Pro Entity Framework 4.0

Scott Klein

Apress®

Pro Entity Framework 4.0

ISBN-13 (pbk): 978-1-59059-990-7

ISBN-13 (electronic): 978-1-4302-0648-4

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Publisher and President: Paul Manning
Lead Editor: Jonathan Gennick
Technical Reviewer: Vidya Vrat Agarwal
Editorial Board: Clay Andres, Steve Anglin, Mark Beckner, Ewan Buckingham, Gary Cornell, Jonathan Gennick, Jonathan Hassell, Michelle Lowman, Matthew Moodie, Duncan Parkes, Jeffrey Pepper, Frank Pohlmann, Douglas Pundick, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh
Project Manager: Anita Castro
Copy Editor: Tiffany Taylor and Mary Ann Fugate
Compositor: Bob Cooper
Indexer: BIM Indexing & Proofreading Services
Artist: April Milne
Cover Designer: Anna Ishchenko

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at www.apress.com/info/bulksales.

To my parents, Richard and Carolyn.

Contents at a Glance

Contents

About the Author

Scott Klein is a Microsoft SQL Server MVP and independent consultant specializing in SQL Server performance and business intelligence. Scott is the author of several books including *Professional SQL Server 2005 XML*, *Professional LINQ*, and *Professional Windows Communication Foundation*, and was a contributing author on *Pro SQL Server 2008 Relational Database Design and Implementation* as well as *Microsoft SQL Server 2008 Bible*. Scott has also written many articles for *SQL Server Standard* magazine. Scott holds the MCDBA, MCSD, and MCSE certifications and is heavily involved in the South Florida community, running two SQL Server user groups and South Florida SQL Saturday events. He frequently speaks at user groups across South Florida. Scott has nearly 20 years' experience with SQL Server, working and consulting in small to enterprise-size environments.

About the Technical Reviewer

Vidya Vrat Agarwal, is a Microsoft .NET Purist and an MCT, MCPD, MCTS, MCSD.NET, MCAD.NET, and MCSD and a lifetime member of the Computer Society of India (CSI). Vidya started working on Microsoft .NET with its 1st beta release and has been involved in software development, evangelism, consultation, corporate training, and T3 programs on Microsoft .NET for various employers and corporate clients. He is a published author for Apress titles *Beginning C# 2008 Databases: From Novice to Professional, Beginning VB 2008 Databases: From Novice to Professional and Pro ASP.NET 3.5 in VB 2008 : Includes Silverlight 2Pro* as well as a technical reviewer of many books published by Apress.

Vidya lives with his beloved wife, Rupali, and lovely daughter, Vamika ("Pearly") and believes that nothing will turn into a reality without them. He is the follower of the concept No Pain, No Gain and believes that his wife is his greatest strength. He is a bibliophile and blogs at `http://dotnetpassion.blogspot.com`. You can reach him via email at `Vidya_mct@yahoo.com`.

Acknowledgments

Anyone who has written a book knows that it isn't the work of a single individual. Although the author (or authors) are at the forefront, a myriad of individuals behind the scenes actually make the book a reality, making sure all the pieces come together smoothly (or as smoothly as possible). It is these behind-the-scenes individuals whom I want to thank profoundly.

First and foremost, the people at Apress. Jonathan Gennick, the person who has stuck by me while getting this project off the ground and through a period when we thought it would never get done: my hat is off to you. Thank you for your support.

Anita Castro, the coordinating editor who kept me honest and on schedule as best she could. I hope I didn't put you through too much stress.

Greg and Keith, what can I say? Great times.

In addition to those already mentioned, you are reading this book because of the support of many friends and family. Herve and Jared, you guys rock, and your friendship is unmeasurable. Joe Healy, what would South Florida do without you?

To my contacts at Microsoft who allowed me to ask a thousand questions: Elisa Flasko, Noam Ben-Ami, Diego Vega, Alex James, Daniel Simmons, and Tim Mallalieu. Thank you for your time and help in this great endeavor.

Nothing, however, compares to the support and love that you receive from those at home. To my wife and children (Lynelle, Sadie, Forrest, Allara, and Shayna), thank you for the love, support, and wonderful home. You make it all worth while.

CHAPTER 1

Introducing the ADO.NET 4.0 Entity Framework

In July 2008 Microsoft released the first version of the ADO.NET Entity Framework as part of the Visual Studio 2008 Service Pack 1 as well as the .NET Framework 3.5 Service Pack 1. At the time, Microsoft LINQ (Language Integrated Query) and LINQ to SQL had been out for a while and were gaining a lot of attention. Both the Entity Framework and LINQ to SQL showed that Microsoft was very serious about improving developer productivity, by providing an infrastructure for managing relational data as objects and programming against a conceptual model instead of directly against a storage schema. While Microsoft did its best to tout the Entity Framework, it went somewhat unnoticed out of the gate, primarily due to the rise in popularity of LINQ to SQL and the misunderstanding from the public of what the Entity Framework really was.

By the time this book hits the shelves, Visual Studio 2010 with the .NET Framework 4.0 and ADO.NET Entity Framework will have just been released or will be very shortly. Yet, I still get questions from people wondering what the Entity Framework is or how it differs from LINQ to SQL.

Thus, the reason for this book is twofold; first, it will answer the question of what the Entity Framework is and why Microsoft is dedicating a lot of resources and energy to its development, advancement, and developer acceptance. Second, it will discuss all the new features and enhancements that will be available in ADO.NET 4.0, which will be released with Visual Studio 2010.

This book is intended to address two types of developers: those who have never worked with the Entity Framework before, and those who have but are looking at what the ADO.NET 4.0 Entity Framework has to offer. As such, it will cover all aspects of the Entity Framework and, where appropriate, point out the new features and enhancements found in the EF 4.0.

The Entity Framework does not, nor is intended to, replace existing ADO.NET data access technologies; rather, it is an enhancement to ADO.NET, providing developers an augmented approach to accessing data, letting them work with a conceptual model, and thus enabling developers to deal with data as objects and properties, a concept already familiar to them.

The Need for an Entity Framework

To really understand what the Entity Framework is and why it is so important, we need to take a step back and look at some of the existing data access technologies. Microsoft has put a lot of time and effort into ADO.NET over the past many years. Prior to that, it was RDO, and prior to that it was DAO. Heaven forbid we look prior to DAO. With ADO.NET, developers were finally sensing that Microsoft had settled on a data access strategy and technology. With the improvements and enhancements that were being made to ADO.NET with each .NET release, it was the "go-to" technology for data access.

Developers' data access technologies of choice have primarily been the `DataReader` and the `DataSet`, and these have been serving developers well for many years. Yet with all of the improvements being made to ADO.NET, there was still a disconnect between the application and the back-end database.

Developers were spending far too much time trying to keep up with changes being made to the database. Any schema change to a table or stored procedure, for example, could potentially break an application.

Take the following code snippet, for example, which runs against Microsoft's standard AdventureWorks example database. Take a good look at this code sample and then ask yourself two questions:

1. Will it compile?
2. Assuming it compiles, when it runs, will you get the "yep, we have rows" message?

```
try
{
    string connectionString = Class1.GetConnectionString();

    using (SqlConnection conn = new SqlConnection(connectionString))
    {
        conn.Open();
        using (SqlCommand cmd = new SqlCommand())
        {
            cmd.Connection = conn;
            cmd.CommandText = "SELECT FirstName, MidleName, LastName FROM Person.Contact↩
 WHERE ContactID = @ContactID";
            SqlParameter param = new SqlParameter("@ContactID", SqlDbType.Int, 50,↩
 "ContactID");
            param.Value = 8;
            cmd.Parameters.Add(param);

            SqlDataReader rdr = cmd.ExecuteReader();
            while (rdr.Read())
            {
                Console.WriteLine(String.Format("{0}, {1}",
                    rdr[0], rdr[1]));
            }
        }
    }

}
catch (Exception ex)
{
    MessageBox.Show(ex.Message);
}
```

If you want to try it out, open up a new instance of Visual Studio 2010 (or 2008—this example is not specific to 2010) and create a new C# Windows Forms application. On the form, place a button on the form and in the click event of that button enter the code as you see it above. Prior to running the project, change the connection string username and password for your environment and add the following `using` statement:

```
using System.Data.SqlClient;
```

If you don't feel like typing, you can download the code from the Apress web site for this book. In any case, run the project and click the button. When the project runs and the form displays, it should be

obvious at that point that the code did compile successfully. Now click the button. Did you get the "yep, we have rows" message?

If you typed in the code just as you see it above, you didn't get the message indicating a successful run. Instead you received a message that the column `MidName` is an invalid column name. The column containing the middle name is actually called MiddleName. The problem is you had to run the application to find that out. Now imagine if yours had been a larger production application and you received a similar error. A larger application makes it much more difficult to find the offending line of code, plus you have the greater overhead of tracking down the actual spelling of the column name.

The point is that developers spend far too much time worrying about things they shouldn't need to when it comes to the database side of things. Developers need to focus only on developing applications and should not be concerned if a table, stored procedure, relationship, or some other database object changes.

In the code example above, the error that you get illustrates an example of a table change being made without informing the developers. When the table was originally created, it possibly could have had a column named `MidName`, but someone decided to change that column's name to `MiddleName` to be consistent with the other name columns in the table. Unless that schema change is disseminated to the appropriate people (the developers in this case), no one is going to find out about the change until run time. Think of what could happen if for some reason the change was not picked up and somehow slipped through QA and into the production release. Oops.

What was needed was a model where the database, application, and data move together. This is exactly what the Entity Framework provides. The Entity Framework is a conceptual model that works with databases and applications, eliminating the gap between data and applications (and application languages) that developers normally have to work with when working with the `DataReader` object and other data access technologies.

This Has Been Tried Before

Modeling applications exist in abundance, but each application has a specific focus such as process data flow or describing objects. The following lists some of the modeling application types and their primary focus.

- *ERM (Entity Relationship Model)*: Used with databases, it is a way to represent logical relationships of entities (objects) in order to create a database.
- *UML (Unified Modeling Language)*: A standard modeling language that is used to describe objects.
- *ORM (Object-Relational Mapping)*: Method of mapping relational databases and object-oriented programming languages.
- *DFD (Data Flow Diagram)*: A graphical representation of the flow of data between processes and between systems.

The problem is that each of these modeling applications is limited in scope. For example, ERM products do logical data models really well, but struggle at UML tasks. UML excels at describing objects, but falls short at performing ERM tasks.

And lest we forget, there are other ERM-like products in the market that have been around for a bit and provide Entity Framework-like capabilities, such as the following:

- *NHibernate*: Ported from Hibernate Core for Java for the .NET Framework, NHibernate for .NET persists plain .NET objects to and from an underlying relational database.

- *SPRINT.net*: An open-source application framework, and based on the Java version of Spring Framework, allowing you to build components that can be integrated into multiple tiers of your application.

While this list is by no means complete, it is provided to illustrate the necessity and need for good ERM application development products that remove many of the complexities of working with databases to help improve productivity.

The ADO.NET Entity Framework does not have the same limitations of other modeling products. This is because the Entity Framework works at a conceptual level that is based on the ERM and, as such, provides a depth and richness of functionality that many of the stand-alone ERMs and UMLs cannot provide.

As the name suggests, the Entity Framework allows you to work directly with Entities that represent your own schema without having to deal with the nuances of `DataReader` and `DataSet` objects. Some developers who have worked with or have some knowledge about Entity Framework compare it to other relational mappers and try to classify it as an object relational mapper. This comparison or thought process is not entirely correct. The Entity Framework does have relational mapping capabilities. But the Entity Framework is much more than just a relational mapper, and the ORM capabilities it does have are implemented much differently than that of other ORM products.

So, What Is the Entity Framework?

The Entity Framework is a set of technologies in ADO.NET that helps fill in the space between object-oriented development (objects) and databases. This gap is commonly known as an "impedance mismatch" and it exists because the mapping and organization of classes does not quite match up to the organization of relational objects. Many mapping solutions have tried to solve this problem by mapping OO (object-oriented) classes and properties straight to tables and columns.

For example, let's use the case of customers and products. First, in a simple mapping scenario, you may have a product class that contains a property that references an instance of a customer class. Second, a customer class might contain a property containing a collection of instances of the product class. While this might seem fine, how do you represent the foreign keys between customer and product in the first case (while in the second case you have a reference of an object-oriented collection of product instances that has no comparable column in the customer table in the database)?

A scenario such as this one is known as the conceptual space and is accomplished by raising the abstraction level to a point that lets developers query entities and relationships in the conceptual model, all while letting the Entity Framework translate the query operations to data source–specific commands. It allows applications to be written against conceptual models and not directly against the database. By doing so, the gap between object-oriented programming and databases has been closed, letting developers focus on the task of developing applications, without concerning themselves about the database (structure or otherwise) or data access.

You should start to see where the gap comes into play, simply because there is no one-to-one mapping. Relationships between relational objects are represented much differently from class relationships. And this is where the Entity Framework takes a much different approach. The EF maps relational objects to conceptual models. Conceptual mapping provides a number of huge benefits. For example, it provides the needed improved flexibility in creating (defining) and enhancing the logical model.

■ **Note** The Entity Framework divides the data model into three separate models: conceptual, logical, and physical. Each of these will be discussed in detail in Chapter 3.

By putting the focus on the conceptual model, the Entity Framework lets you query entities and relationships, letting the EF do the work of translating the query operations into commands specific to the data source, such as Microsoft SQL Server.

Developers can also work in an object-oriented environment that is familiar to them, all while gaining the benefits of operating in such an environment, such as IntelliSense and compile-time checking. For example, had the example a few pages ago been done using the Entity Framework, the error discussed earlier would have been caught during application compilation.

Keep in mind there are many more benefits to Entity Framework than what I've discussed so far. They will be discussed throughout this chapter and the rest of this book.

Database vs. Model

You now have two data access choices. On one hand are the tried and true `DataReader` and `DataSet` objects. On the other is the Entity Framework. Which should a developer choose? Since this book is about the Entity Framework, obviously the book is going to steer you in that direction. However, the next two sections are going to point out the differences between both options.

Database-Driven

Developers familiar with `DataReaders` and `DataSets` know that the majority of the time and code is making a connection to the database, getting data from the database, and then performing some action on the results as the results are assigned to classes. At the very minimum, the code to open a connection and execute a query would look something like the following (modified from the example above):

```
using (SqlConnection conn = new SqlConnection(@"Data Source=(local);Initial↩
 Catalog=AdventureWorks;UID=sa;PWD=pwd"))
{
     conn.Open();
     using (SqlCommand cmd = new SqlCommand())
     {
          cmd.Connection = conn;
          cmd.CommandType = CommandType.Text;
          cmd.CommandText = "SELECT FirstName, MiddleName, LastName FROM Person.Contact↩
 WHERE LastName = @LastName";
          cmd.Parameters.Add(new SqlParameter("@LastName",↩
 System.Data.SqlDbType.char)).Value = “Smith”;
          using (SqlDataReader rdr = cmd.ExecuteReader())
          {
         if (rdr.HasRows)
              {
                   Rdr.Read();
                   //
              }
          }
     }
}
```

While this type of coding has served developers well in the past, it leaves developers with a mixed model where the application code and data (i.e., schema) are all mixed together; the application implicitly contains the model. In the system I have just described, the application is tied to a specific set of database features and neither the application nor the database makes sense without each other.

Model-Driven

With the Entity Framework, you do not have to worry about the database. Rather, you simply code and query against a set of objects (entities) that reflects the business model. Results are returned as objects, and unlike other data access options, the developer does not have to spend time (code) figuring out rows and columns in the returned results just to bind them to objects. Since results are returned as objects, this work is automatically done.

A term or concept that that you should familiarize yourself with is an Entity Data Model (EDM). The EDM is the foundation of the Entity Framework and is comprised of the three models mentioned earlier: the conceptual model, the logical model, and the storage model. Think of the EDM as an "enhanced" version of an ERM. The EDM is the component that describes the overall structure of your business objects. The Entity Framework does not map your database objects in a simple 1:1 (one-to-one) mapping. While it can do that, it can do much more than that. Consider, for example, the following figure from the AdventureWorks database.

In Figure 1-1 there are three tables: Contact, Employee, and AdditionalContactInfo. Person provides information about a person, while the Employee table provides additional data regarding that person as an employee. The Business Entity table is the source of the ID that connects many of the tables such as people (persons), employees, and other information.

For most, if not all, developers this structure looks very familiar. Developers who write applications that access data are extremely familiar with working with tabular data. The past few pages, however, have highlighted many of the pitfalls when working in that environment.

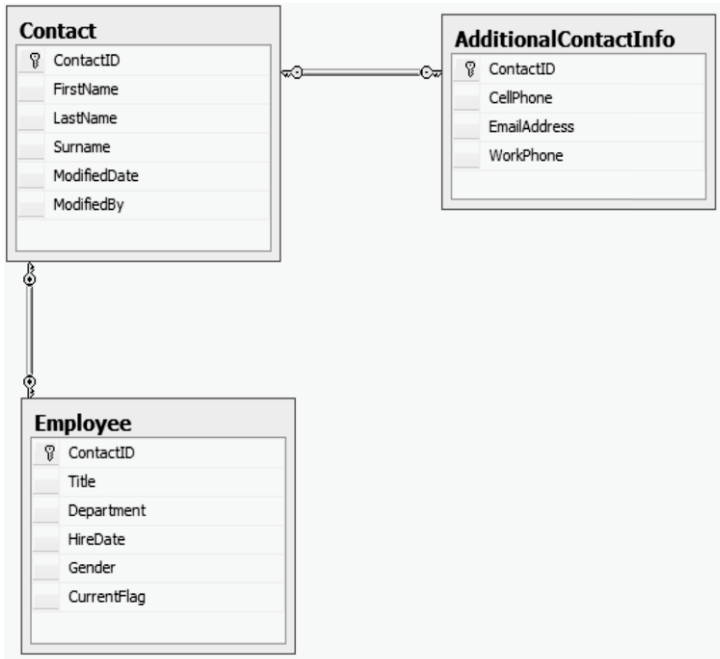

Figure 1-1. *Database Model*

A developer might need to write a query that pulls information from all of these tables. For example, the application might require that a form that displays the employee's first name, last name, job title, hire date, and possibly some other information. A SQL Server query to pull data might look something like the following:

```
SELECT c.FirstName, c.LastName, e.Title, e.HireDate, aci.CellPhone, aci.EmailAddress
FROM Contact c
INNER JOIN AdditionalContactInfo aci ON c.ContactID = aci.ContactID
INNER JOIN Employee e ON c.ContactID = e.ContactID
```

The)Entity Data Model of the Entity Framework is not the same as the database model, and it is important that a distinction between the two be made. The EDM describes the structure of your business objects; the database model describes the database schema. The two are distinct and separate, but the goal of the EF is to reform or reorganize the database objects in such a way that your EDM matches the goal of your business layer instead of an exact match of your database schema.

For example, using the Entity Framework and the EDM, a developer can take the above three tables and reshape them using inheritance as follows:

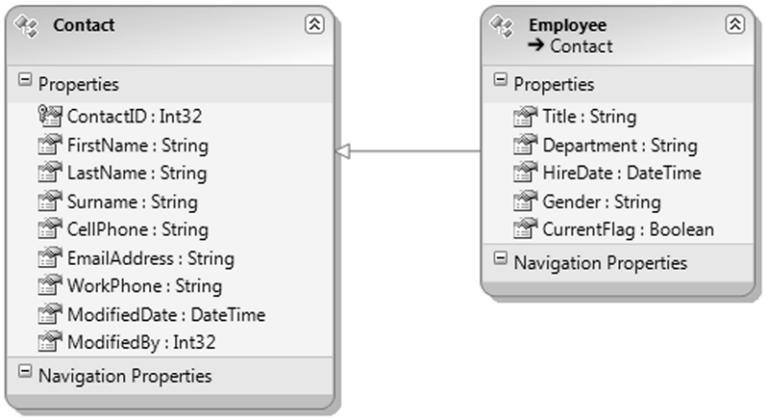

Figure 1-2. *Reshaped schema using inheritance*

With LINQ the pervious query can now be written much more clearly and cleanly, as follows:

```
From c in Contact.TypeOf<Employee> select c;
```

Remember that with the Entity Framework developers now work at the object level, not at the tabular level. No more tabular translation or the need to manually map results to objects. The next chapter, Chapter 2, will discuss the EDM in great detail.

Working with Entities

A key term any Entity Framework developer needs to know is the term *entity*. The objects in Figure 1-2 are called entities. Chapter 4 will discuss entities in much greater detail, but they are introduced here since they are vital in understanding how to work with the EDM.

In some ways, entities are like objects, for example:

- Entities have a known type.
- Entities have properties, and these properties can hold scalar values.
- Entity properties can hold references to other entities.
- Each entity has a distinct identity.

Even though entities have properties, entities have very little behavior. What behavior they do have is strictly limited to the methods that enable change tracking.

Entities also differ from objects, for example:

- Entities live within a collection.
- Each entity has associations with other entities.
- Entities have primary keys that uniquely identify the entity.

Equally important, entities also have some similarities to relational data:

- Entities live within an entity set.
- Entities have relationships to other entities.
- They have a primary key.

On the flip side, they differ from relational data:

- Entities support complex types.
- Entities support inheritance.
- Entities do not have physical storage knowledge.

Entities are extremely flexible, meaning that they can be reshaped as you saw a few pages ago when models were discussed. Entities can have relationships between them, and those relationships can be defined directly in the EDM Designer.

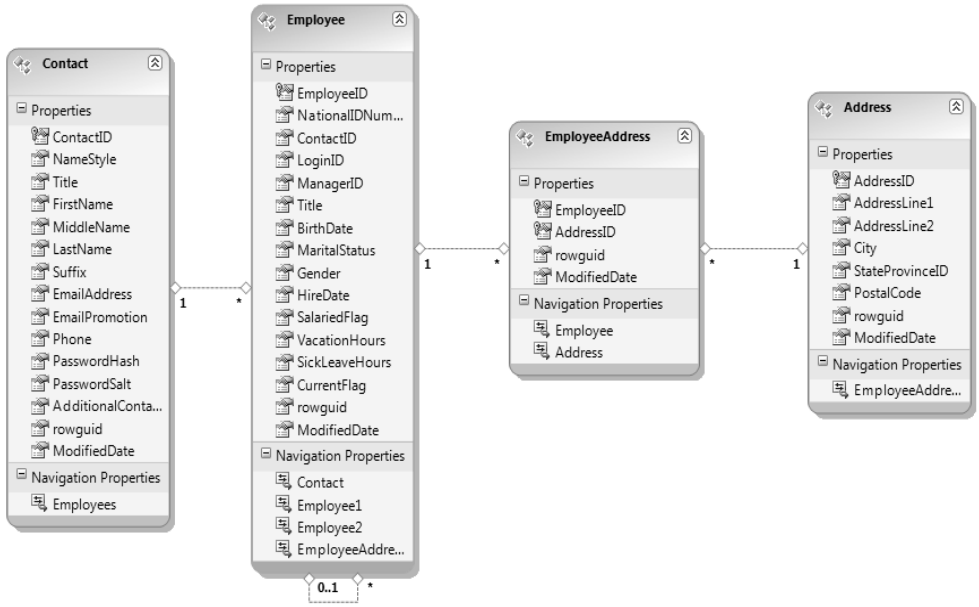

Figure 1-3. *Entities and relationships*

You saw earlier how easy it is to query against this type of model. The model provides all the appropriate and necessary navigation between the entities, and you will see a lot more of this in Chapter 4. The EDM makes it extremely simple to visually see how the entities are linked. In Figure 1-3, each entity has a navigation property. This property visually illustrates how a developer can easily "navigate" between the entities.

One of the many nice features of the Entity Framework is the ability to retrieve what is called a graph. This does not refer to the pie/bar chart type of graph. An Entity Framework graph is the ability to return shaped data. For example, in a single result I can return data such as a salesperson and all of that person's contact details along with it.

Entity Framework 4.0 Features

Lastly, this chapter is going to briefly cover some of the new features in version 4.0 of the Entity Framework. There are a lot of new features and enhancements, but these next few pages will highlight some of the most requested and needed features.

Following are some of the more important features and enhancements to be aware of. Some saw the lack of some of these features in the first version of the Entity Framework as a notable shortcoming.

- Plain Old CLR Objects (POCO) Support
- Model-First Support
- Deferred Loading of Related Objects
- Functions in LINQ to Entities Queries

- Plurality Naming Support
- Complex Type Support
- Customized Object Layer Code Generation
- Model Browser Improvements

The following sections will briefly discuss each item. A more detailed discussion of each item is included throughout the book.

POCO Support

One of the more powerful new features of the Entity Framework is the ability to add and use your own custom data classes in conjunction with your data model. This is accomplished by using CLR objects, commonly known as "POCO" (Plain Old CLR Objects). The added benefit comes in the form of not needing to make additional modifications to the data classes. This is also called persistence-ignorance.

The flexibility of extending these partial classes means more control over the core entity object functionality. This is a huge advantage as developers can now leverage and preserve valuable customizations and business logic, which they might not have been able to do previously.

Model-First Support

In the first version of the Entity Framework, developers could create a conceptual model using the Create Database Wizard, but that model could not be persisted (created) based on that model. This changes with the ADO.NET 4.0 Entity Framework.

When creating your initial conceptual model via the Create Database Wizard you can now create the database based on the conceptual model. This is a huge plus for developers who like to create the object model first and generate the database based on the model. This functionality supports the data-driven design that the EDM is purely based on.

Related Object–Deferred Loading

Deferred loading is also known as lazy loading, and in the first version of the Entity Framework, related objects were not automatically loaded from the data source as navigation properties were accessed.

In ADO.NET 4.0 Entity Framework, query results can be shaped by composing queries that explicitly navigate the relationships via the navigation properties.

LINQ-to-Entities Function Support

Function support in the first version of the Entity Framework was limited. A *function* represented either a stored procedure or a UDF in the database.

Two new classes have been added to the Entity Framework for this release to address this issue, the `EntityFunctions` and `SqlFunctions` classes. These classes provide developers the ability to access canonical and database functions via LINQ to Entities queries.

Additionally, a new attribute called `EdmFunctionAttribute` gives the Entity Framework the ability to use a CLR method to serve as a proxy for a function defined in the conceptual model.

More on function support is discussed in Chapter 5.

Plurality Naming

One of the big complaints in the first version of the Entity Framework was how naming conventions were applied to EDM objects such as entities and navigation properties when using the model wizards.

The first version of the Entity Framework gave the Entity Name and the Entity Set Name the same name. There was no attempt to singularize or pluralize names when generating a model from a database.

The problem is that this caused some confusion when referencing the database table or EntityType in code. For example, if your database has a table called Employees, then you will also get an EntityType called Employees as well. This causes confusion about whether you are referencing the table or the EntityType, such as in the code fragment below.

```
Customers customer = new Customers();
```

Luckily, this issue has been addressed. The model wizards, both the Entity Data Model and Update Model Wizards, now provide the option of using singular or plural forms of names for entities, entity sets, and navigation properties.

The goal of this change was to make the application code much easier to read and avoid a lot of the confusion between object names.

More on the EDM is discussed in Chapter 2.

Complex Types

A big addition to this version of the Entity Framework is the support for complex types. Complex types are like entities in that they consist of a scalar property or one or more complex type properties. Thus, complex types are non-scalar properties of entity types that enable scalar properties to be organized within entities.

Complex types are discussed in detail in Chapter 3.

Customized Object-Layer Code Generation

Object-layer code is generated, by default, by the EDM using the Entity Model Code Generator tool. This version of the Entity Framework allows developers to add text templates to a project that replaces the default tool to generate the object-layer code.

By using custom text templates, the EDM will generate the object context and entity classes. The Entity Framework makes it very easy to add custom templates via the EDM.

Complex Types are discussed in detail in Chapter 3.

Model Browser Improvements

Several improvements have been made to the model browser that make working with the EDM much more pleasant. Improvements include the following:

- Updating the model when changes are made to the underlying database.
- Deleting objects from the model.
- Searching for a specified string in the storage and conceptual models.
- Locating entity types on the design surface.

This list is by no means complete, as many more improvements have been made to the model browser.

The model browser enhancements will be discussed in detail in Chapter 2.

Back-End Support

The great thing about the Entity Framework is that in essence it does not really care about the data store from which the data is being queried. It doesn't need to. Neither the type of database nor the schema itself is completely unknown to the Entity Framework, and they will have no impact on your model.

Out of the box, the Entity Framework ships with two providers:

- *EntityClient Provider for the Entity Framework*: Used by Entity Framework applications to access data described in the EDM. This provider uses the .NET Framework Data Provider for SQL Server (SqlClient) to access a SQL Server database.
- NET Framework Data Provider for SQL Server (SqlClient) for the Entity Framework: Supports the Entity Framework for use with a SQL Server database.

The great thing about the Entity Framework is that it is database-, or data source–, independent, in that you can create custom providers to access other databases. For example, through third party providers you can access the following:

- Oracle
- MySql
- PostgreSQL
- SQL Anywhere
- DB2
- Informix
- U2
- Ingres
- Progress
- Firebird
- Synergy
- Virtuoso

This is quite a list and it shows you that the Entity Framework is gaining in popularity. This list is by no means complete, as providers are continuously being created by third-party vendors. At the time of this writing, a complete list of providers and their vendors can be found here:

```
http://msdn.microsoft.com/en-us/data/dd363565.aspx
```

The great thing about this is that the provider does all of the work for you pertaining to query reshaping. You are responsible for providing the connection information to the specific data store, but the provider takes care of the rest when working with the Entity Framework. Why? Because of the need to learn databases or figure out the nuances of different data stores. Instead, you use the Entity Framework's query syntax such as LINQ to Entities or Entity SQL and forgo the headache of remembering the database differences.

This book will use Microsoft SQL Server 2008 as its database and will use the .NET Framework Data Provider for SQL Server (SqlClient) for the Entity Framework as the data provider. Through this provider you can use SQL Server as far back as version SQL Server 2000 up through SQL Server 2008. Microsoft even supports SQL Server Compact Edition.

CHAPTER 2

The Entity Data Model

Chapter 1 spent quite a bit of time discussing the need for, and introducing, the Entity Framework. As part of that discussion the chapter introduced the EDM (Entity Data Model) and its many benefits to you. As you have learned, the EDM is the bridge between your application and your data and is the component that allows you to work with your data conceptually rather than going directly against your database and trying to figure out the back-end schema.

This chapter is going to build on the first chapter and spend the entire time taking an in-depth look at an EDM and how it works. This chapter will walk you through creating an EDM, and then we will lift the lid, take a look underneath, and explore every nook and cranny of the EDM. This chapter will spend quite a bit of time looking at the EDM Designer and its related files. The first thing this chapter is going to walk you through is the creation of your first Entity Data Model. From there, we'll lift the lid and discuss everything there is about it.

Creating an EDM

In the previous version of the Entity Framework you could generate a model from an existing database, and you could also start with an empty model and create the conceptual model from scratch.

However, that is where the functionality ended for creating an EDM. If you created a model from scratch, you could not create the database from that model and in some cases you were required to work with the raw XML (more on that later in this chapter).

There have been significant changes to the Entity Framework and the EDM. With the EF 4.0, some significant improvements have been made. Along with creating an EDM from an existing database schema (called database-first), you can now also do the following:

- *Model-first*: Allows you to start with an empty model, define your model and then generate the database, mappings, and classes from the defined model.
- *Code-only*: Allows you to use the Entity Framework using Plain Old CLR Objects (POCO) entities and without an EDMX file.

The database-first approach has been available since the very beginning of the EF with the release of .NET 3.5 SP1. The model-first approach is new to Visual Studio 2010 and .NET 4.0 and allows you to create an EDM from scratch. The code-only approach lets developers view their code as their model.

The following three sections will discuss each of the approaches for creating an EDM. I'm going to let you know now that there are a lot of screen shots, and they are included for two reasons. The first is to help those who are not familiar with the EF to get up and going with ease. The second is to help those already familiar with the EF to know where the new changes and enhancements are. All the examples in this book will use the AdventureWorks database for SQL Server 2008. That database can be downloaded from the CodePlex web site, from the following URL:

```
http://msftdbprodsamples.codeplex.com/Release/ProjectReleases.aspx?ReleaseId=18407
```

So, let's begin.

Taking a Database-First Approach

Fire up Visual Studio 2010 and create a new Windows Forms Application project. Make sure on the New Project dialog that you have the .NET Framework 4.0 selected (it should select this by default, however). I called my project EFDemo but you can call your project whatever you want.

Once the project is created, open solution explorer and right-click on the project. Select Add ➤ New Item from the context menu. This will open the Add New Item dialog shown below in Figure 2-1.

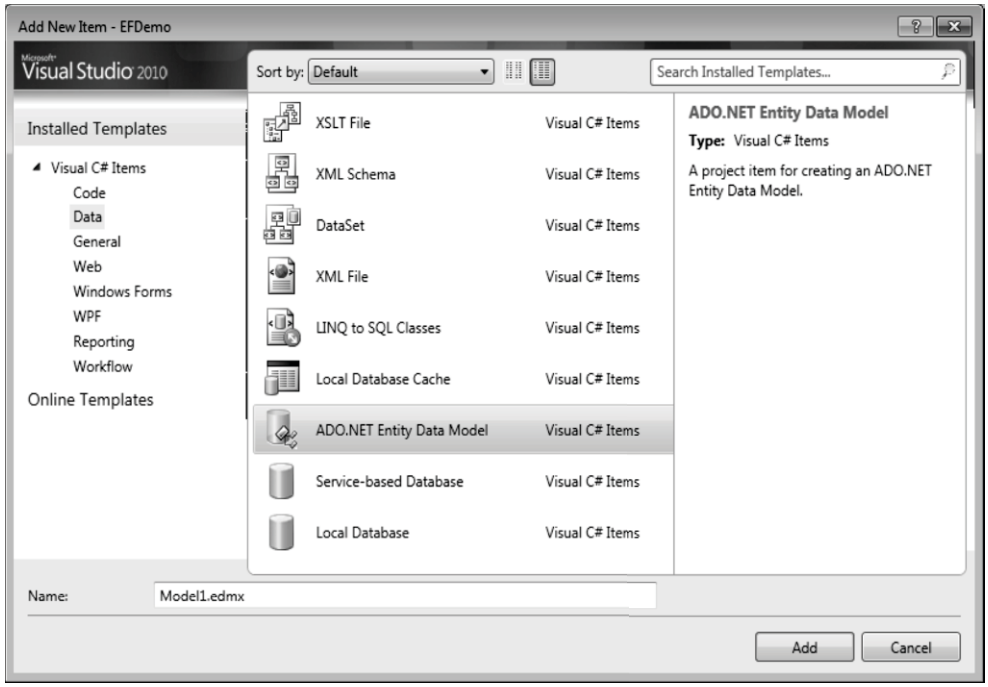

Figure 2-1. *The Add New Item dialog*

In the list of Installed Templates on the left side of the dialog, select the Data node. This will list all the Data object templates, among them the ADO.NET Entity Data Model template. Select the ADO.NET Entity Data Model template and click Add.

Selecting the ADO.NET Entity Data Model template will begin the Entity Data Model Wizard. The first step in the wizard lets you choose the type of model you want to use. In this step, you have the option of generating a model from a database or starting with an empty model.

This section deals with generating a model from an existing database schema; select that option as shown in Figure 2-2 and click Next.

Figure 2-2. *Entity data model wizard: choose model contents*

The next page in the wizard lets you specify the data connection for your EDM. If you have previously created connections they will show up in the list. If you have a connection to the AdventureWorks database, select that connection. Figure 2-3 shows this step in the wizard prior to making any selections.

If you haven't created any connections you will need to create one, and this can be done by clicking the New Connection button. This button will open the Connection Properties dialog, shown in Figure 2-4.