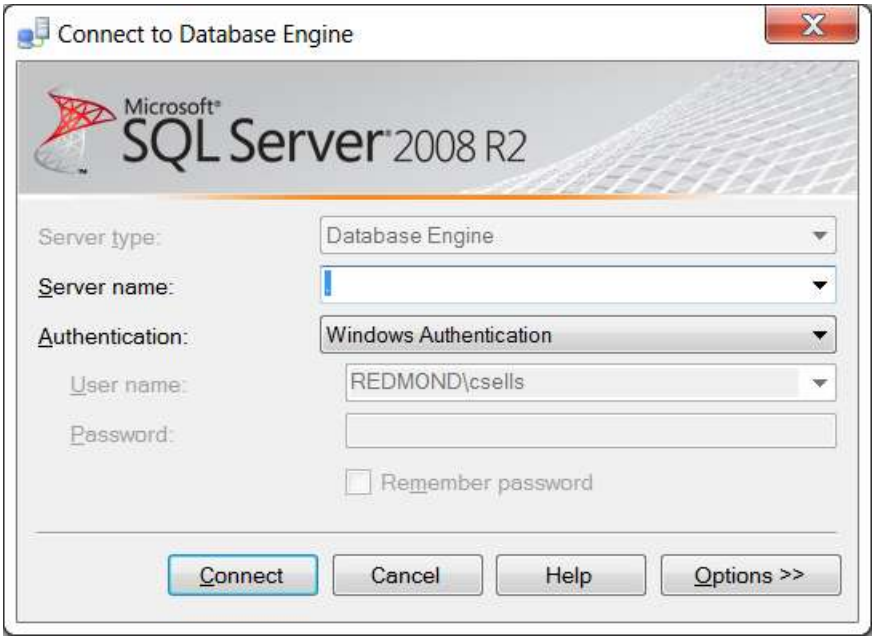


Figure 12:Connection to Database Engine dialog for executing SQL from within Visual Studio 2010

After the connection is made, the generated SQL is executed and voila' – instant database!

Managing Our Data with Visual Studio

To prove to ourselves that our database has been created, choose View | Server Explorer and you'll see our newly created entry under the Data Connections node of the tree. Drilling into that shows the two new tables we're expecting (Figure 13).

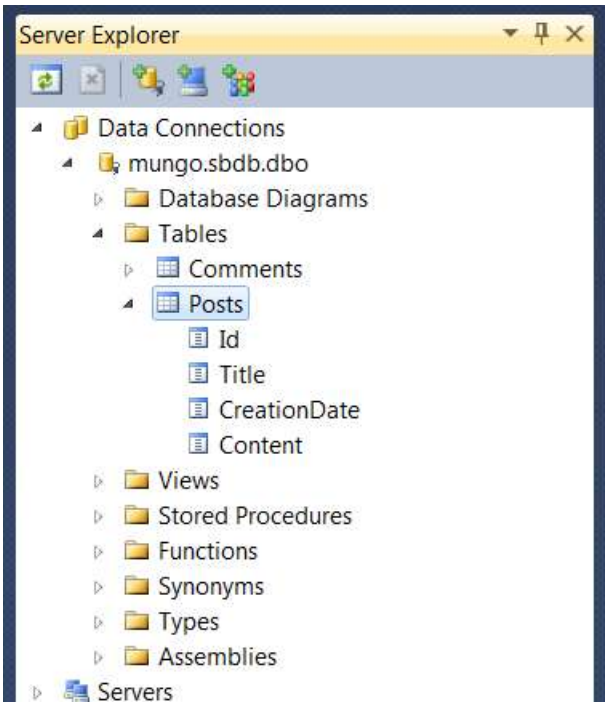


Figure 13: Showing the newly created database in the Server Explorer

Since we haven't yet entered any data, now would be a good time to do so by right-clicking on the Posts table and choosing Show Table Data. This gives you a read-write grid for you to enter data. It's nothing fancy, but it'll let you get started in a hurry when you need test data, e.g. Figure 14.

Posts: Query(mungo.sbdb)

	Id	Title	CreationDate	Content
	1	Data modeling rocks!	5/16/2010 12:00:00 AM	I'm a huge data modeling fan.
	2	SQL rocks!	5/17/2010 12:00:00 AM	A language for structured query? I'm in!
▸*	*NULL*	*NULL*	*NULL*	*NULL*

Figure 14: Show Table Data for the Posts table

Now that we've got some data, we can do data things with it. For a guided experience, you can right-click pretty much anywhere under your data connection node and choose New Query, which will give you a Query-By-Example helper. Or, if you're a fan of the world's most popular language for query over structured data, you can write SQL directly by simple creating a new SQL file (via the File | New | File menu) and executing it with a right-click on the SQL file | Execute SQL. Figure 15 shows an example.

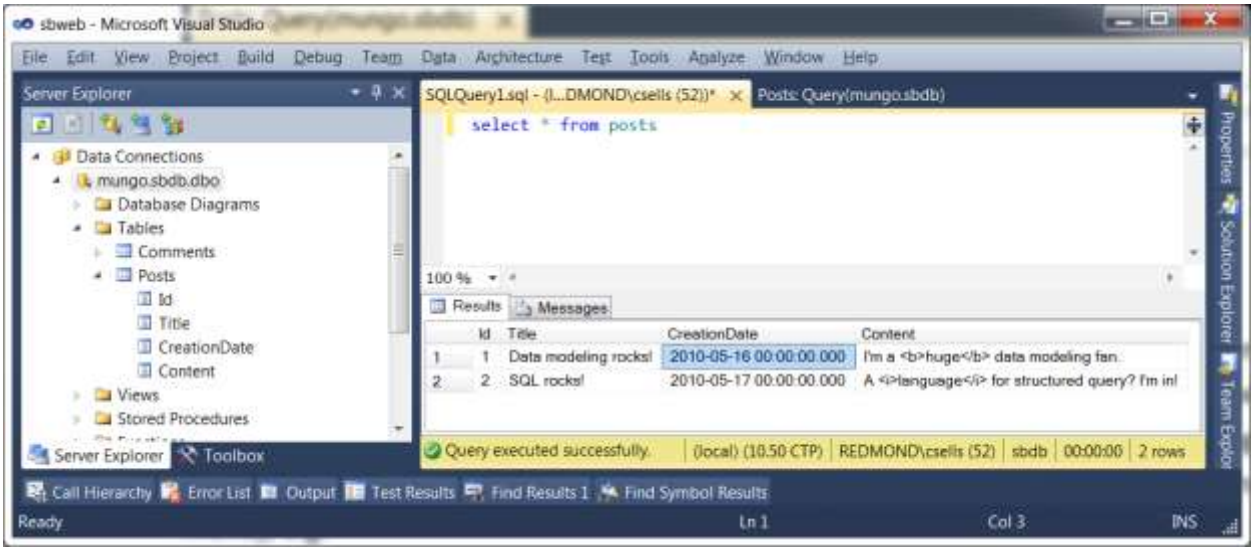


Figure 15: Executing a SQL statement from Visual Studio 2010

The beauty and wonder of being able to model your data, create the corresponding database, enter test data and write SQL against it all inside of Visual Studio is that it's very easy to build your database and your application at the same time. While you're doing that, however, you need to be careful with the

"Generate Database from Model" option if you expect to keep your test data. Unfortunately, it will not migrate the data to the new schema but rather flush it along with the old tables[11].

However, if you would like to migrate your data forward as your schema changes, and you don't faint at the sight of SQL, Visual Studio can help you with that, too. It performs this magic with Database Projects.

Database Projects

The idea of a Database Project is that, like your C# and Visual Basic-based applications and libraries, you'd like to keep your SQL code in a project environment with text-based editing tools, Intellisense, refactoring, a syntax-checking build, etc., all outside of the database. Then, when you're ready, you want to deploy, which in the case of a database means you need to be able to change the schema and migrate existing data forward. These are the kinds of problems that Database Projects were invented to solve.

If you create a new project in Visual Studio under the Database category, you'll see a set of SQL Server templates, as shown in Figure 16.

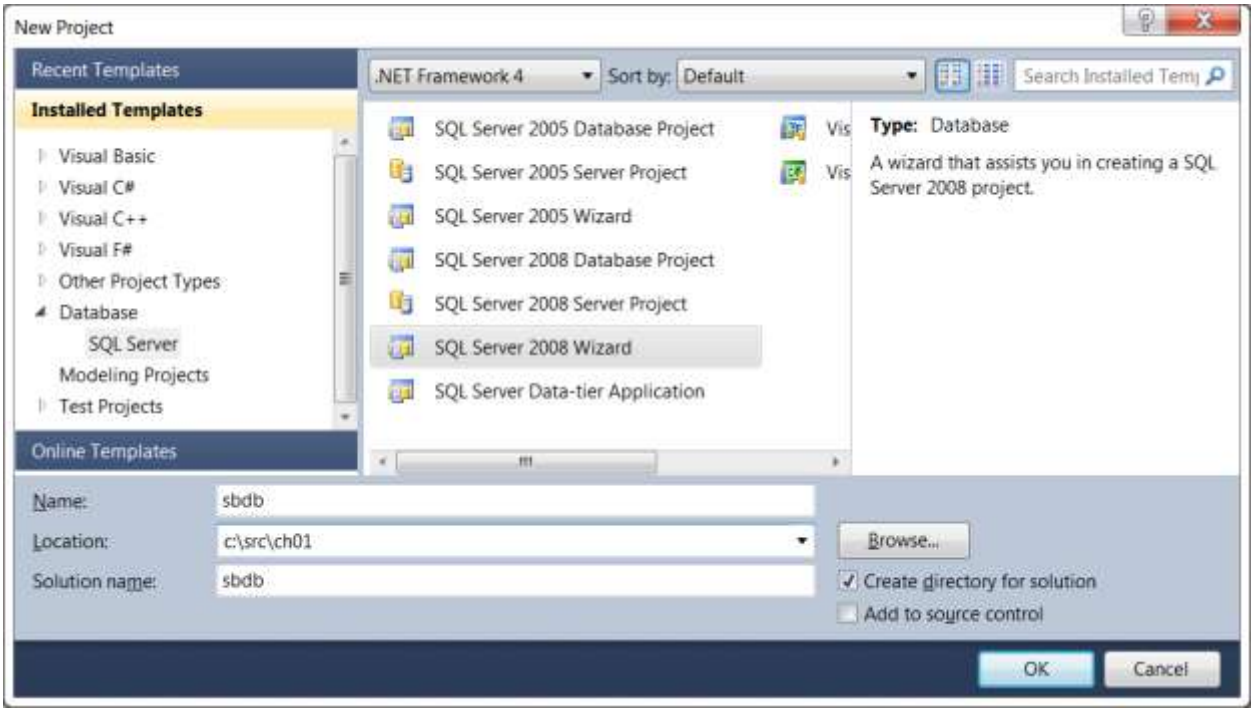


Figure 16: Choosing the SQL Server 2008 Wizard project template

Once we narrow it down to the edition of SQL Server we're using (2008 in our case), it looks like we've got three choices:

- **SQL Server 200x Database Project:** This is for managing a SQL Server "user" database, i.e. the thing we just created with our Posts and Comments tables in it.

[11] The Entity Designer Database Generation Power Pack is a tool that can migrate your data forward as you change your model: http://visualstudiogallery.msdn.microsoft.com/en-us/df3541c3-d833-4b65-b942-989e7ec74c87 (http://tinysells.com/123).

- **SQL Server 200x Server Project:** This is for managing server-level things in SQL Server, like server-wide logins and error messages. Most of the time you don't want this one.
- **SQL Server 200x Wizard:** This is just like the Database Project except you get a wizard for importing existing database schema information.

If you're going to build a database from scratch using SQL, the Database Project is what you want. If you're going to import existing schema, you can create a Database Project and then choose the option to import from an existing database or you can choose the Wizard project and you'll be lead through that process. Either way is fine, but you almost never want to choose a Server Project unless you're doing server-wide things that affect every database. That's not what we want.

What we want is the Wizard project, which brings up several loving screenfulls of options, none of which do we care about until we get to the "Import Database Schema" page, which lets us choose our existing sbdb database in SQL Server (Figure 17).

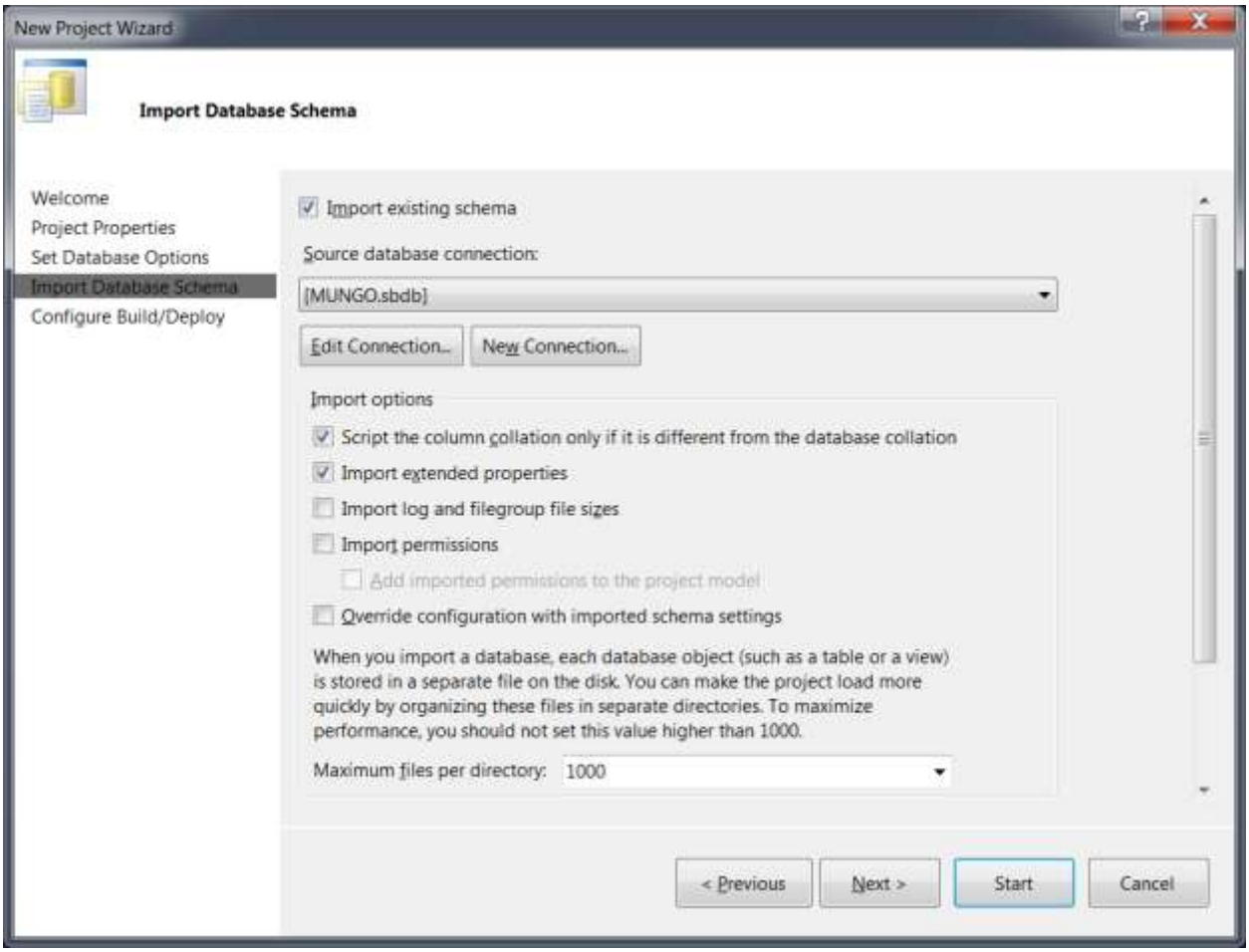


Figure 17: Importing a database schema into a new database project

Pressing the Finish button imports the database and we get a set of folders and files in the Solution Explorer that looks like Figure 18.

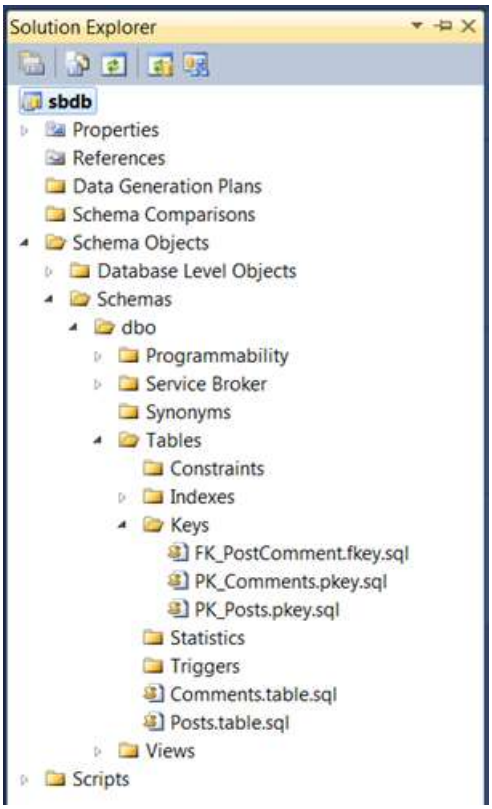


Figure 18: The SQL imported by the Database Project Wizard

The layout shown in Figure 18 relates to the folders and sub-folders on disk where Visual Studio looks for definitions of specific database objects in .sql scripts, one per file. These files are then pulled together at build time and the dependencies are arranged properly for us in a single SQL script that describes the SQL commands necessary to bring our database in line with the SQL in our project.

If you dig into the files that the wizard created for us, they should seem familiar:

```
-- Posts.table.sql
CREATE TABLE [dbo].[Posts] (
    [Id]           INT            IDENTITY (1, 1) NOT NULL,
    [Title]        NVARCHAR (128) NOT NULL,
    [CreationDate] DATETIME       NOT NULL,
    [Content]      NVARCHAR (MAX) NOT NULL
);


-- Comments.table.sql
CREATE TABLE [dbo].[Comments] (
    [Id]           INT            IDENTITY (1, 1) NOT NULL,
    [Author]       NVARCHAR (128) NOT NULL,
    [CreationDate] DATETIME       NOT NULL,
    [Content]      NVARCHAR (MAX) NOT NULL,
    [PostId]       INT            NOT NULL
);
```

You'll see that this is the exact same table schema that was generated for us by the Entity Design, but now split between two separate files. This is because the Database Project requires each database object to be split into its own file. If you have had fancier features like stored procedures, schemas or anything else, those would be in their proper folder and into their own file, like the pkey.sql and fkey.sql files shown in Figure 18.

Now that our schema has been imported into a project, we can do obvious things like check the files into source code control, edit those files and, most importantly, build the project, which will check that our SQL is valid after we make changes without committing anything to the database. For example, if we wanted to change our Comments table name to something more pompous like CriticalObservation, we could edit the file and then check to make sure everything's OK with Build | Build Solution. In this case, changing the Comments table name causes several errors:

```
SQL03006: Primary Key: [dbo].[PK_Comments] has an unresolved reference to object [dbo].[Comments].
C:\src\ch01\sbdb\sbdb\Schema Objects\Schemas\dbo\Tables\Keys\PK_Comments.pkey.sql
SQL03006: Primary Key: [dbo].[PK_Comments] has an unresolved reference to Column [dbo].[Comments].[Id].
C:\src\ch01\sbdb\sbdb\Schema Objects\Schemas\dbo\Tables\Keys\PK_Comments.pkey.sql
...
```

Our problem here, besides our high fallootin' table names, is that the Posts table references Comments, which means that our change in the file that defines the table must be followed up with changes to all of the files that reference that table, e.g. all stored procedure definitions, all indexes, all keys, etc.

To solve this problem with a language like C#, which has support for refactoring, you'd right-click on the name of a type from the text editor, choose Refactor | Rename, then get a chance to preview all of the files that will be updated based on the change to the name. In a Database project, you can do the same thing to the name of a database object from the Schema View[12], which you can get to via the View | Database Schema View menu item, which shows our little sample look like Figure 19.

[12] Unfortunately, the 2010 version of Visual Studio does not support refactoring from the SQL text editor. Keep your fingers crossed that future versions will do so.

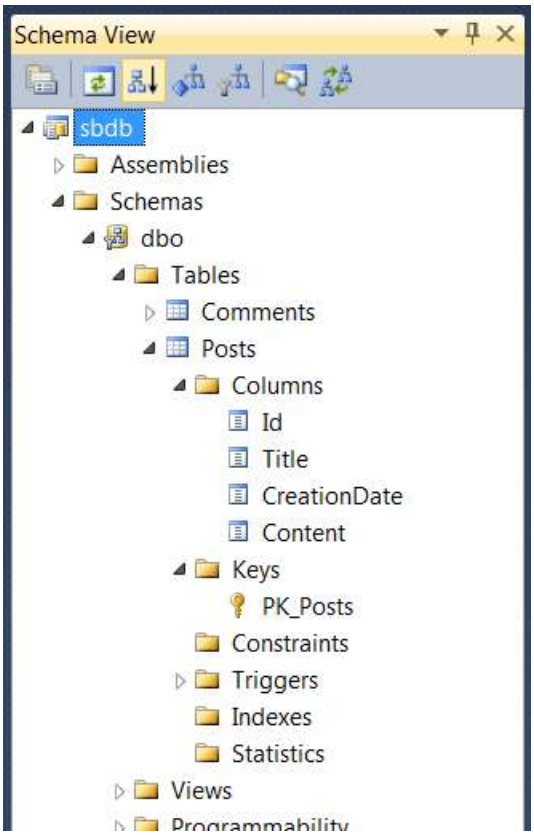


Figure 19: The Database Schema View

The Schema View provides the same kind of logical view of the objects in your database like the one you'd see in the Object Browser for .NET types. If you'd like to rename the Comments table, you can do so by right-clicking on the Comments table and choosing Refactor | Rename. By default, entering a new name and pressing OK gives you the preview window just like in C#, as shown in Figure 20.

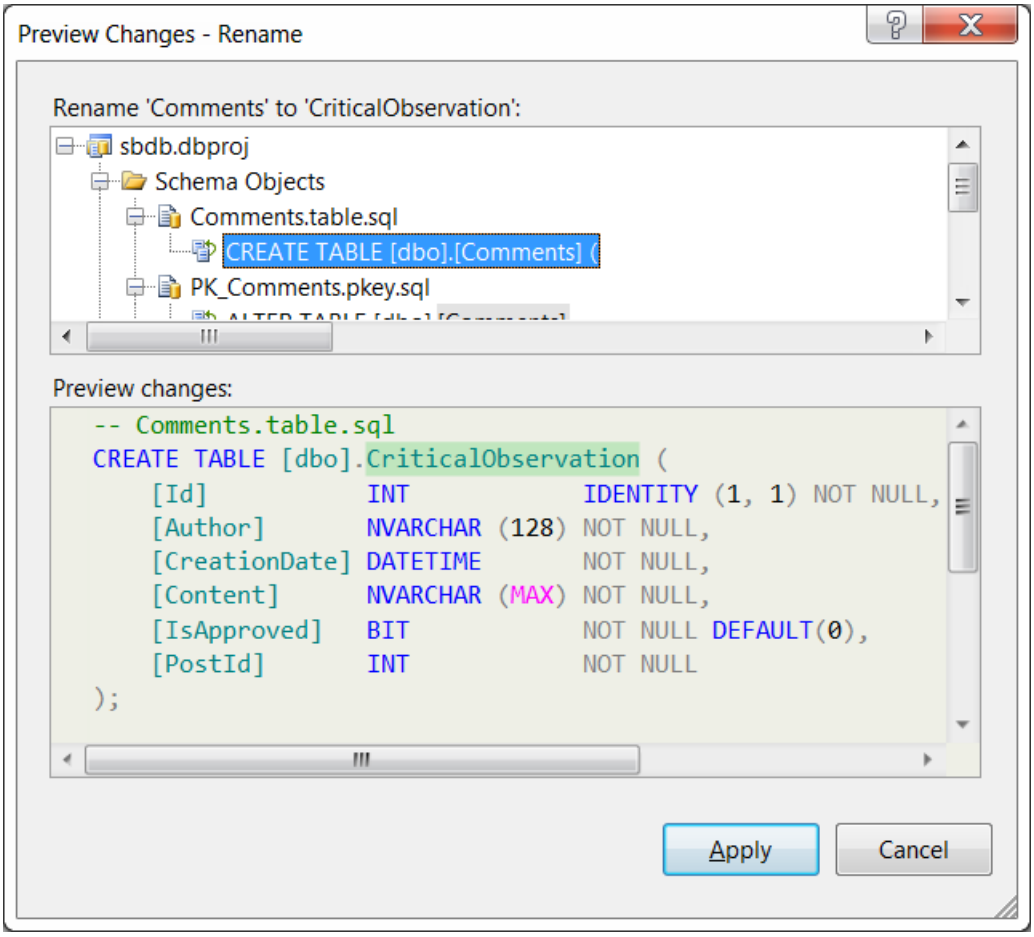


Figure 20: Previewing the changes for a rename refactoring operation

Pressing Apply would do the renaming and you could test that it was successful by building and enjoying the dearth of errors. We're not going to do that, however, because CriticalObservation is a terrible name.

Something we will do, however, is add a column to our Comments table. For example, in the big, wide world, some people's comments are inappropriate. And by that, I don't mean that they contain profanity or differ from my opinion. I encourage both of those. No, of course, I refer to the dreaded "comment spam," which plagues blogs the world over. There are all manner of ways to deal with the detection of unbridled, unwanted marketing in the content of my web site, but whatever means I choose, I want to be able to mark a comment as either approved or not so that I can know whether to include it in the HTML that makes up my web site. Opening the Comments.table.sql file and adding an IsApproved column is just as easy as it sounds:

```
-- Comments.table.sql
CREATE TABLE [dbo].[Comments] (
    [Id]           INT            IDENTITY (1, 1) NOT NULL,
    [Author]       NVARCHAR (128) NOT NULL,
    [CreationDate] DATETIME       NOT NULL,
    [Content]      NVARCHAR (MAX) NOT NULL,
```

```
    [IsApproved]    BIT             NOT NULL DEFAULT(1),
    [PostId]        INT             NOT NULL
);
```

Just to make sure I've typed my SQL correctly, I can build my Database Project and notice the lack of errors. You can do that with all manner of SQL constructs and Visual Studio will check the syntax and relations of all of them without ever involving the database.

Deploying Database Changes

When it's time to involve the database, you can see the complete SQL script to create your database that VS creates for you by right-clicking on the database project in the Solution Explorer and choosing Deploy. By default, this process will create a <<ProjectName>>.sql file[13] that in our case looks like the SQL we'd expect:

```
PRINT N'Creating [dbo].[Comments]...';
GO
CREATE TABLE [dbo].[Comments] (
    [Id]           INT             IDENTITY (1, 1) NOT NULL,
    [Author]       NVARCHAR (128)  NOT NULL,
    [CreationDate] DATETIME        NOT NULL,
    [Content]      NVARCHAR (MAX)  NOT NULL,
    [IsApproved]   BIT             NOT NULL,
    [PostId]       INT             NOT NULL
);
GO
...
PRINT N'Creating [dbo].[Posts]...';
GO
CREATE TABLE [dbo].[Posts] (
    [Id]           INT             IDENTITY (1, 1) NOT NULL,
    [Title]        NVARCHAR (128)  NOT NULL,
    [CreationDate] DATETIME        NOT NULL,
    [Content]      NVARCHAR (MAX)  NOT NULL
);
GO
...
```

Except for the change we made, i.e. the new column, you'll notice that the SQL looks what we've already seen. In fact, because of that, you don't want to execute this script at all. It won't migrate the Comments data forward with the new IsApproved column.

Instead, what we need is a somewhat more complicated script to do the following:

- Create a transaction.
- Create a new table with a new name.
- Insert all of the old Comments data into the new table.
- Drop the old table.
- Rename the new table to Comments.
- Commit the transaction.

[13] You can check the Output window for the specific path to the generated SQL deployment file.

The reason we have to do it this way is so that we can manage things like preserving the existing identity column values and handling new columns that are not null but that have defaults (like our IsApproved column). Of course, we'd have to do this kind of thing for every construct that changed, which makes building scripts to manage it all very tedious and error prone for humans. Luckily, Visual Studio will do all of this for us. All we have to do is tell it what database to compare against and it will find the differences between that database and our project's SQL so that it can generate the correct SQL to update the schema to match and migrate the data along the way.

To get started, bring up the Project properties by right-clicking on the project in the Solution Explorer and choosing Properties, then choose the Deploy tab. You want to set the Target Database settings, which start as empty, to what you see in Figure 21.

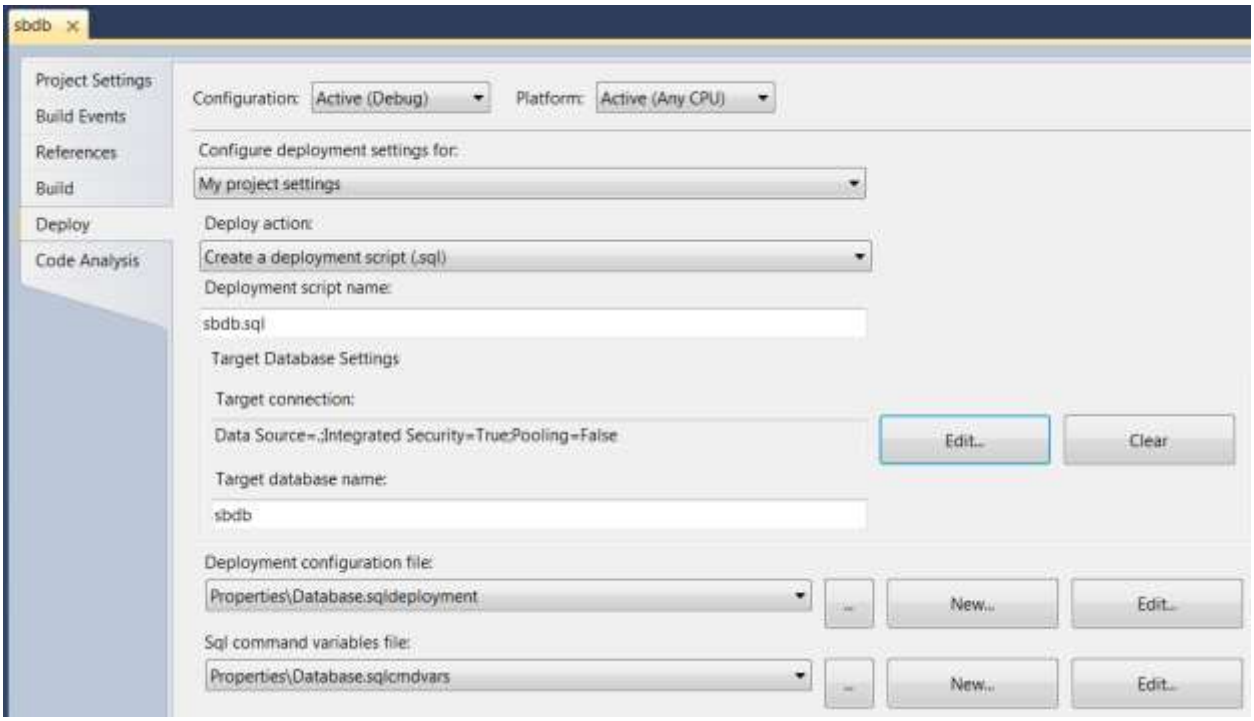


Figure 21: Setting the target database settings for deployment

By pressing the Edit button next to the Target connection field and filling in another Connection Properties dialog, we've specified our "sbdb" database. Notice also that the Deploy action is set to "Create a deployment script (.sql)". That means when we choose Deploy this time, we'll see a very different script:

```
...
BEGIN TRANSACTION;

CREATE TABLE [dbo].[tmp_ms_xx_Comments] (
    [Id]           INT            IDENTITY (1, 1) NOT NULL,
    [Author]       NVARCHAR (128) NOT NULL,
    [CreationDate] DATETIME       NOT NULL,
    [Content]      NVARCHAR (MAX) NOT NULL,
    [IsApproved]   BIT            DEFAULT (0) NOT NULL,
```

```
    [PostId]       INT            NOT NULL
);

ALTER TABLE [dbo].[tmp_ms_xx_Comments]
    ADD CONSTRAINT [tmp_ms_xx_clusteredindex_PK_Comments] PRIMARY KEY CLUSTERED ([Id]
ASC) WITH (ALLOW_PAGE_LOCKS = ON, ALLOW_ROW_LOCKS = ON, PAD_INDEX = OFF, IGNORE_DUP_KEY =
OFF, STATISTICS_NORECOMPUTE = OFF);

IF EXISTS (SELECT TOP 1 1
           FROM   [dbo].[Comments])
    BEGIN
        SET IDENTITY_INSERT [dbo].[tmp_ms_xx_Comments] ON;
        INSERT INTO [dbo].[tmp_ms_xx_Comments] ([Id], [Author], [CreationDate],
[Content], [PostId])
        SELECT   [Id],
                 [Author],
                 [CreationDate],
                 [Content],
                 [PostId]
        FROM     [dbo].[Comments]
        ORDER BY [Id] ASC;
        SET IDENTITY_INSERT [dbo].[tmp_ms_xx_Comments] OFF;
    END

DROP TABLE [dbo].[Comments];

EXECUTE sp_rename N'[dbo].[tmp_ms_xx_Comments]', N'Comments';

EXECUTE sp_rename N'[dbo].[tmp_ms_xx_clusteredindex_PK_Comments]', N'PK_Comments',
N'OBJECT';

COMMIT TRANSACTION;
...
```

Visual Studio has gone to our existing sbdb database and noticed that the Comments table in our project is different, so creates the script to migrate the existing table definition to our new one and keep the existing data.

At this point, with the migration script in hand, you can provide it to the DBA and have him/her run it in whatever test/validation/don't-trust-developers environment they like or, if you've got the permission, you can go back to the Deploy project settings page and change the Deploy action from "Create a deployment script (.sql)" to "Create a deployment script (.sql) and deploy to database".

With that option set, when you deploy your database project, the target database will be differenced to create the migration script and then the script will be run, which you can see in the Output window:

```
------ Build started: Project: sbdb, Configuration: Debug Any CPU ------
    Loading project files...
    Building the project model and resolving object interdependencies...
    Validating the project model...
    Writing model to sbdb.dbschema...
  sbdb -> C:\src\ch01\sbdb\sbdb\sql\debug\sbdb.dbschema
------ Deploy started: Project: sbdb, Configuration: Debug Any CPU ------
    Deployment script generated to:
C:\src\ch01\sbdb\sbdb\sql\debug\sbdb.sql
```

```
========== Build: 1 succeeded or up-to-date, 0 failed, 0 skipped ==========
========== Deploy: 1 succeeded, 0 failed, 0 skipped ==========
```

To convince yourself that the migration has been done properly, you can pull your database up again in the Server Explorer (right-clicking on your database and choosing Refresh if necessary) to see the changes (Figure 22).

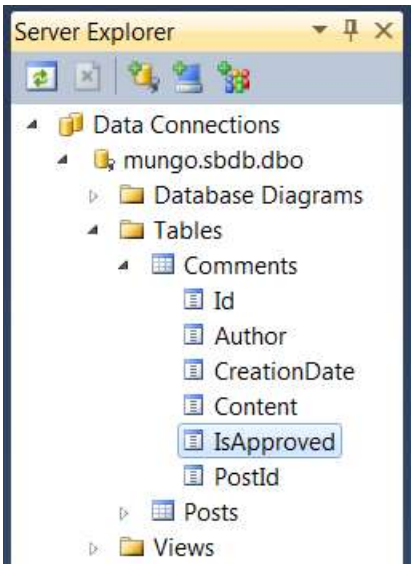


Figure 22: Showing the table updated by the deployment SQL script

Now that we've got our database just the way we like it, we can get back to building our application.

Accessing Our Data with the Entity Framework

Thus far we've been using the Entity Framework for defining the schema of our database. However, if you open up the "sbweb" project again, you'll notice a file called "sbdb.Designer.cs" under the "sbdb.edmx" file. It's driven by the entities defined in the data model and contains the C# (or VB) code necessary to access the database, e.g.

```
...
namespace sbweb.Models {
    ...
    public partial class sbdbContainer : ObjectContext {
        ...
        public sbdbContainer() : base("name=sbdbContainer", "sbdbContainer") {...}
        ...
        public ObjectSet<Post> Posts { get {...} }
        ...
        public ObjectSet<Comment> Comments { get {...} }
        ...
    }
    ...
    public partial class Comment : EntityObject {
        ...
```

```
        [EdmScalarPropertyAttribute(EntityKeyProperty=true, IsNullable=false)]
        [DataMemberAttribute()]
        public global::System.Int32 Id { get {...} set {...} }
        ...
        [EdmScalarPropertyAttribute(EntityKeyProperty=false, IsNullable=false)]
        [DataMemberAttribute()]
        public global::System.String Author { get {...} set {...} }
        ...
        [EdmScalarPropertyAttribute(EntityKeyProperty=false, IsNullable=false)]
        [DataMemberAttribute()]
        public global::System.DateTime CreationDate { get {...} set {...} }
        ...
        [EdmScalarPropertyAttribute(EntityKeyProperty=false, IsNullable=false)]
        [DataMemberAttribute()]
        public global::System.String Content { get {...} set {...} }
        ...
        [EdmScalarPropertyAttribute(EntityKeyProperty=false, IsNullable=false)]
        [DataMemberAttribute()]
        public global::System.Int32 PostId { get {...} set {...} }
        ...
        [XmlIgnoreAttribute()]
        [SoapIgnoreAttribute()]
        [DataMemberAttribute()]
        [EdmRelationshipNavigationPropertyAttribute("sbdb", "PostComment", "Post")]
        public Post Post { get {...} set {...} }
        ...
    }

    // Post is similarly constructed
    public partial class Post : EntityObject { ... }
}
```

There are several interesting things to notice about this automatically generated code. The first is the sbdbContainer, known as the “context” class, which keeps the context associated with a set of entity data. The context object caches the data as it’s pulled down and keeps track of changes made to that data so it send just the updates back to the database as requested. It has two collection properties, one for each “entity set,” that is, each collection of entity objects we have in our model, Posts and Comments.

The second thing to notice about this code is the generated Comment and Post classes, each of which represent an instance of an entity as serialized from the database. Each property on the entities maps to a property on the class, which in turn maps to one or more columns in the rows from our database. You can see this mapping by right-clicking on an entity in the designer and choosing Table Mapping, which will show the Mapping Details window, as you can see in Figure 23.

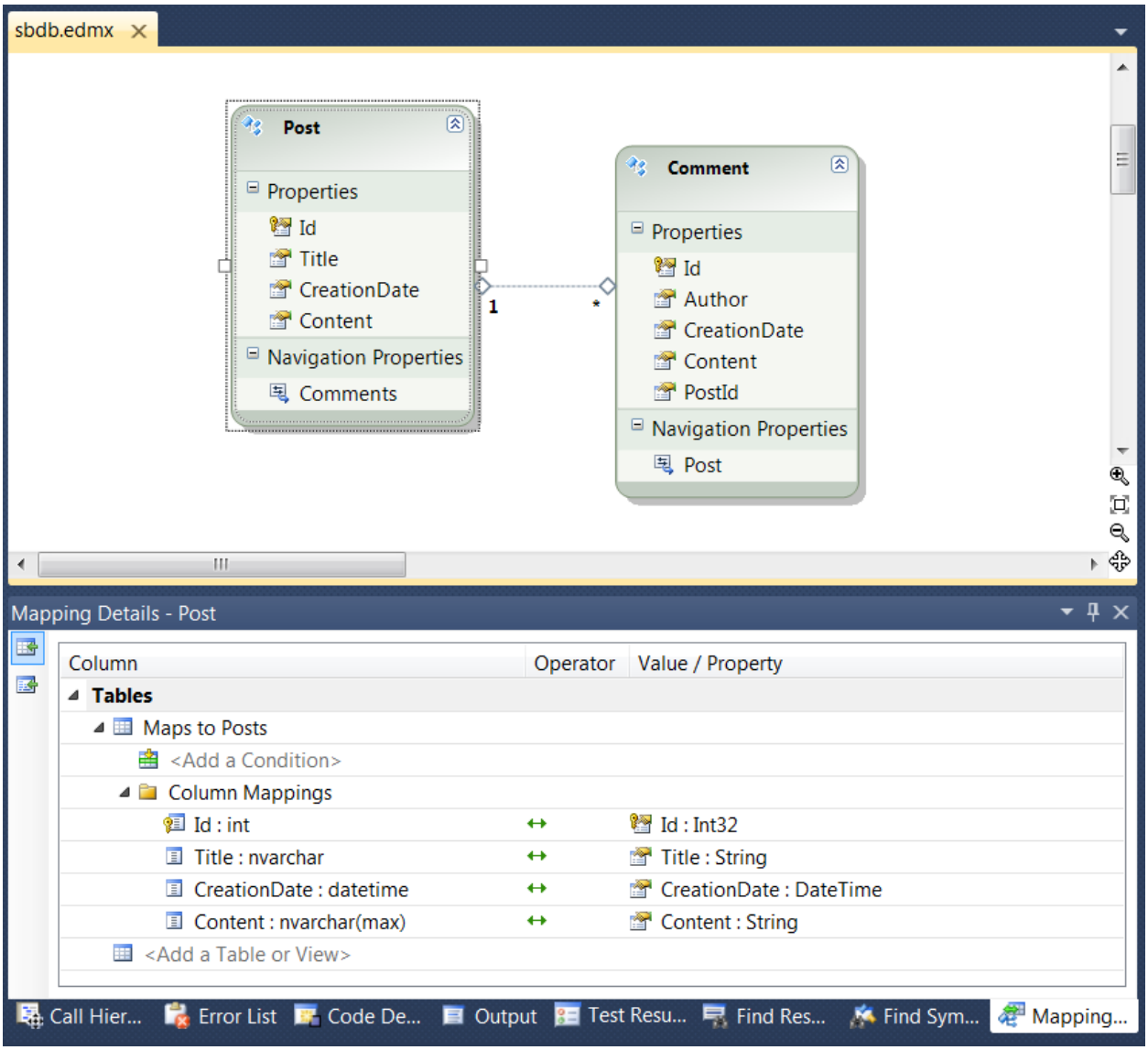


Figure 23: The Mapping Details view

Right now, as you can see, we have a one-to-one mapping between entities in our data model and tables, as well as a one-to-one mapping between properties and columns, but that doesn't need to be the case. One of the real benefits of EF is that the mapping doesn't have to be one-to-one at all, which you can read about in Chapter 2: Entity Framework.

The third and final thing to notice about this code is that it's wrong: there is no IsApproved property. That's because in the previous section we added the IsApproved column to the Comments table in the database, but our model is now out of sync with our database. To solve this problem, right-click on the designer surface and choose "Update Model from Database", which will give you the Update Wizard as shown in Figure 24.

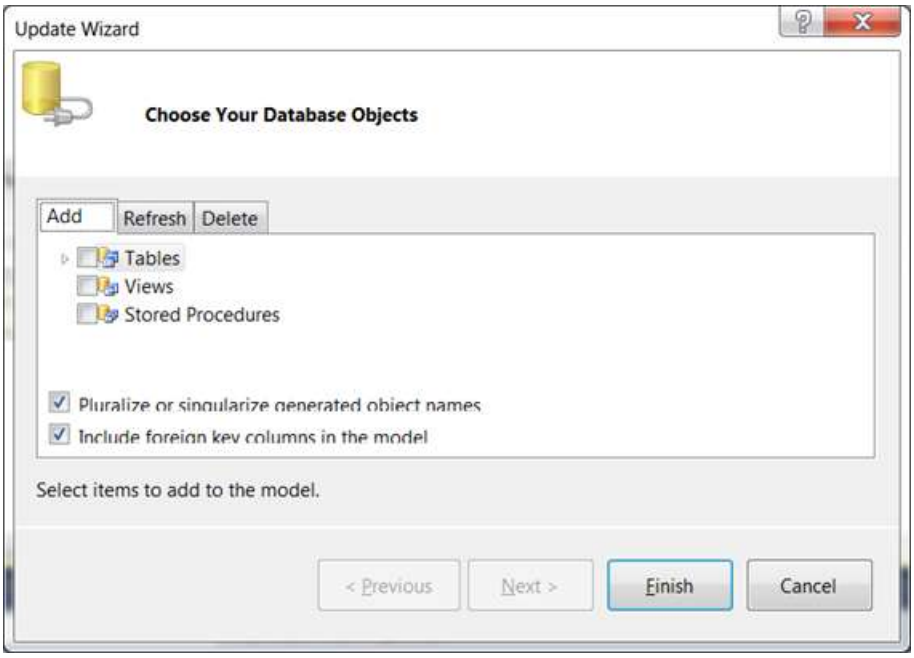


Figure 24: The Update Wizard

On the Add tab, you'll find a list of tables, views and stored procedures that are in the database but that aren't in the model. On the other hand, the Refresh and Delete tabs show the existing tables, views and stored procedures in the model that can be refreshed or deleted from our model. We want the default, which is to refresh everything and to add and delete nothing, so pressing Finish updates our Comment entity (Figure 25).

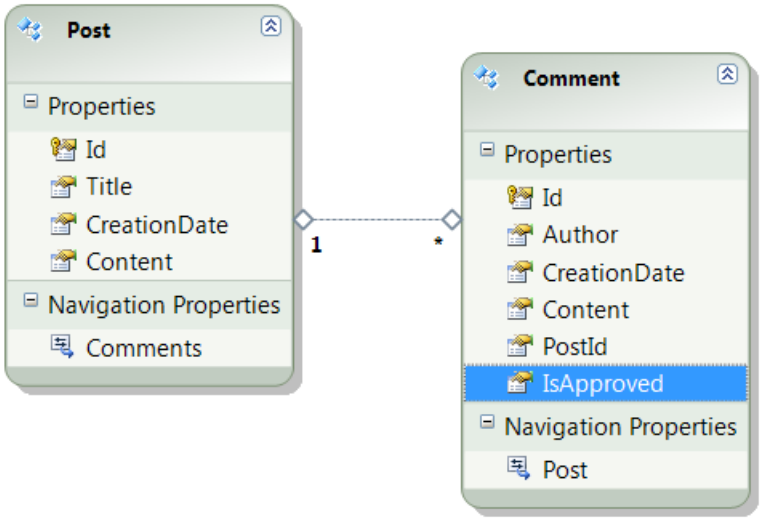


Figure 25: The updated entities

You can check the generated EF code if you like to verify that it's been fixed or we can just go write our application code. I'd prefer the latter and it's my book, so…

Inside the Controllers folder in the Solution Explorer, you'll find the HomeController.cs file. Open it and update the Index method like so:

```
using System.Linq;
using System.Web.Mvc;

namespace sbweb.Controllers {
  [HandleError]
  public class HomeController : Controller {
    public ActionResult Index() {
      // Create the context, grab the newest posts and pass them to the view
      var context = new sbweb.Models.sbdbContainer();
      var newestPosts = context.Posts.OrderByDescending(p=>p.CreationDate).Take(25);
      return View(newestPosts);
    }

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

A "Controller," in standard MVC parlance, is the thing that takes commands from the View and does something useful, generally by doing work on the "Model" (which is just the data) before handing the data off to the "View" to be displayed. The View then takes input from the user, passes it along to the Controller and the circle of life continues.

In our case, the HomeController that a new MVC 2 project gives us out of the box doesn't do anything useful except serve as a placeholder for our functionality. We're updating it here to create a new instance of the context class generated from our Entity Data Model (sbdb.edmx). Accessing the Posts property gives as access to an ObjectSet<Post>, which is the Entity Framework's implementation of IQueryable. It's the IQueryable implementation that translates the OrderByDescending and Take method (and tons of others) into SQL statements. In other words, instead of EF executing a "select * from Posts" and then doing the sort on CreationDate and trimming off the top 25 in memory, IQueryable allows EF to translate the whole thing into a single SQL statement[14], like so:

```
SELECT TOP (25)
[Extent1].[Id] AS [Id],
[Extent1].[Title] AS [Title],
[Extent1].[CreationDate] AS [CreationDate],
[Extent1].[Content] AS [Content]
FROM [dbo].[Posts] AS [Extent1]
ORDER BY [Extent1].[CreationDate] DESC
```

The beauty of this scheme is that I can write C# in my programs and know that it'll be turned into efficient SQL to reduce round-trips to my database. Win-win.

[14] You can see exactly what EF is sending to your SQL Server database using the SQL Server Profiler.

The only other interesting line of code is the return, which finds the view to create based on convention. By default, it'll look for the Index.aspx view under the Views\Home folder in the Solution Explorer. We need to update this view to get it to show our posts:

```
<%@ Page Language="C#" MasterPageFile="~/Views/Shared/Site.Master"
Inherits="System.Web.Mvc.ViewPage<IEnumerable<sbweb.Models.Post>>" %>

<asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">
    Home Page
</asp:Content>

<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
  <% foreach (var post in Model) { %>
      <h1><%= Html.Encode(post.Title) %></h1>
      <p><%= post.Content %></p>
      <p><i><%= String.Format("{0:f}", post.CreationDate) %></i></p>
  <% } %>
</asp:Content>
```

First, because we're creating a view with the result of our query, which is an ObjectSet<Post>, we need a view that takes a type that's compatible with that. We could use ObjectSet<Post> as our type parameter to ViewPage in the @Page directive, but instead we use IEnumerable<Post>. The reason is that we don't want our view performing any more round-trips to the database or modifying the collection, so by passing in a bare naked IEnumerable, the view gets just what it needs[15].

The second thing we need to do is to output the data itself in HTML format. We do that inside the content element with a simple foreach, iterating over each post from the Model variable, which is populated with whatever we passed from the controller when we created the view (the collection of Post objects in our case). Because we set the template parameter to ViewPage, as you type "post." in the editor, Intellisense will show you the properties we originally specified in our data model, e.g. Title, Content and CreationDate. The results of these simple changes yield the beginnings of a new sellsbrothers.com (Figure 26).

[15] The "Bear Necessities" if you will.

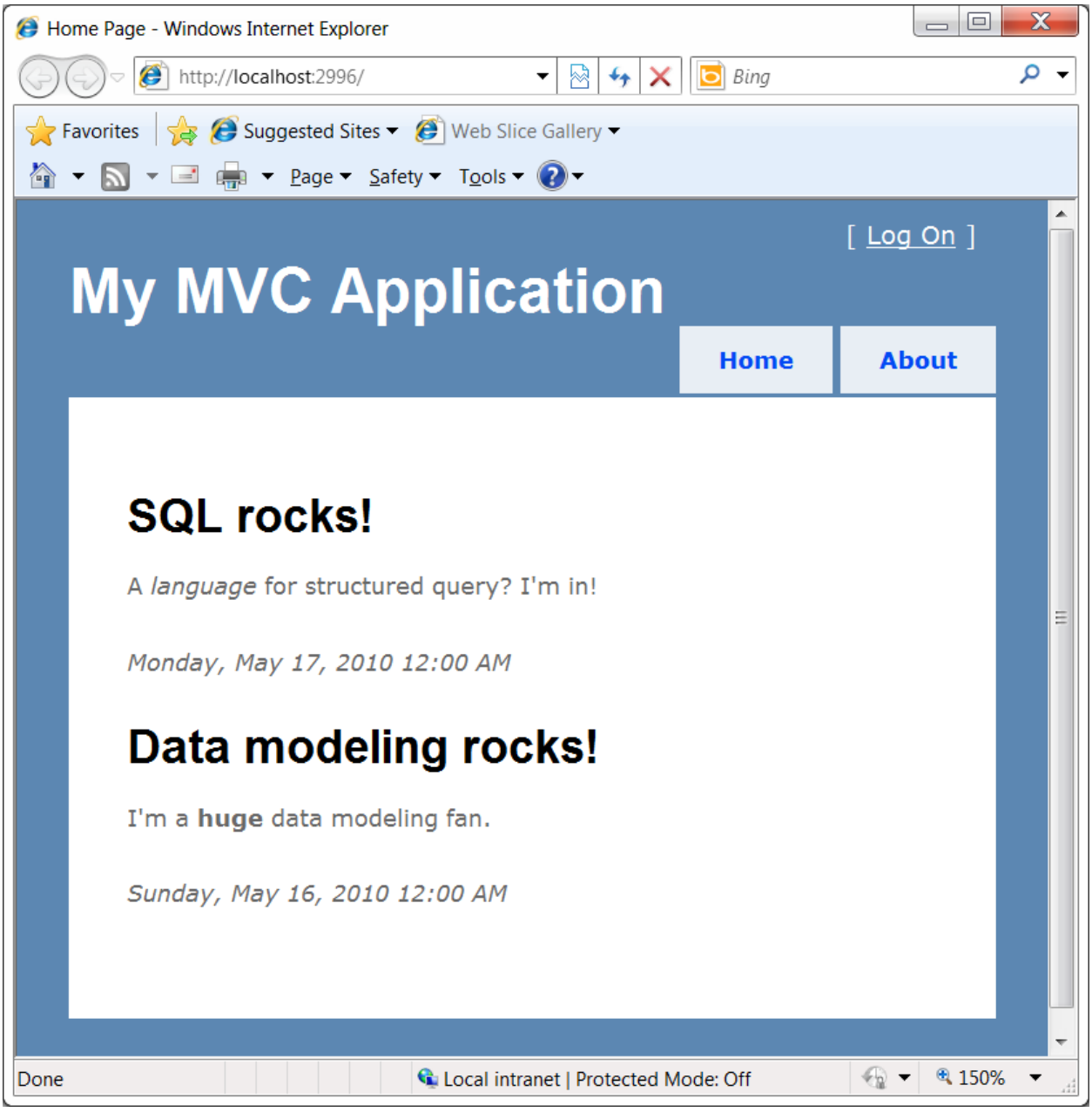


Figure 26: The running sample MVC application

To review, so far we've described the shape of our data in the Entity Designer, created the database to hold the data and added test data from Visual Studio, refactored the database to better suit our needs using a database project, updated the EF-generated code from the database schema and accessed the data using Entity Framework so that we could display it in HTML via ASP.NET. The results aren't stunning, but it's not too hard to see how we'd link to comments, pretty things up, etc. now that we have the basics in place[16].

[16] In fact, the new version of sellsbrothers.com started from exactly this beginning.

Exposing Our Data with the Open Data Protocol

However, with the basics done, we need to talk about how we get real content into the database. The table editor in Visual Studio isn't great for more than test data. Maybe I want to build myself a rich client application to handle my blog musings. To do that, I'll need programmatic access to the data, but once I have my database published to my hosting company, I may not have direct access to the SQL database. Or, even if I did have such access, maybe I want to expose the raw data from sellsbrothers.com to others in some form that they could use to write their own programs. As nice as it is for viewing by humans, HTML is not good for programmers looking for more than a blob of content, ads, header graphics and menu items.

One common solution to this problem is to build myself a custom web service that exposes web site content for programmatic access. In fact, there are several existing blogging APIs defined as web services for just this purpose, e.g. the Metaweblog API. And these APIs are great, but limited to that one purpose – managing blog posts. What if I also want to do ad hoc queries against the blog posts to see which ones have the most comments or track status and figure out which have the most readers? What I'd really like is the ability to do arbitrary queries as well as managing my blog posts, perhaps through a flexible, standardized protocol with a robust implementation provided in .NET. And for that, we have WCF Data Services and the Open Data Protocol.

Exposing OData with Data Services

Windows Communication Foundation Data Services (or just Data Services for short) provides the ability to expose your database over HTTP using the same techniques you used to map a database into your .NET application: the Entity Data Model. In fact, if you start with the ASP.NET MVC 2 sample application we just built using a .edmx file, you're two-thirds of the way there. The last step is to right-click on the web project in the Solution Explorer, choose Add | New Item, then choose WCF Data Service, as shown in Figure 27.

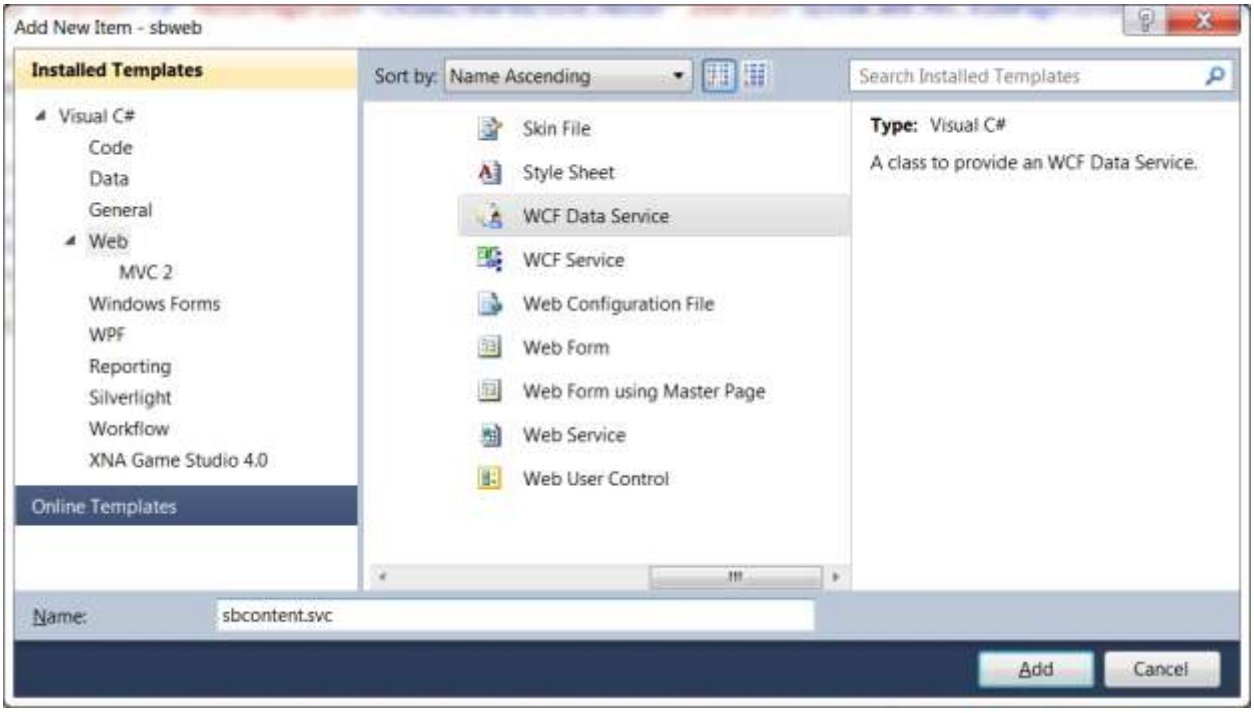


Figure 27: Adding a WCF Data Service endpoint

After pressing the Add button, you'll be presented with a Data Services file (*.svc) that only requires you to add the name of the entities class in the template parameter to the DataService class and choose which parts of the EDM you'd like to expose and how. Here we're exposing our sample entities and giving full read-write access to them:

```
using System.Data.Services;
using System.Data.Services.Common;

namespace sbweb.Views.Home {
  public class sbcontent : DataService<sbweb.Models.sbdbContainer> {
    public static void InitializeService(DataServiceConfiguration config) {
      // Giving the world read-write access to our data w/o authentication is a bad idea!
      // See Chapter 3: Data Services for the right way to do things
      config.SetEntitySetAccessRule("Posts", EntitySetRights.All);
      config.SetEntitySetAccessRule("Comments", EntitySetRights.All);

      config.DataServiceBehavior.MaxProtocolVersion = DataServiceProtocolVersion.V2;
    }
  }
}
```

The DataService base class provides the mapping of the functionality of the entities class generated by EF from the .edmx file to the core verbs of the HTTP protocol, specifically GET, POST, PUT and DELETE, which map to the common data CRUD operations (Create, Read, Update and Delete). This read-write mapping is based on the philosophy of REST (REpresentational State Transfer), which is itself a handy means to expose programmatic functionality across the web in a way that takes advantage of the world-wide infrastructure to serve resources efficiently and securely.

Accessing OData from the Browser

With this small file in place, Data Services has all the information it needs to expose your database as modeled in the .edmx file. To see how, start your web project running and surf to the .svc file. You'll see your list of entities, as Figure 28 shows.

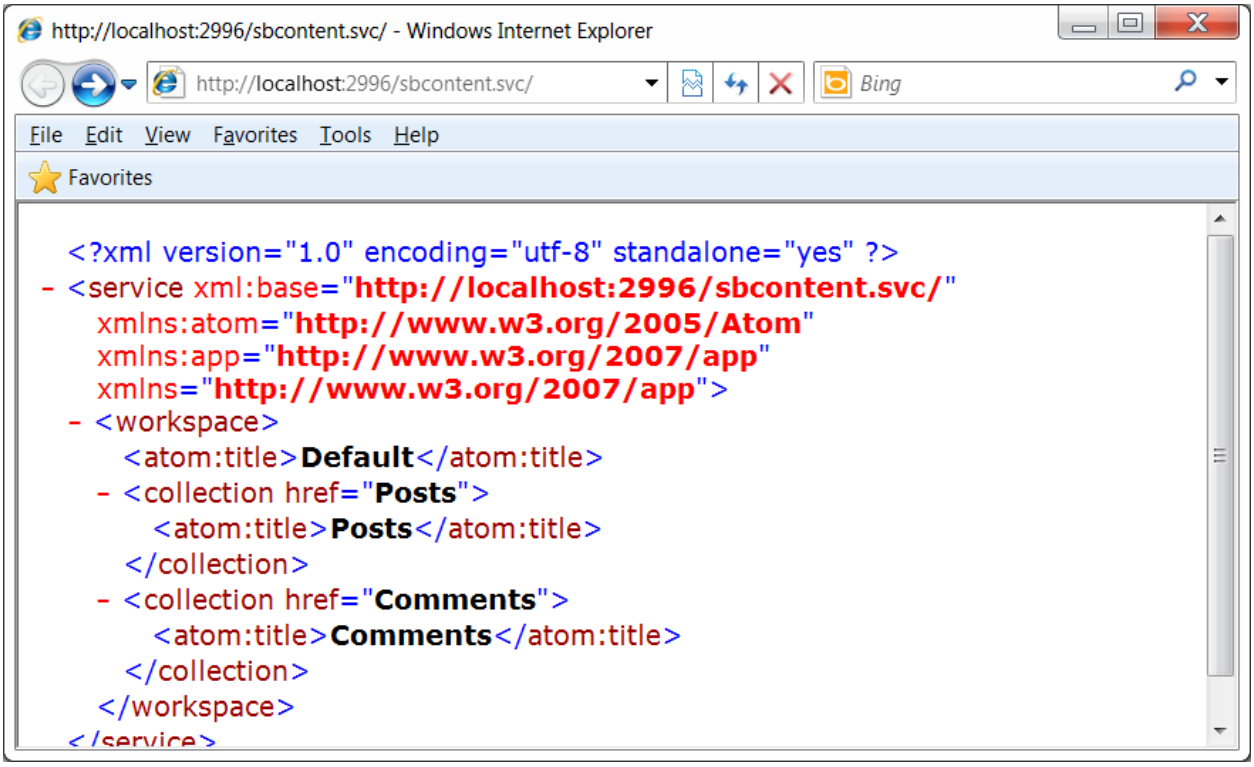


Figure 28: The top-level OData service document from our sample

Drilling down one level further, you can see our Posts (Figure 29).

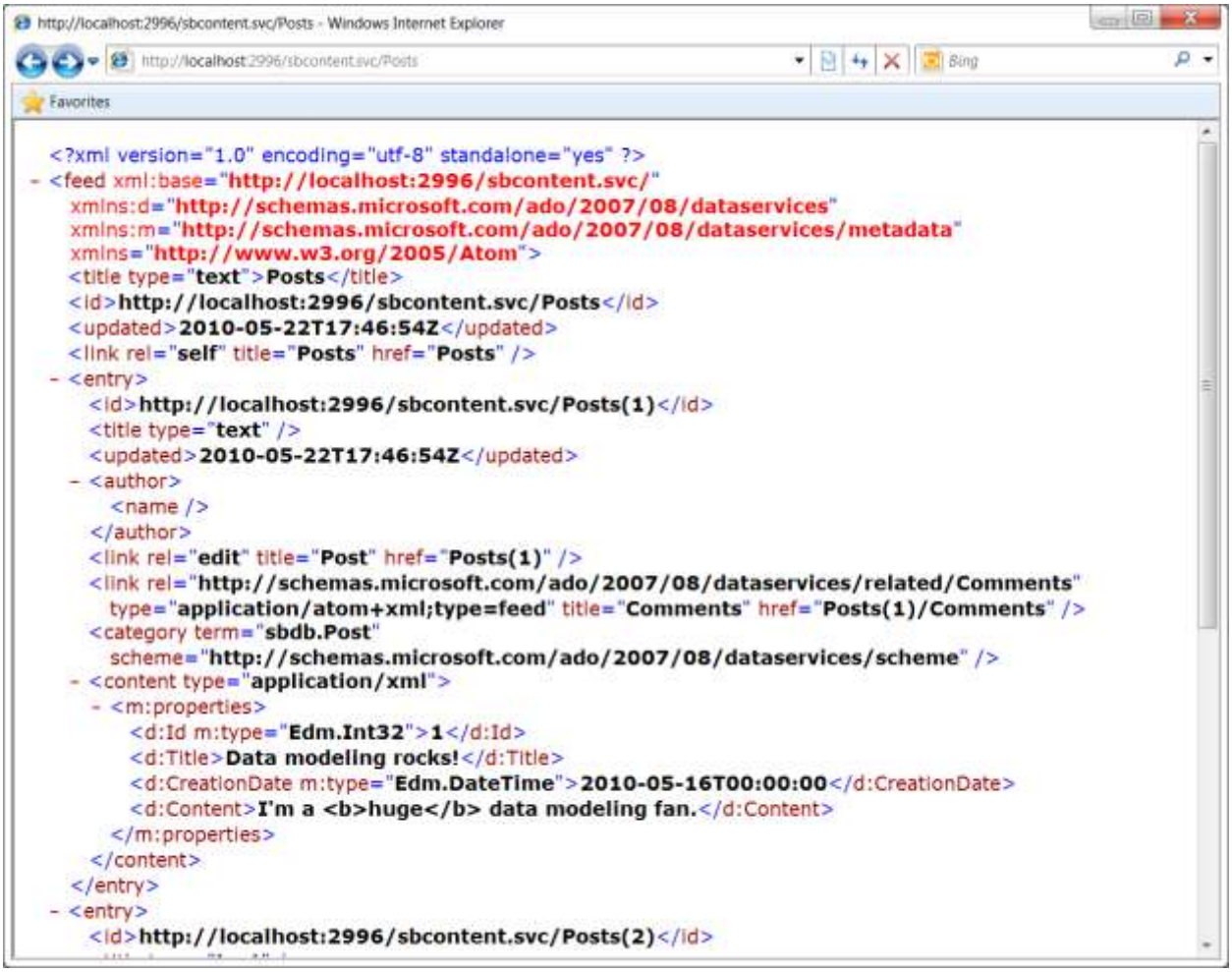


Figure 29: Exploring the Posts entities via OData

The data you're seeing in Figure 29 is formatted according to the Open Data Protocol (OData)[17], which was developed at Microsoft to leverage the IETF standard Atom Publishing Protocol (RFC 5023), itself based on the Atom Syndication Format (RCF 4287). These protocols are cross-platform. In fact, any platform that supports HTTP and XML is enough to support OData.

However, to make programming more convenient than composing and parsing HTTP and XML messages by hand, there are several client platforms with optimized support for consuming OData, including .NET, Silverlight, AJAX, Java, PHP and Excel (via PowerPivot). Further, there are several service-side frameworks that support OData include .NET, SQL Azure, SharePoint and IBM's WebSphere, with more coming all the time. What all of this means to you is that if you use Data Services to expose your data, it can literally be consumed securely[18] from practically any platform or device on the planet. And you won't be the only one[19]; we've got a wave of data available in OData, with enables rich metadata, a

[17] http://odata.org

[18] OData relies on the same security mechanisms that are already used in web applications, e.g. HTTPS, certificates, etc.

[19] You can see the growing list of OData services listed on http:// odata.org/producers.

powerful query language (expressed in the URL itself) and a full set of client and server-side libraries for consuming it and producing it.

Accessing OData from .NET

Now that we've exposed our data via OData, we can consume it from .NET as well. To demonstrate this, I built myself an editing environment for posts that is slightly better than the table editor in VS. I started by creating a new WPF Application project laying out the WPF as shown in Figure 30.

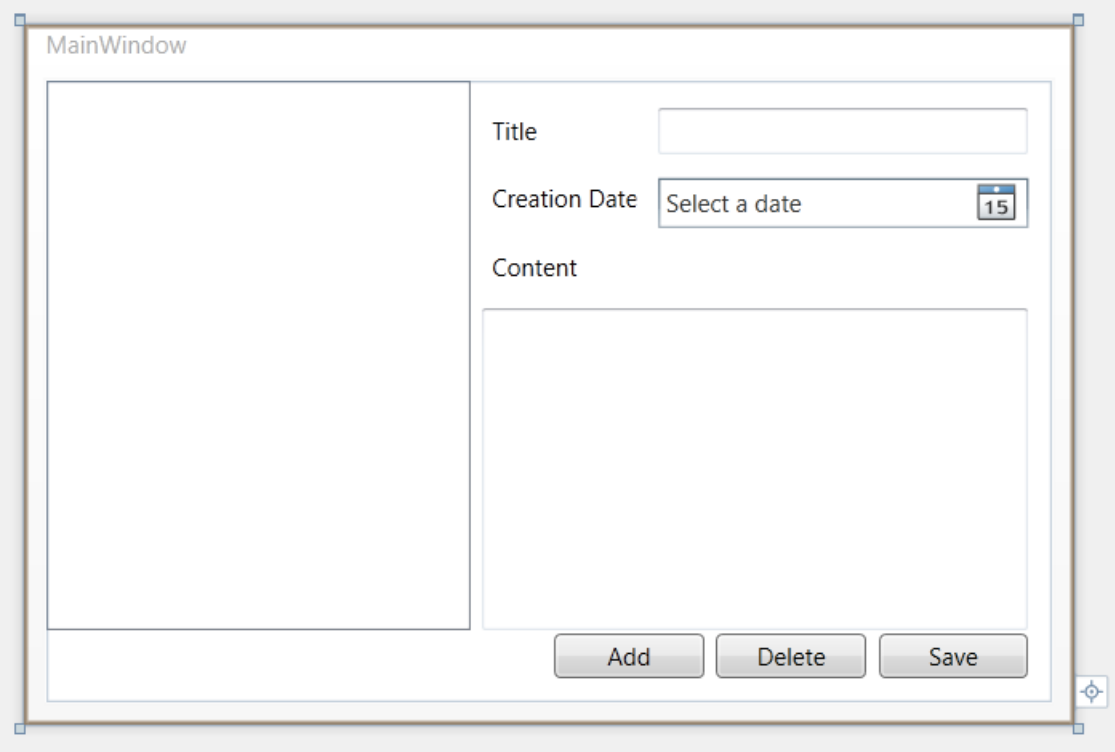


Figure 30: Laying out the sample editor application in the WPF Designer

The XAML[20] is pretty simple:

```
<Window Title="MainWindow" Height="350" Width="525" ...>
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="275*" />
            <RowDefinition Height="36*" />
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="212*" />
            <ColumnDefinition Width="291*" />
        </Grid.ColumnDefinitions>

        <ListBox DisplayMemberPath="Title" ItemsSource="{Binding}"
                 IsSynchronizedWithCurrentItem="True" />

        <Label Content="Title" ... />
```

[20] XAML is the layout language for WPF windows. Think HTML for desktop applications.

```
        <Label Content="Creation Date" ... />
        <Label Content="Content" ... />
        <TextBox Text="{Binding Path=Title}" ... />
        <DatePicker Grid.Column="1" SelectedDate="{Binding Path=CreationDate}" ... />
        <TextBox Text="{Binding Path=Content}" ... />
        <Button Content="Add" Click="addButton_Click" ... />
        <Button Content="Delete" Click="deleteButton_Click" ... />
        <Button Content="Save" Click="saveButton_Click" ... />
    </Grid>
</Window>
```

The only thing interesting in this code is the use of data binding to bring in the data. To make the data available to the program, we need to create a Data Services proxy, which we can do by right-clicking on the project in the Solution Explorer and choosing Add Service Reference, which brings up the dialog in Figure 31.

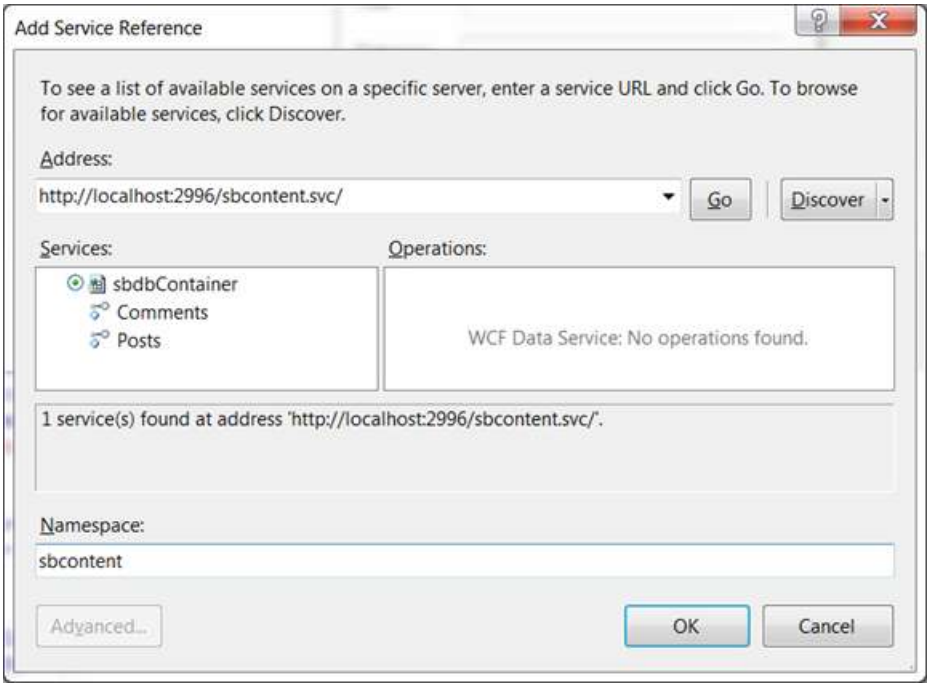


Figure 31: The Add Service Reference dialog

You'll notice that I put the URL to the OData endpoint we just built into the dialog and, after I pressed Go, now Visual Studio can read the metadata that describes our endpoint. I also set the namespace to something meaningful and pressed OK.

When I did that, I got myself a set of types very much like what the Entity Designer gave me, one for the context and one for each of the entities. Using these, I can implement my little post editor:

```
using System;
using System.Data.Services.Client;
using System.Linq;
using System.Windows;
using System.Windows.Data;
using sbedit.sbcontent; // from our generated service reference
```

```
namespace sbedit {
  public partial class MainWindow : Window {
    sbdbContainer context =
      new sbdbContainer(new Uri(@"http://localhost:2996/sbcontent.svc/"));
    DataServiceCollection<Post> posts;

    public MainWindow() {
      InitializeComponent();
      posts =
        new DataServiceCollection<Post>(
          context.Posts.OrderByDescending(p => p.CreationDate).Take(25));
      this.DataContext = posts;
    }

    void addButton_Click(object sender, RoutedEventArgs e) {
      var newPost = new Post() {
        Title = "New Post",
        CreationDate = DateTime.Now,
        Content = "your content here"
      };
      posts.Add(newPost);
      CollectionViewSource.GetDefaultView(this.DataContext).MoveCurrentTo(newPost);
    }

    void deleteButton_Click(object sender, RoutedEventArgs e) {
      var currentPost =
        (Post)CollectionViewSource.GetDefaultView(this.DataContext).CurrentItem;
      posts.Remove(currentPost);
    }

    void saveButton_Click(object sender, RoutedEventArgs e) {
      context.SaveChanges();
      MessageBox.Show("Changes saved.", "SB Editor");
    }

  }
}
```

First and foremost, notice the creation of the context object, constructed with the URL to our service endpoint. This exposes the Posts and Comments collections, very much like how the EF context works.

Second, notice that we're composing a query over our Posts collection using the same constructs we were using before. This will be reflected in the URL that gets sent to the OData endpoint.

Next, notice the use of the DataServiceCollection<> type to wrap the query over our Posts collection. The DataServiceCollection<> type adds change notification tracking to support WPF data binding. Without it, you'll be tracking changes yourself, which very much defeats the purpose of data binding in

the first place. By placing it into the main window's DataContext property, everything is sharing the same set of data against which to be bound[21].

The Add button's Click event handler pulls the posts collection back out of the window's data context and adds a new post. The Delete button finds the currently selected post and removes it from the posts collection.

And finally, the Save button's Click event handler takes all of the changes that have been made against the local context and pushes them back to the database.

With this in place, our post editor works just the way we'd like it to, as shown in Figure 32.

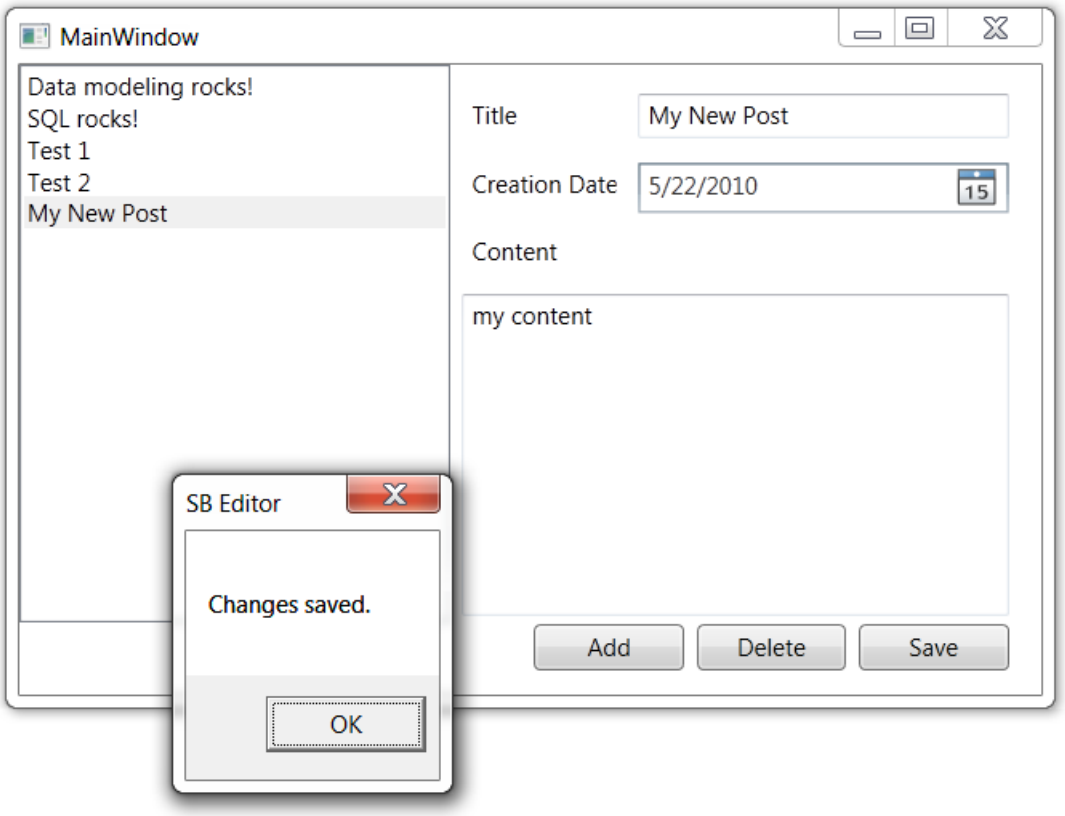


Figure 32: The sample editor application using OData

And while the Data Services client makes it look almost like programming against the Entity Framework, instead of turning our C# into SQL on the wire, it turns it into OData requests. For example, the initial query to pull in the set of posts when our WPF editor application starts up looks like this:

OData HTTP Request

[21] WPF data binding, and WPF itself, is a large topic and very much beyond the scope of this chapter. However, I recommend "Programming WPF," 2ed by Ian Griffiths and Chris Sells, for in depth coverage. (http://tinysells.com/121)

```
GET /sbcontent.svc/Posts()?$orderby=CreationDate%20desc&$top=25 HTTP/1.1
User-Agent: Microsoft ADO.NET Data Services
DataServiceVersion: 1.0;NetFx
MaxDataServiceVersion: 2.0;NetFx
Accept: application/atom+xml,application/xml
Accept-Charset: UTF-8
Host: localhost:8080
Connection: Keep-Alive
```

OData HTTP Response

```
HTTP/1.1 200 OK
Server: ASP.NET Development Server/10.0.0.0
Date: Sat, 22 May 2010 22:11:35 GMT
X-AspNet-Version: 4.0.30319
DataServiceVersion: 1.0;
Content-Length: 4798
Cache-Control: no-cache
Content-Type: application/atom+xml;charset=utf-8
Connection: Close

<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<feed xml:base="http://localhost:8080/sbcontent.svc/"
  xmlns:d="http://schemas.microsoft.com/ado/2007/08/dataservices"
  xmlns:m="http://schemas.microsoft.com/ado/2007/08/dataservices/metadata"
  xmlns="http://www.w3.org/2005/Atom">

  <title type="text">Posts</title>
  <id>http://localhost:8080/sbcontent.svc/Posts</id>
  <updated>2010-05-22T22:11:35Z</updated>
  <link rel="self" title="Posts" href="Posts" />
  <entry>
    <id>http://localhost:8080/sbcontent.svc/Posts(1)</id>
    <title type="text"></title>
    <updated>2010-05-22T22:11:35Z</updated>
    <author>
      <name />
    </author>
    <link rel="edit" title="Post" href="Posts(1)" />
    <link rel="http://schemas.microsoft.com/ado/2007/08/dataservices/related/Comments"
      type="application/atom+xml;type=feed" title="Comments" href="Posts(1)/Comments" />
    <category term="sbdb.Post"
      scheme="http://schemas.microsoft.com/ado/2007/08/dataservices/scheme" />
    <content type="application/xml">
      <m:properties>
        <d:Id m:type="Edm.Int32">1</d:Id>
        <d:Title>Data modeling rocks!</d:Title>
        <d:CreationDate m:type="Edm.DateTime">2010-05-16T00:00:00</d:CreationDate>
        <d:Content>I'm a &lt;b&gt;huge&lt;/b&gt; data modeling fan.</d:Content>
      </m:properties>
    </content>
  </entry>
  <entry>
    <id>http://localhost:8080/sbcontent.svc/Posts(2)</id>
    ...
  </entry>
  ...
```

```
</feed>
```

The power of OData is that it can be served from any data source, hosted on any server, consumed on any client, used from any platform and programmed on any language (so long as HTTP and XML are supported). The fact that .NET makes it very easy to serve OData and consume OData is just gravy.

Where are we?

So far, we've gone from designing a database schema to deploying the database, from populating the database with test data to migrating the data along with the schema, from accessing the data in our .NET application to exposing it and accessing it via the standard Open Data protocol.

All of that is a lot, but it's just the beginning of what you can do with the Microsoft data platform. In the other chapters of this book, you'll read about deploying your databases, using SQL Azure for hosting your database in the cloud, Integration Services for transforming your data in bulk between multiple sources, Reporting Services so that you can give users behind-to-scenes access to your data without building custom application code, Modeling Services so that you can have a text experience for editing your data models as well as a graphical one and synchronization so that you can keep multiple stores of the same data in sync.

In addition, you'll also get much more depth on Entity Framework, Data Services and the data support in Visual Studio so you have what you need to write real-world applications against the Microsoft data platform.

And speaking of real-world applications, by extending the model described in this chapter and doing a bit more with web design and layout, the functionality of the old sellsbrothers.com, implemented in 591 C# and ASP.NET code files, was reimplemented in 48 code files, while maintaining complete functionality and getting noticeably faster. That's a 10x gain in my book and the results are shown in Figure 33.

Figure 33: The new hotness

Bio

Chris Sells is a Program Manager for the Business Platform Division. He's written several books, including Programming WPF, Windows Forms 2.0 Programming and ATL Internals. In his free time, Chris hosts various conferences and makes a pest of himself on Microsoft internal product team discussion lists. More information about Chris, and his various projects, is available at http://www.sellsbrothers.com.

References

This book is an excerpt from “Programming Data,” by Chris Sells, with Shawn Wildermuth, from Addison-Wesley, 2010 (hopefully!). You can see a full list of draft chapters on Chris’s web site at http://www.sellsbrothers.com/writing/databook/