Chapter 19
ADS and the .NET Entity Framework

Note: This chapter accompanies the book Advantage Database Server: A Developer's Guide, 2nd Edition, by Cary Jensen and Loy Anderson (2010, ISBN: 1453769978). For information on this book and on purchasing this book in various formats (print, e-book, etc), visit: http://www.JensenDataSystems.com/ADSBook10

In nearly all of the chapters in Part III of this book, we have looked at specific Advantage drivers that permit you to execute SQL queries against your data. This chapter is a bit different in that we will be looking at a specific framework within .NET. Specifically, while the driver is the Advantage .NET Data Provider, we will be focusing on a framework built on top of ADO.NET, the .NET Entity Framework.

As is the case with the other chapters in Part III of this book, this discussion assumes that you are already familiar with the development environment that is being used, which in this chapter is Visual Studio and the C# language. As a result, the focus of this chapter is on the .NET Entity Framework, and the code that works with the data dictionary you have been using throughout this book.

What Is the .NET Entity Framework?

The .NET Entity Framework is an object-oriented system of types, interfaces, and objects that you can use in your applications to work an underlying database. When you use the .NET Entity Framework, you do not write traditional SQL statements. Instead, you use the methods of the classes of the entity framework to retrieve, edit, and save your data. In turn, those classes generate and execute the necessary SQL statements that perform these operations.

At first glance, the .NET Entity Framework appears to be an alternative to ADO.NET (which was covered in detail in Chapter 18, *ADS and ADO.NET*). In reality, when you use the .NET Entity Framework you are also using ADO.NET, though not directly. The classes that you interact with when using the .NET Entity Framework use the ADO.NET classes, such as DbConnections, DbDataReaders, and DbCommands, to perform the operations that you request.

Similarly, choosing to use the .NET Entity Framework in no way restricts you from using ADO.NET directly. In fact, there are a number of Advantage-related operations that are not directly available to you through the .NET Entity Framework, such as executing system stored procedures or reading from system tables. In those cases, the easiest way to perform those operations may be to connect an AdsConnection object to your database and access those operations through an AdsCommand. (If you want to use the .NET Entity Framework exclusively, you would need to write a SQL stored procedure that performs the particular operation you are interested in, and import that stored procedure into your entity model.)

There are two primary motivations behind the .NET Entity Framework. The first is to provide a mechanism for working with your data that is database agnostic (works the same regardless of which database you are using). For example, in the .NET Entity Framework you query your data using either Language INtegrated Query (LINQ) or Entity SQL, and neither of these languages are specific to a particular database (you use the same statements regardless of which database your data is stored in).

By comparison, when using ADO.NET, the SQL statements that you assign to your DbCommand instances must employ the dialect of SQL appropriate for the database to which they will be sent. In other words, DbCommands pass the SQL you write directly to your database. With the .NET Entity Framework, the SQL you write is interpreted by the classes of the entity framework, and those classes emit the appropriate dialect of SQL for the underlying database.

The second motivation to the .NET Entity Framework is to provide a consistent, object-oriented interface to your data, treating your data like other objects that you work with. For example, you often use LINQ to Entities to access your data using the .NET Entity Framework. LINQ to Entities is very similar in syntax and usage to LINQ to Objects, a declarative programming language that you can use in the .NET framework to manipulate collections, arrays, or IEnumerable objects. In other words, you data becomes just another type of object to use in code.

When using ADO.NET directly, you are accessing your data using the set-oriented approach provided for by the structured query language (SQL), and you must be comfortable with both the nature of SQL and that of relational databases. Using the .NET Entity Framework, your objects are responsible for maintaining the associations between the underlying tables of your relational database.

Both of these goals are achieve in the .NET Entity Framework through classes and collections of the System.Data.DataClasses namespace, including EntityObject and EntityCollection. These classes are base classes from which the objects that represent your data descend.

These classes represent what is called the *conceptual data model*. The conceptual data model is an abstraction of your underlying data, and not only defines the objects you will use to access your data, but also the relationships between these objects, based on your underlying relational database model.

Another set of classes represents your actual database architecture. This is what is referred to as the *storage model*. The storage model, which you do not work with directly in your code, is responsible for actually reading and writing data to and from your underlying database.

The relationship between the conceptual model and the storage model is defined by a set of mappings, which define the relationships between the various entities. Together, the conceptual model, the storage model, and their mappings work together to define what is called the *entity data model*.

The following section provides you with a more detailed description of entity data models, and how you create them.

Entity Data Models

In most cases, you will generate your entity data models, storage models, conceptual models, and their mappings, in Visual Studio using the provided wizards for C# and Visual Basic for .NET. However, in order to use these wizards, your database tables must include certain definitions. Specifically, your tables must support primary key definitions. In addition, those fields that participate in the primary key must have their Allow NULLS property set to False.

Currently, only the ADT and Visual FoxPro table formats associated with a data dictionary support these features. Clipper and FoxPro DBF tables, and any Advantage table type not associated with a data dictionary, do not support these features.

It is possible, however, to use these unsupported table types in entity framework applications. However, you cannot use the wizards provided by Visual Studio. Instead, you must manually define the storage schema definition (.SSDL file), and then use the edmgen2.exe tool provided by .NET 4.0 to generate your entity data model designer (*.EDMX), your conceptual schema definition (*.CSDL), and the mappings between the storage and conceptual models (*.MSL).

The .NET 3.5 framework (SP1) provides you with the edmgen.exe tool. This tool can generate *.CSDL and *.MSL files from your *.SSDL file, but it cannot generate the *.EDMX file.

Note: If you must create the storage schema definition language (SSDL) file manually, consider first exporting your tables to one of the formats that support non-null primary keys. Then, use the wizards in Visual Studio to generate the EDMX file, after which you can build your solution. When building a solution that includes an EDMX file, Visual Studio generates the associated SSDL file (search for a subdirectory named edmresources, which is a subdirectory of your project directory). You can the use this SSDL file as the starting point for your manually created SSDL file.

In addition to defining the entity types associated with tables in your database, the ADO.NET Entity Data Model wizards can also define associations between your table's entity types. For example, consider the CUSTOMER and INVOICE tables in the DemoDictionary data dictionary that you have been working with in this book. The

ADO.NET Entity Data Model wizards can recognize that there is a relationship between these two tables (a customer may be associated with zero or more invoices, and a given invoice is associated with zero or one customers).

In order for the ADO.NET Entity Data Model wizard to automatically determine the relationships between your tables, you must define the referential integrity (RI) definitions that represent these relationships. On the other hand, you might recall that in Chapter 5, *Defining Constraints and Referential Integrity*, we noted that referential integrity can sometimes introduce unwanted side effects, such as placing locks on all tables involved in the referential integrity definition even if only one of the tables involved is being edited.

It is possible to add related tables to your entity model without defining referential integrity constraints. Once again, however, this involves some manual adjustment. Specifically, you can add the tables to the model without RI, in which case the entity model generated by Visual Studio will not include associations. You can then using the entity model designer to manually add any required associations.

Creating the Entity Data Model

This section describes the steps necessary to create an entity data modal for selected tables from the DemoDictionary data dictionary. These steps make use of the ADO.NET Entity Data Model wizard for C# included in Visual Studio 2010 to define the various entity types and their associations.

Because we are using the wizard to generate the type definitions we will use in the sample application, it is important that the database objects (the tables and their table and field properties) are defined correctly. In addition, we need to ensure that the necessary referential integrity definitions are in place before using the template. As a result, this discussion begins by ensuring that your database is in the correct state.

Preparing the Database for Building the Entity Model

Even if you have followed all of the step-by-step instructions given in Chapters 4 and 5, your tables will still require a slight modification to at least one field-level constraint. If you did not follow the steps in Chapters 4 and 5, there may be even more changes that you will need to make.

Since each table will require some attention, we have simplified this example somewhat by adding only three of the tables from the DemoDictionary database to the entity data model. These tables are CUSTOMER, EMPLOYEE, and INVOICE. Use the following steps to ensure that these tables are in the correct state for use with the ADO.NET Entity Data Model wizard:

1. With DemoDicationary connected in the Connection Repository, right-click the CUSTOMER table and select Properties.

2. On the Fields tab of the Table Designer, select Customer ID in the Field Names list. With Customer ID selected, ensure that the Index property is set to Primary, and that NULL Valid is set to No. (NULL Valid is a property that you will likely have to change for all three of these tables, as none of the examples in earlier chapters of this book has required you to change this property from its default of Yes.) When you are done, click OK to save your changes.
3. Next, display the Table Designer for the EMPLOYEE table. With the Employee Number field selected in the Field Names list, ensure that Index is set to Primary and NULL Valid is set to No. Click OK when you are done.
4. Display the INVOICE table in the table designer. Select the Invoice No field in the Field Names list, and make sure that Index is set to Primary and NULL Valid is set to No. Click OK to close the Table Designer.
5. The final step is to verify that you have referential integrity constraints set between both the CUSTOMER and INVOICE tables, as well as between the EMPLOYEE and INVOICE tables. Begin by right clicking the RI OBJECTS node for the DemoDictionary data dictionary in the Connection Repository and select Visual Designer. If the RI Visual Designer contains the two referential integrity definitions shown in Figure 19-1, you are ready to go. If not, refer to Chapter 5, *Defining Constraints and Referential Integrity*, and follow the steps in the sections "Defining Referential Integrity Constraints" and "Using the Visual Designer" to create the two referential integrity constraints shown in Figure 19-1.

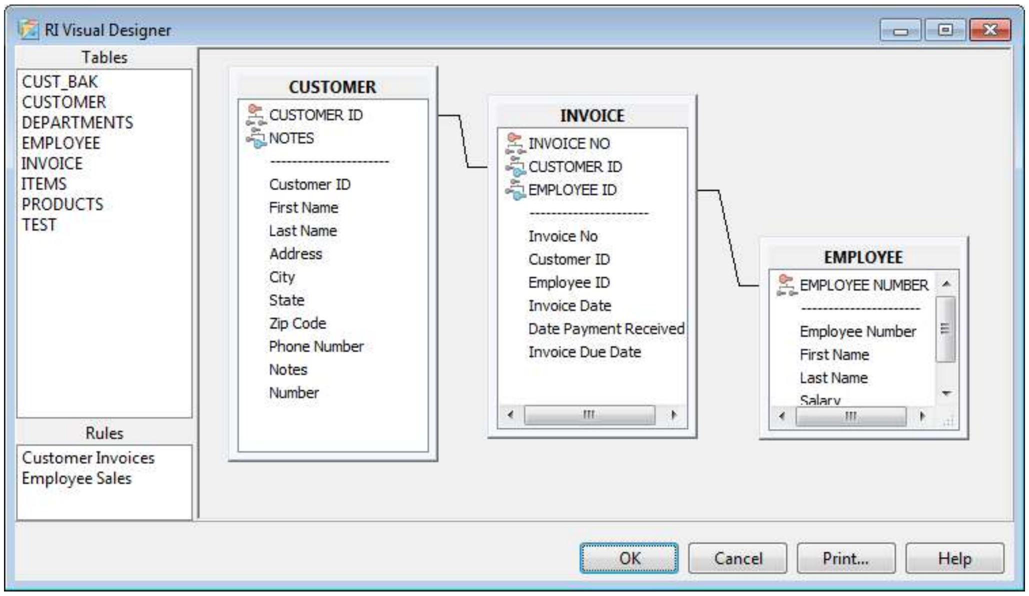


Figure 19-1: Two referential integrity constraints in the RI Visual Designer

Once your tables have been configured as described above, you are ready to use the ADO.NET Entity Data Model wizard.

Using the ADO.NET Entity Data Model Wizard

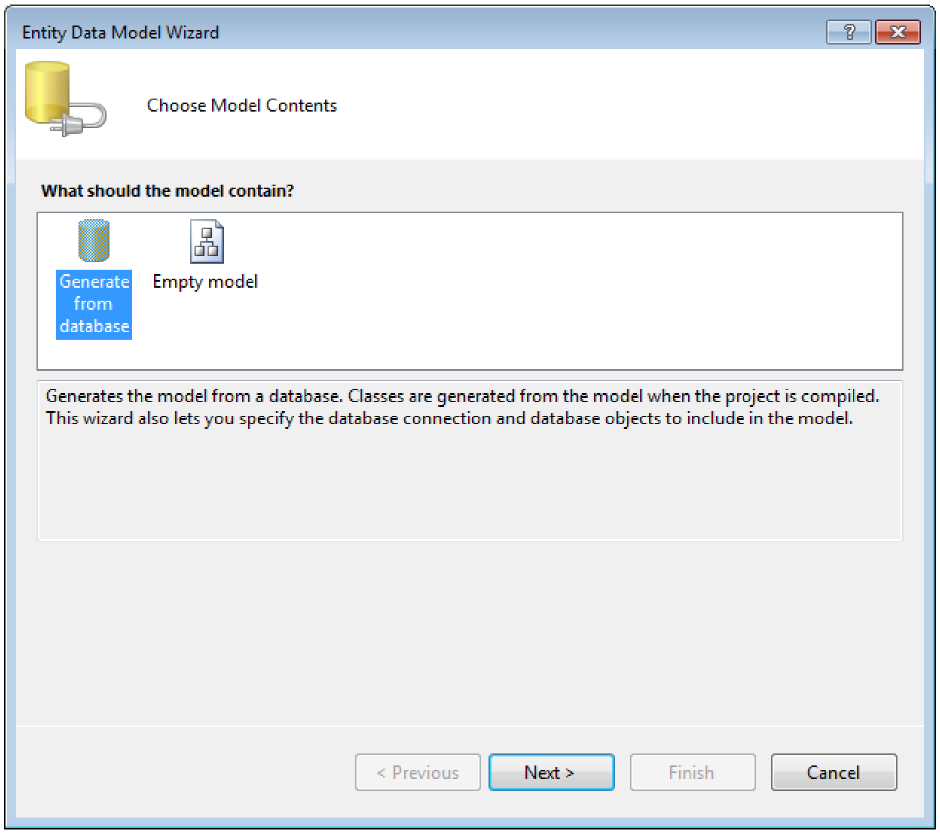
In order to use the .NET Entity Framework, it is necessary to define the classes that you will use to access your data. This requires that you define both a conceptual model and a storage model, as well as the mapping between them. While this can be done manually by creating the necessary files using the storage schema definition language, the conceptual schema definition language, and the mapping specification language (all XML-based languages), and then running the appropriate tool, such as edmgen2.exe. Taking this approach would be tedious, to say the least.

Fortunately, Visual Studio provides the ADO.NET Entity Data Model wizards for C# and Visual Basic for .NET developers. These wizards create an entity data model, as well as the necessary classes that you can use in your applications to work with your data.

Code Download: The examples provided in this chapter can be found in the C# project AdvantageEntityFramework available with this book's code download (see Appendix A). These projects were written in Visual Studio 2010.

The following steps demonstrate how to create the simple entity data model that is used in the sample application discussed in this chapter:

1. Begin by creating a new Windows forms application project using either the Windows Forms Application wizard for C# or Visual Basic for .NET. Save the project using a name of your choice.
2. Right-click the project name in the Solution Explorer and select Add | New Item.
3. From the Add New Item dialog box, select the ADO.NET Entity Data Model wizard and set name to InvoiceModel.edmx. When ready, click Add. Visual Studio displays the Entity Data Model Wizard, shown in Figure 19-2.
4. From the Entity Data Model Wizard, select Generate from database and click Next.

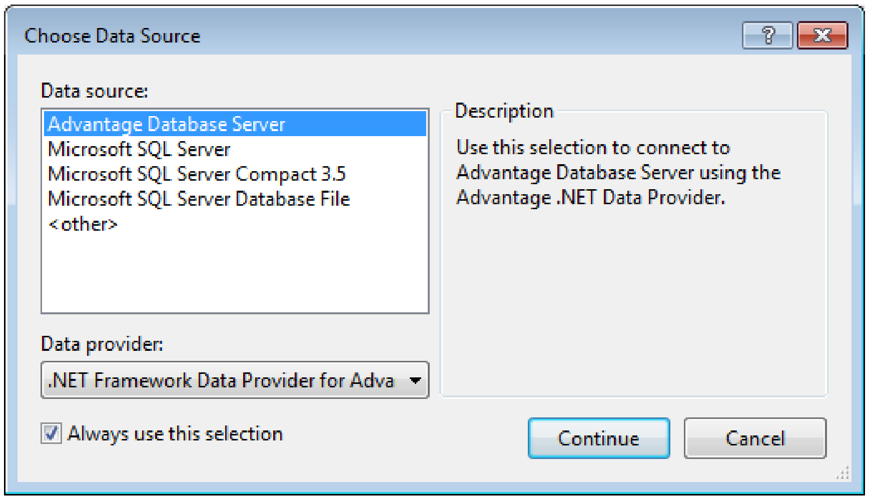


What should the model contain?

Generates the model from a database. Classes are generated from the model when the project is compiled. This wizard also lets you specify the database connection and database objects to include in the model.

Figure 19-2: The Entity Data Model Wizard

5. When asked to choose a connection, select Advantage Database Server as shown here and click Continue.



Data source:

Advantage Database Server
Microsoft SQL Server
Microsoft SQL Server Compact 3.5
Microsoft SQL Server Database File
<other>

Data provider: .NET Framework Data Provider for Adva

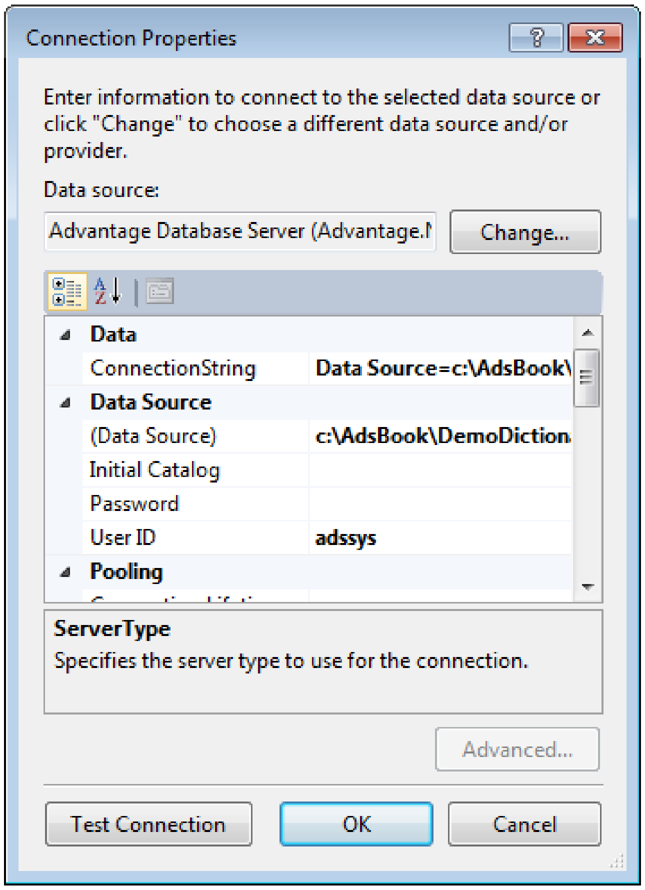
Description

Use this selection to connect to Advantage Database Server using the Advantage .NET Data Provider.

☑ Always use this selection

6. Next, use the Connection Properties dialog box to configure your connection string. Set Data Source to the path of your DemoDictionary data dictionary

file (we have been using c:\AdsBook\DemoDictionary.add throughout this book). Also, set User ID to adssys, ServerType to Remote, and TrimTrailingSpaces to True. Your Connection Properties dialog box should look something like that shown in Figure 19-3.



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

Data source:

Advantage Database Server (Advantage.N

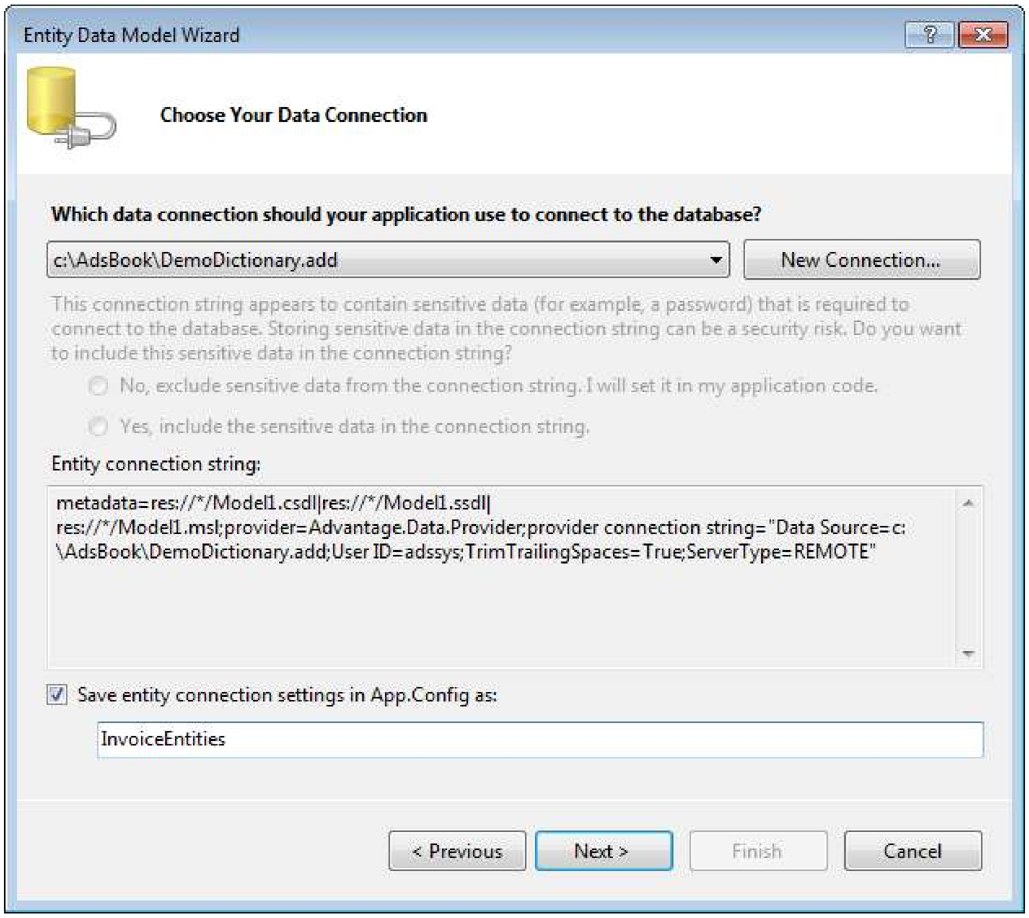
Data	
ConnectionString	**Data Source=c:\AdsBook\**
Data Source	
(Data Source)	**c:\AdsBook\DemoDiction**
Initial Catalog	
Password	
User ID	**adssys**
Pooling	

ServerType
Specifies the server type to use for the connection.

Figure 19-3: Configuring the connection string

7. Click Test Connection to test your connection using these settings. If there is an error, fix your connection properties and test the connection again. Once you get confirmation that you connection succeeded, click OK to return to the Choose Your Data Connection page of the Entity Data Model Wizard.

8. From this dialog box, make sure that Save entity connection settings in App.Config as is checked, and the name is set to InvoiceEntities, as shown in Figure 19-4. Click Next to continue.



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

c:\AdsBook\DemoDictionary.add

This connection string appears to contain sensitive data (for example, a password) that is required to connect to the database. Storing sensitive data in the connection string can be a security risk. Do you want to include this sensitive data in the connection string?

- No, exclude sensitive data from the connection string. I will set it in my application code.
- Yes, include the sensitive data in the connection string.

Entity connection string:

```
metadata=res://*/Model1.csdl|res://*/Model1.ssdl|
res://*/Model1.msl;provider=Advantage.Data.Provider;provider connection string="Data Source=c:
\AdsBook\DemoDictionary.add;User ID=adssys;TrimTrailingSpaces=True;ServerType=REMOTE"
```

☑ Save entity connection settings in App.Config as:

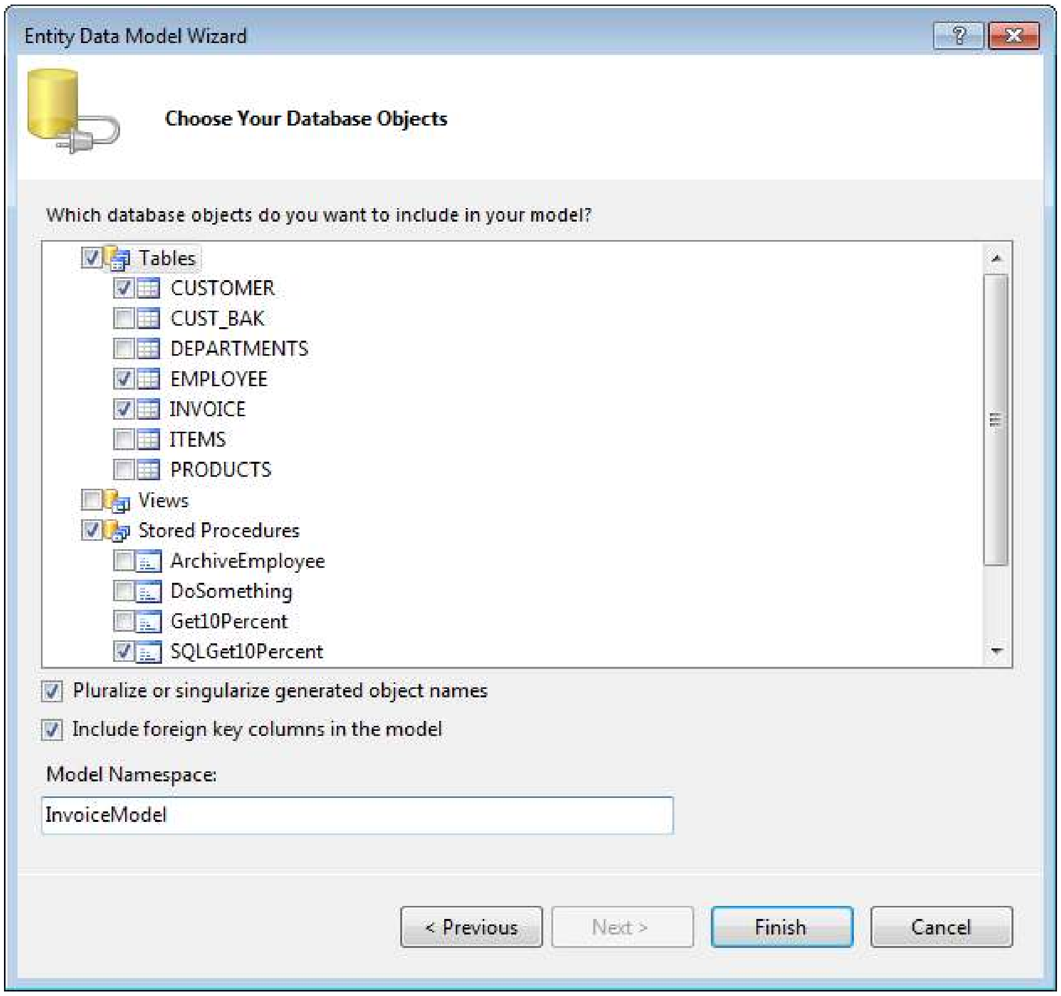
InvoiceEntities

Figure 19-4: Defining the data connection

9. The Entity Data Model Wizard now retrieves metadata from the database. Once it has completed this process, it displays the tables, views, and stored procedures defined in your data dictionary, and permits you to select which of these you want to use in your application. Since we want to include only the CUSTOMER, EMPLOYEE, and INVOICE tables in this model, expand the Tables node and place a check mark next to CUSTOMER, EMPLOYEE, and INVOICE.

10. Next, expand the Stored Procedures node, and place a check mark next to SQLGet10Percent.

11. Finally, set Model Namespace to InvoiceModel. When you are doing, the wizard should look something like that shown in Figure 19-5. Click Next to continue.



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

- [✓] Tables
  - [✓] CUSTOMER
  - [ ] CUST_BAK
  - [ ] DEPARTMENTS
  - [✓] EMPLOYEE
  - [✓] INVOICE
  - [ ] ITEMS
  - [ ] PRODUCTS
- [ ] Views
- [✓] Stored Procedures
  - [ ] ArchiveEmployee
  - [ ] DoSomething
  - [ ] Get10Percent
  - [✓] SQLGet10Percent

[✓] Pluralize or singularize generated object names

[✓] Include foreign key columns in the model

Model Namespace: InvoiceModel

Figure 19.5: The final step in configuring the entity data model

12. The Entity Data Model Wizard now examines the objects that you selected and generates a visual depiction of your entity objects in the ADO.NET Entity Data Model Designer, as shown in Figure 19-6.

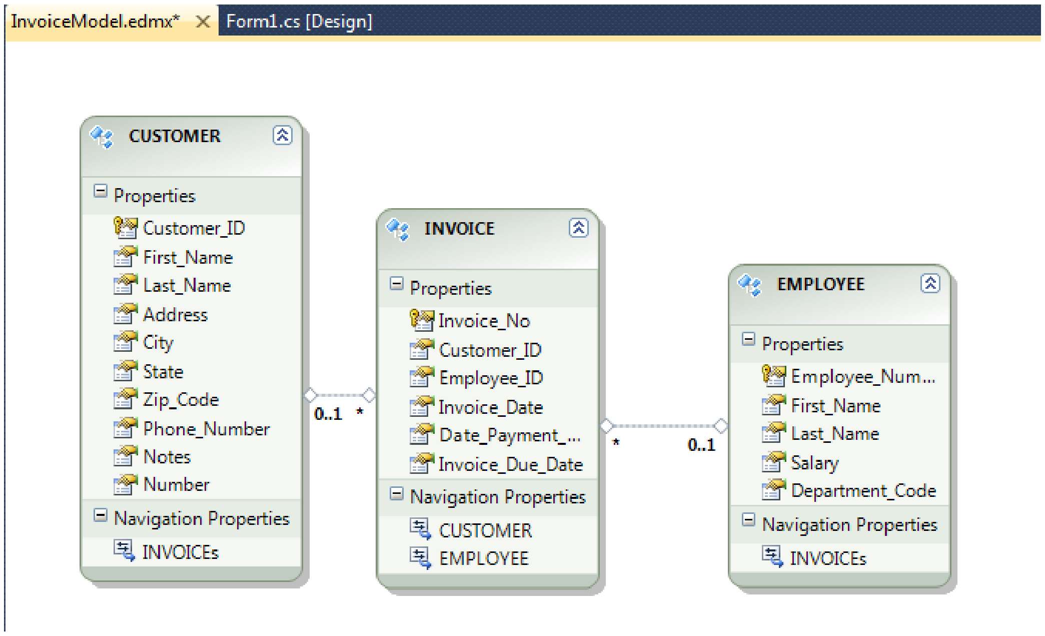
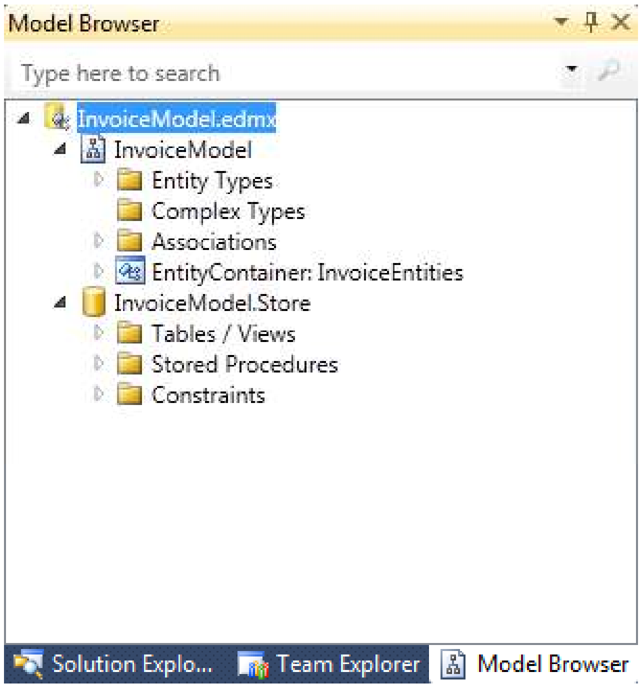


Figure 19-6: The InvoiceModel in the ADO.NET Entity Data Model Designer

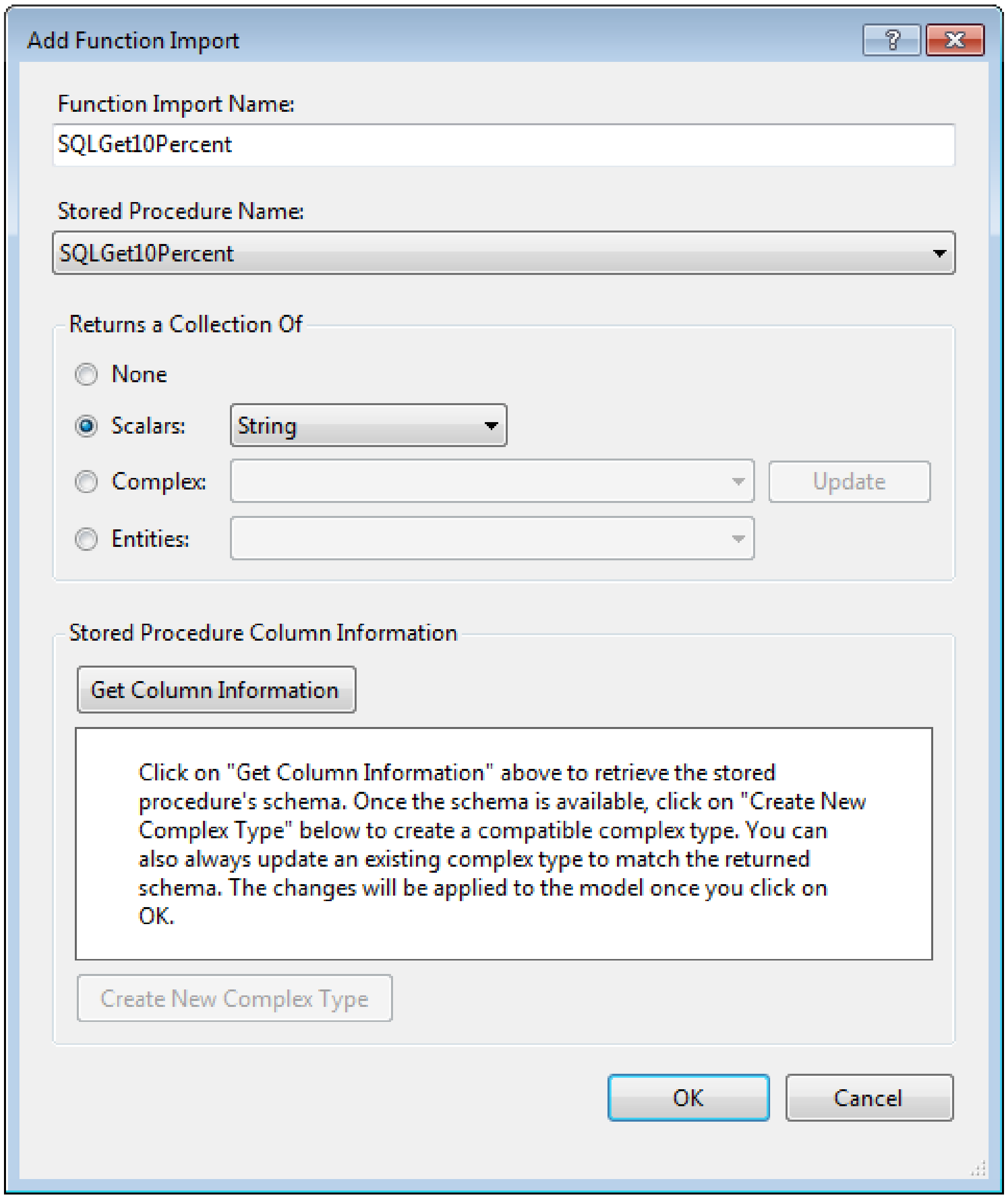
At this point, our entity data model is complete, as far as the table objects are concerned. However, we did select a stored procedure, and there is an additional step that you must take before you will be able to use it. Specifically, you need to import each of the stored procedures that you have included in your entity model.

Use the following steps to import the SQLGet10Percent stored procedure:

1. Begin by selecting the Model Browser tab. (By default, the Model Browser tab is located in the same tab group as the Solution Explorer, and is only visible when your entity model edmx file is selected in the editor. If you do not see the Model Browser tab, select the InvoiceModel.edmx file in your editor).



2. Expand the Stored Procedure node beneath InvoiceModel.Store.
3. Right-click SQLGet10Percent and select New Function Import.
4. Leave Function Import Name and Stored Procedure Name set to SQLGet10Percent.
5. In the Returns a Collection Of radio button group, select Scalars, and then select Strings from the associated combo box. When you are done, the Add Function Import dialog box should look like that shown in Figure 19-7.
6. Click OK to close. (Although the memo that appears on this dialog box instructs you to click the Get Column Information button, do not click that button. If you do, you will get a System.NotSupportedException exception.)


Add Function Import

Function Import Name: SQLGet10Percent

Stored Procedure Name: SQLGet10Percent

Returns a Collection Of

( ) None
(●) Scalars: String
( ) Complex:
( ) Entities:

Stored Procedure Column Information

Click on "Get Column Information" above to retrieve the stored procedure's schema. Once the schema is available, click on "Create New Complex Type" below to create a compatible complex type. You can also always update an existing complex type to match the returned schema. The changes will be applied to the model once you click on OK.

Figure 19-7: The Add Function Import dialog box

TE: The following paragraph appears to be incorrect. I actually removed the Advantage.Data.Entity.dll assembly from the sample project, and it ran fine. What is this assembly used for, and when would I need it? It seems like I should remove this following paragraph and the associated steps. Cary

The Advantage.Data.Entity.dll provides the translation from Linq to Entity and Entity SQL to the Advantage SQL syntax. I don't know how your project worked, but you definitely need this assembly. Including it in the project also makes it easier to ship a completed project since all the assemblies you need are part of the project and can be copied local. InstallShield can also check your dependencies at install time to make sure that everything is included.

There is one more step that you will need to take before you can start using your entity data model objects with Advantage. You will need to add the Advantage assembly that supports the entity framework to your project's references. This is described in the following steps:

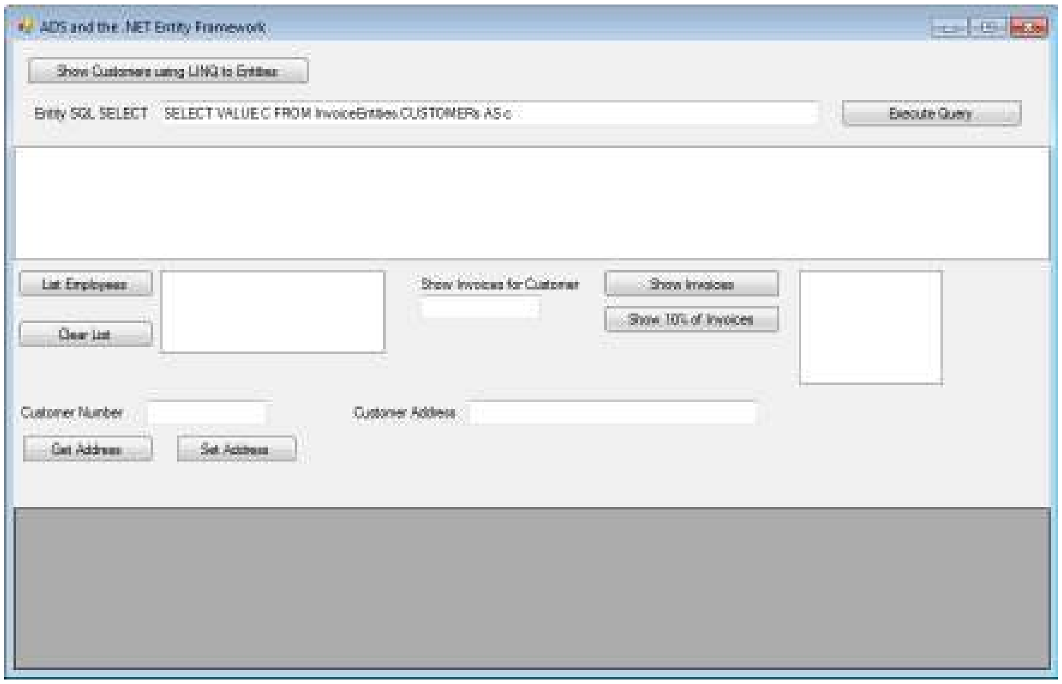
1. Right-click your project in the Solution Explorer and select Add Reference.
2. From the Add Reference dialog box, select the Browse tab. Use the controls on this page to navigate to the directory where the Advantage Data Provider is installed. By default, this will be in c:\Program Files\Advantage\ado.net. (If you are running a 64 bit operating system, the Advantage folder is beneath the Program Files (x86) directory.)
3. From that location, select either 3.5 (for 3.5 .NET Entity Framework support) or 4.0 (for 4.0 .NET Entity Framework support). Select the file named Advantage.Data.Entity.dll and click OK.

This brings up one more issue. The entity framework is support only by .NET 3.5 and later. Any project that you create and add an entity data model to must target .NET 3.5 or later.

Unlike the ADO.NET project described in Chapter 18, *ADS and ADO.NET*, it is not always necessary to add the Advantage.Data.Provider assembly as a reference to your project. The only time that Advantage.Data.Provider needs to be added as a reference is when you specifically want to use one of the concrete classes of the Advantage .NET Data Provider.

The Sample Application

The sample application associated with this chapter is roughly similar to the sample applications that appear in the other chapters in this section. The main form of this running application is shown in Figure 19-8.



Entity SQL SELECT SELECT VALUE C FROM InvoiceEntities.CUSTOMERs AS c

Figure 19-8: The main form of the AdvantageEntityFramework application

In creating this application, we wanted to demonstrate a variety of ways to interact with your data using the entity framework. As a result, while most of the examples employ entity objects to perform tasks, we have also included several examples that employ Entity SQL. In addition, many of the examples make use of Language INtegrated Query (LINQ).

If you are going to build applications using the .NET Entity Framework, you are going to need to become fluent in a number of technologies. In addition to LINQ, and possibly Entity SQL, you will also need to familiarize yourself with Lambda expressions (which we have not made use of in this chapter). All of these topics are outside the scope of this book. As a result, the descriptions of the code samples that follow do not attempt to make any detailed comments about these technologies, other than noting their presence in the code.

Performing Basic Tasks with Advantage and the .NET Entity Framework

This section describes some of the more common tasks that you can perform with the entity framework. These include connecting to your data, querying data using LINQ to Entities, querying data using Entity SQL, using a parameterized Entity SQL query, and executing a stored procedure.

Connecting to Data

The entity model used in the AdvantageEntityFramework project was created using the same steps shown earlier in this chapter. As a result, the connection string information associated with this project was stored in the App.Config file. Nonetheless, before we can use any of the entity types exposed by our entity model, we must instantiate an instance of the ObjectContext object generated by the Entity Data Model Wizard.

In this particular project, the name of the ObjectContext is InvoiceEntities. We created a private variable in our form class to hold a reference to this object. The following is this declaration:

```
private InvoiceEntities invoiceContext;
```

The instance of InvoiceEntities is created and assigned to the invoiceContext variable in the Load event handler for this form. The following is this event handler:

```
private void Form1_Load(object sender, EventArgs e)
{
  //Initialize the ObjectContext
  invoiceContext = new InvoiceEntities();
}
```

The InvoiceContext class is an IDisposable implementing class. As a result, we are responsible calling the Dispose method of this class when we no longer need it. The following is the FormClosing event handler for the main form:

```
private void Form1_FormClosing(object sender, FormClosingEventArgs e)
{
  //dispose of the object context
  invoiceContext.Dispose();
}
```

Once the form is loaded, the variable invoiceContext can be used to access the various objects of our conceptual model. For example, the following line of code demonstrates how the stored procedure that we imported in our model can be called from the ObjectContext:

```
invoiceContext.SQLGet10Percent(custNo);
```

If you want to use the EntityProvider (a .NET data provider included in the .NET Framework that you can use to work with an entity model), you are going to have to use another mechanism for defining the connection to your data dictionary. The following code shows how to configure an EntityProvider:

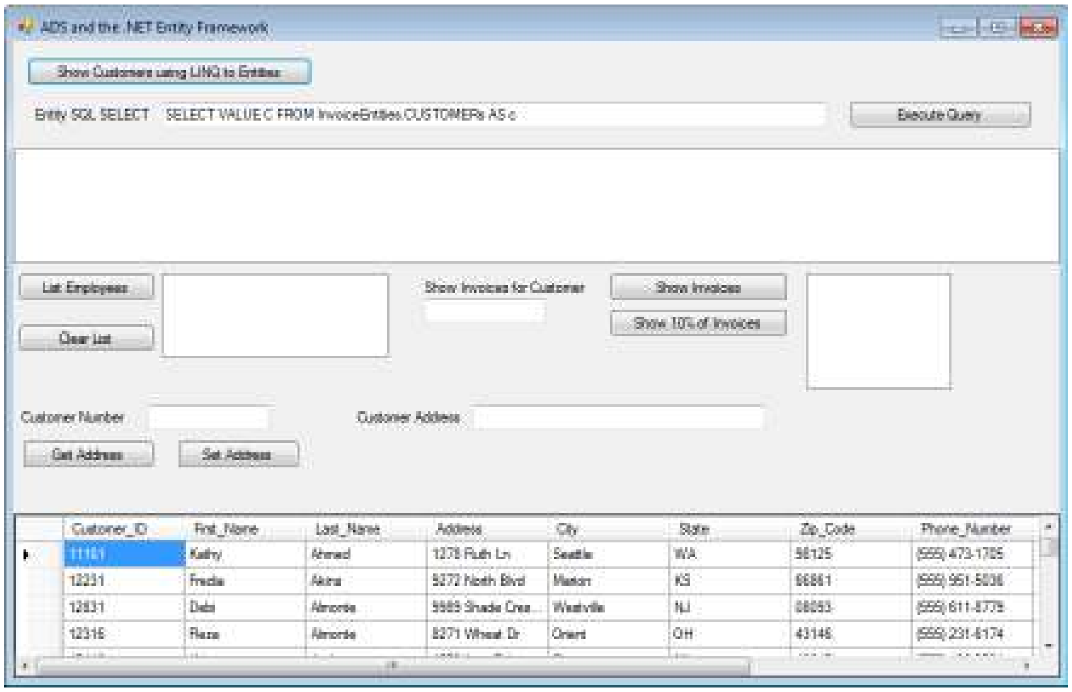
```
EntityConnectionStringBuilder entityBuilder =
  new EntityConnectionStringBuilder();
entityBuilder.Provider = "Advantage.Data.Provider";
entityBuilder.ProviderConnectionString = @"Data Source=
        c:\AdsBook\DemoDictionary.add;user ID=adssys;
        ServerType= LOCAL | REMOTE;TrimTrailingSpaces=True";
entityBuilder.Metadata = @"res://*/InvoiceModel.csdl|
         res://*/InvoiceModel.ssdl|
         res://*/InvoiceModel.msl";
EntityConnection conn = new
  EntityConnection(entityBuilder.ToString());
conn.Open();
// You can now get create EntityCommand and EntityDataReader
// classes to work with your data
```

Access Data Using LINQ to Entities

LINQ to Entities permits you to use LINQ to query the entity types in your entity data model. The code associated with the button labeled Show Customers Using LINQ to Entities demonstrates a simple LINQ query that retrieves all customers from the CUSTOMER table, sorted by the Last Name field. The result of this query is then bound to the data grid view named dataGridView1:

```
private void btnShowCustomers_Click(object sender, EventArgs e)
{
  // Define a query that returns all customer
  // objects ordered by last name.
  var customerQuery = from c in invoiceContext.CUSTOMERs
                      orderby c.Last_Name
                      select c;
  try
  {
    //Display the returned result in the data grid view
    this.dataGridView1.DataSource =
      ((ObjectQuery)customerQuery).Execute(MergeOption.AppendOnly);
  }
  catch (Exception ex)
  {
    MessageBox.Show(ex.Message);
  }
}
```

After clicking this button, the main form looks something like that shown in Figure 19-9.



Entity SQL SELECT SELECT VALUE C FROM InvoiceEntities.CUSTOMERs AS c

Customer_ID	First_Name	Last_Name	Address	City	State	Zip_Code	Phone_Number
11161	Kathy	Ahmed	1278 Ruth Ln	Seattle	WA	98125	(555) 473-1705
12231	Fredia	Akira	5272 North Blvd	Marion	KS	66861	(555) 951-5036
12831	Debi	Almonte	5585 Shade Cres...	Westville	NJ	08093	(555) 611-8779
12316	Reza	Almonte	8271 Wheat Dr	Orient	OH	43146	(555) 231-6174

Figure 19-9: A LINQ query retrieved the CUSTOMER data

LINQ queries can be very sophisticated, and can include a wide variety of LINQ methods and functions, as well as Lambda expressions, and can also perform transformations on the values that they return.

Accessing Data Using Entity SQL

Entity SQL is a SQL-like language for querying an entity data model. One advantage of Entity SQL is that it is storage independent, meaning that you use the same syntax regardless of the database from which your entity objects get their data.

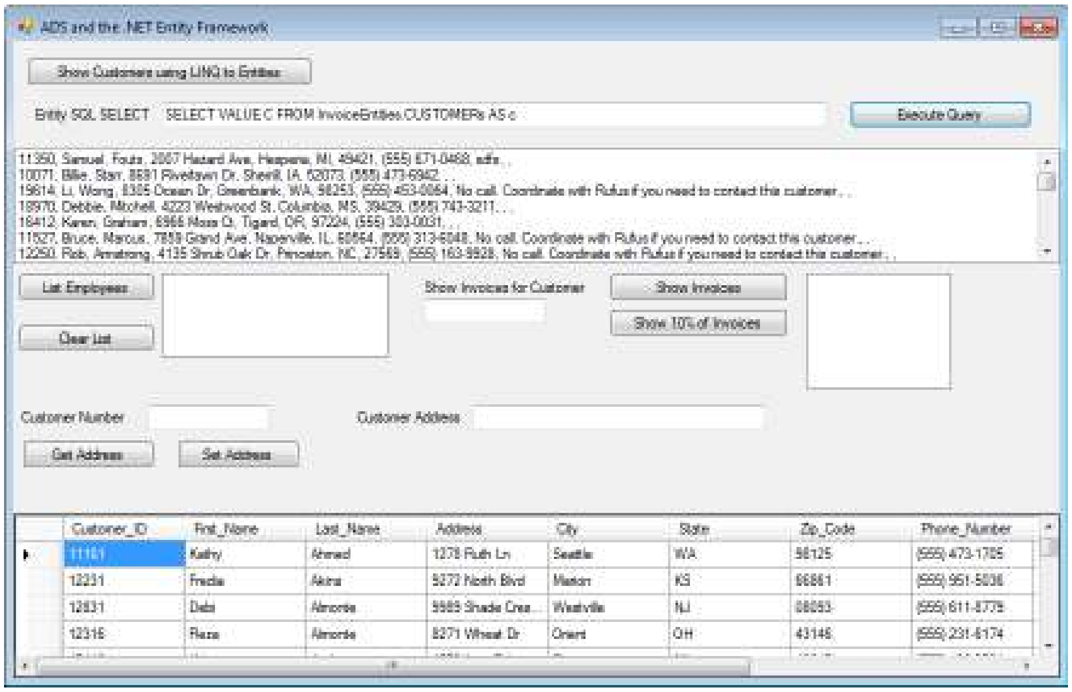
On the other hand, one of the limitations of Entity SQL is that there is no EntityDataAdapter class that you can use to fill DataTable objects. As a result, if you want to bind the data returned from Entity SQL queries, you will have to do so programmatically. Similarly, if you want to write changes made to data retrieved using Entity SQL, you will again have to do so programmatically.

The main form of the AdvantageEntityFramework project contains a button labeled Execute Query. This button configures an EntityConnection object, after which an EntityCommand instance is created and used to execute the Entity SQL query that appears in the TextBox to left of the button.

The following is the Click event handler associated with the button labeled Execute Query:

```
private void entitySQLQuery_Click(object sender, EventArgs e)
{
  try
  {
    listBox1.DataSource = null;
    EntityConnectionStringBuilder entityBuilder =
      new EntityConnectionStringBuilder();
    entityBuilder.Provider = "Advantage.Data.Provider";
    entityBuilder.ProviderConnectionString = @"Data Source=
         c:\AdsBook\DemoDictionary.add;user ID=adssys;
         ServerType= LOCAL | REMOTE;TrimTrailingSpaces=True";
    entityBuilder.Metadata = @"res://*/InvoiceModel.csdl|
         res://*/InvoiceModel.ssdl|
         res://*/InvoiceModel.msl";
    EntityConnection conn = new
      EntityConnection(entityBuilder.ToString());
    conn.Open();
    EntityCommand command = conn.CreateCommand();
    //Get the Entity SQL command
    command.CommandText = entitySQLTextBox.Text;
    StringBuilder sb = new StringBuilder();
    EntityDataReader dataReader =
      command.ExecuteReader(
        System.Data.CommandBehavior.SequentialAccess);
    while (dataReader.Read())
    {
      for (int i = 0; (i < dataReader.FieldCount); i++)
      {
        sb.Append(dataReader.GetValue(i).ToString().Trim() + ", ");
      }
      listBox1.Items.Add(sb.ToString());
      sb.Clear();
    }
    conn.Close();
  }
  catch (Exception ex)
  {
    MessageBox.Show(ex.Message);
  }
}
```

In this example, the query results are returned in an EntityDataReader, a read-only, forward navigating cursor to the result set. This data reader is then used to assemble a string representing each record of the query result, and these strings are added to the listbox that appears below the SQL statement, as shown in Figure 19-10.



```
SELECT VALUE C FROM InvoiceEntities.CUSTOMERs AS c
```

```
11350, Samuel, Fouts, 2007 Hazard Ave, Hesperia, MI, 49421, (555) 871-0468, adfs, ,
10071, Billie, Starr, 8691 Rivetlawn Dr, Sherill, IA, 52073, (555) 473-6942, , ,
19614, Li, Wong, 8305 Ocean Dr, Greenbank, WA, 98253, (555) 453-0064, No call. Coordinate with Rufus if you need to contact this customer., ,
18970, Debbie, Mitchell, 4223 Westwood St, Columbia, MS, 39429, (555) 743-3211, , ,
16412, Karen, Graham, 6966 Moss Ct, Tigard, OR, 97224, (555) 303-0031, , ,
11527, Bruce, Marcus, 7859 Grand Ave, Naperville, IL, 60564, (555) 313-6048, No call. Coordinate with Rufus if you need to contact this customer., ,
12250, Rob, Armstrong, 4135 Shrub Oak Dr, Princeton, NC, 27569, (555) 163-9828, No call. Coordinate with Rufus if you need to contact this customer., ,
```

Customer_ID	First_Name	Last_Name	Address	City	State	Zip_Code	Phone_Number
11161	Kathy	Ahmed	1278 Ruth Ln	Seattle	WA	98125	(555) 473-1705
12231	Fredia	Akins	5272 North Blvd	Marion	KS	66861	(555) 951-5036
12831	Debi	Almonte	5569 Shade Cre...	Westville	NJ	08093	(555) 611-8779
12316	Reza	Almonte	8271 Wheat Dr	Orient	OH	43146	(555) 231-6174

Figure 19-10: Data returned from an Entity SQL query is displayed in a list box.

Navigating a Query Result using LINQ

In the previous LINQ to Entities example shown in this section, the results of the LINQ query were cast as an ObjectQuery, which enabled it to be bound to a data source. The LINQ query results can also be iterated over, which permits the data to be read record-by-record.

An example of iterating over the results of a LINQ query is demonstrated in the Click event handler associated with the button labeled List Employees, as shown here:

```
private void btnListEmployees_Click(object sender, EventArgs e)
{
  //define the LINQ query
  var employeeQuery = from emp in invoiceContext.EMPLOYEEs
                      orderby emp.Employee_Number
                      select emp;
  try
  {
    //Add data about each employee to the list box
    foreach (var m in employeeQuery)
    {
      lbEmployees.Items.Add(String.Format("{0} : {1} {2}",
        m.Employee_Number, m.First_Name, m.Last_Name));
    }
  }
  catch (Exception ex)
  {
    MessageBox.Show(ex.Message);
  }
}
```

The results of this query execution can be seen later in this chapter in Figure 19-11.

Executing Parameterized Queries

It is arguable whether LINQ to Entities supports parameterized queries or not. Specifically, there is no parameter-type object in a LINQ query. If you need to have variable parts of a LINQ query, you achieve that end by including variables in the where clause. This is a bit like concatenating data from an outside source into a SQL query.

The following is a segment of code that demonstrates how to include external data in a LINQ query. In this case, the data is obtained from a text box on the main form:

```
  Int32 custNo;
  if (! Int32.TryParse(tbCustNumber1.Text, out custNo))
  {
    MessageBox.Show(tbCustNumber1.Text + " is not a number");
    return;
  }
  //define a query that selects a customer based on
  //the entered value
  var customerQuery = from c in invoiceContext.CUSTOMERs
                      where (c.Customer_ID == custNo)
                      select c;
```

While the custNo variable in this code segment is not a true parameter, there is not a security issue introduced by the use of external data from its use. Specifically, the nature of LINQ queries do not make them vulnerable to SQL injection in the same way that SQL scripts do.

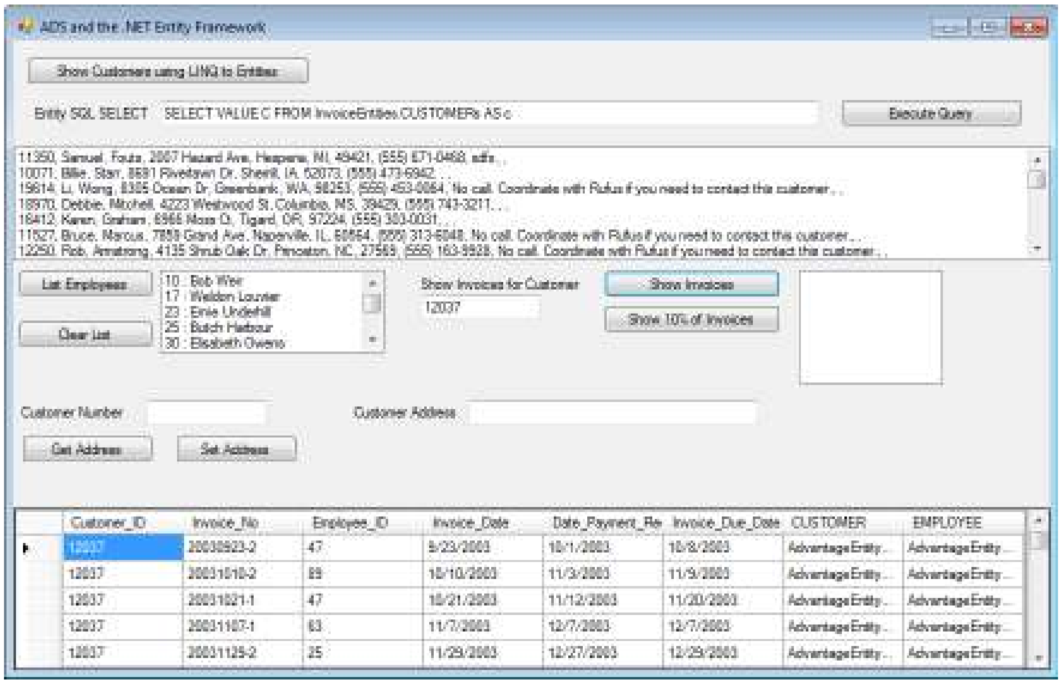
For truly parameterized queries, you can either use Entity SQL or a generic ObjectQuery. In both cases, your query string can include one or more named parameters, which are identified by the @ character. You then create one parameter object for each of the parameters in the query string before the query is executed.

An example of a parameterized query is provided in the Click event handler associated with the button labeled Show Invoices. After entering a customer number into a text box named tbCustNumber2, clicking the button constructs a parameterized verbatim string that is associated with an ObjectQuery obtained from the entity context. The value of the text box is then associated with the query parameter before the query is executed and its results displayed in a data grid view on the form.

The following is the code associated with the Click event handler from the Show Invoices button. After clicking this button, the main form looks something like that shown in Figure 19-11.

```
private void btnShowInvoices_Click(object sender, EventArgs e)
{
  if (tbCustNumber2.Text.Equals(String.Empty))
  {
    MessageBox.Show("Enter a customer number");
    return;
  }
  int custNo;
  if (! int.TryParse(tbCustNumber2.Text, out custNo))
  {
     MessageBox.Show(tbCustNumber2.Text + " is not a number");
     return;
  }
  try
  {
    string queryString =
      @"SELECT VALUE Invoices FROM invoiceEntities.INVOICEs
      AS Invoices WHERE Invoices.Customer_ID = @id";
    ObjectQuery<INVOICE> invoiceQuery =
      new ObjectQuery<INVOICE>(queryString, invoiceContext);

    // Add any required parameters
    invoiceQuery.Parameters.Add(new ObjectParameter("id", custNo));
    //execute the query
    dataGridView1.DataSource = invoiceQuery;
  }
  catch (Exception ex)
  {
    MessageBox.Show(ex.Message);
  }
}
```



Entity SQL SELECT SELECT VALUE C FROM InvoiceEntities.CUSTOMERs AS c

```
11350, Samuel, Fouts, 2867 Hazard Ave, Hesperia, MI, 49421, (555) 671-0468, adfs, ,
10071, Billie, Starr, 8691 Rivetown Dr, Sherill, IA, 52073, (555) 473-6942, , ,
19814, Li, Wong, 8305 Ocean Dr, Greenbank, WA, 98253, (555) 453-0884, No call. Coordinate with Rufus if you need to contact this customer, , ,
18970, Debbie, Mitchell, 4223 Westwood St, Columbia, MS, 39429, (555) 743-3211, , ,
16412, Karen, Graham, 6966 Moss Ct, Tigard, OR, 97224, (555) 303-0031, , ,
11527, Bruce, Marcus, 7858 Grand Ave, Naperville, IL, 60564, (555) 313-6048, No call. Coordinate with Rufus if you need to contact this customer, , ,
12250, Rob, Armstrong, 4135 Shrub Oak Dr, Princeton, NC, 27569, (555) 163-9928, No call. Coordinate with Rufus if you need to contact this customer, , ,
```

```
10 : Bob Weir
17 : Weldon Louvier
23 : Ernie Underhill
25 : Butch Harbour
30 : Elisabeth Owens
```

Show Invoices for Customer 12037

Customer Number

Customer Address

Customer_ID	Invoice_No	Employee_ID	Invoice_Date	Date_Payment_Re	Invoice_Due_Date	CUSTOMER	EMPLOYEE
12037	20030923-2	47	9/23/2003	10/1/2003	10/8/2003	AdvantageEntity...	AdvantageEntity...
12037	20031010-2	89	10/10/2003	11/3/2003	11/9/2003	AdvantageEntity...	AdvantageEntity...
12037	20031021-1	47	10/21/2003	11/12/2003	11/20/2003	AdvantageEntity...	AdvantageEntity...
12037	20031107-1	63	11/7/2003	12/7/2003	12/7/2003	AdvantageEntity...	AdvantageEntity...
12037	20031125-2	25	11/29/2003	12/27/2003	12/29/2003	AdvantageEntity...	AdvantageEntity...

Figure 19-11: A parameterized query returns all invoices for customer 12037.

Executing Stored Procedures from the Entity Data Model

The Entity Framework permits you to execute those stored procedures that you selected when you created your entity data model, as well as those that you have added later on. (Each of these stored procedures must also be imported, as described earlier in this chapter.)

The execution of the stored procedure named SQLGet10Percent is demonstrated in the code associated with the button labeled Show 10% of Invoices. Since the stored procedure returns a collection of strings, in the form of a generic ObjectResult, the return value can be bound to a control that accepts collections, which in this case is a list box.

The following code shows the Click event handler for the button labeled Show 10% of Invoices:

```
private void btnShow10Percent_Click(object sender, EventArgs e)
{
  if (tbCustNumber2.Text.Equals(String.Empty))
  {
    MessageBox.Show("Enter a customer number");
    return;
  }
  int custNo;
  if (! int.TryParse(tbCustNumber2.Text, out custNo))
  {
    MessageBox.Show(tbCustNumber2.Text + " is not a number");
    return;
  }
  try
  {
    lbInvoices.DataSource =
      invoiceContext.SQLGet10Percent(custNo);
  }
  catch (Exception ex)
  {
    MessageBox.Show(ex.Message);
  }
}
```

As in the preceding code example, once the code has verified that a customer number has been entered, that value is passed to the SQLGet10Percent method of the entity context (this method was created when you imported the function). The resulting value is then bound to the list box.

Note: If you need to execute Advantage system stored procedures from your .NET application, attach to your data dictionary using the Advantage Data Provider, and then use an AdsCommand object to execute your EXECUTE PROCEDURE queries. Alternatively, create your own SQL stored procedure that calls the system stored procedure you are interested in, and then import your SQL stored procedure into your entity data model.

Reading and Writing Data

One of the more interesting aspects of the .NET Entity Framework is that it takes responsibility for writing changes made to your entity objects back to the underlying storage. This provides the advantage of relieving you of the mundane details associated with your underlying data organization. On the other hand, since it is the classes of the .NET Entity Framework that generate the delete, insert, and update statements that will be executed on your data, you cannot take advantage of query optimizations.

In short, once you obtain data in the form of an EntityObject from your entity data model's ObjectContext, any changes made to the properties of the EntityObject are tracked. If you then call the SaveChanges method of the ObjectContext, the changes are persisted back to your database.

The changes to your entity objects can be made either programmatically or through bound controls. For example, you can select one or more records from your database through a LINQ query. You can then change the data in the query result, after which a call to SaveChanges will write those changes to your underlying database.

Alternatively, you can retrieve data from a LINQ query and bind that result to two or more visual controls that your end user can interact with. If the end user entered, modified, or deleted data using those controls, a subsequent call to SaveChanges will write those changes to your database.

The use of bound controls to edit data is demonstrated on the Click event handlers of the Get Address and Set Address buttons on the main form. When Get Address is clicked, the address of the specified customer is bound to the text box named tbCustomerAddress. If changes are made to the contents of this text box, subsequently clicking the button labeled Set Address calls the SaveChanges method of the object context.

Actually, the code is slightly more involved, since runtime binding is performed. Both of the Click event handlers use a helper function named BindingPosition, which returns the position of a named property binding for a given control, or -1 if the control does not have a data binding on that property. The Get Address button uses this function to test for the presence of the data binding, removing it if it already exists (otherwise an error might be generated when a new binding is created). The Set Address button uses this function to ensure that a data binding already exists (which indicates that an address was retrieved, and can now be saved).

The following code shows the BindingPosition helper function, as well as the Get Address and Set Address Click event handlers:

```
private int BindingPosition(Control control, string propertyName)
{
  for (int i = 0; i < control.DataBindings.Count; i++)
  {
    if (control.DataBindings[i].PropertyName.Equals(propertyName))
       return i;
  }
  return -1;
}

private void btnGetAddress_Click(object sender, EventArgs e)
{
  //If the address field is already data bound, remove the binding
  int bindingPosition = this.BindingPosition(tbCustomerAddress, "Text");
  if (bindingPosition >= 0 )
  {
    tbCustomerAddress.DataBindings.RemoveAt(bindingPosition);
  }
  // Verify that a valid customer number has been entered
  Int32 custNo;
  if (! Int32.TryParse(tbCustNumber1.Text, out custNo))
  {
    MessageBox.Show(tbCustNumber1.Text + " is not a number");
    return;
  }
  //define a query that selects a customer based on
  //the entered value
  var customerQuery = from c in invoiceContext.CUSTOMERs
                      where (c.Customer_ID == custNo)
                      select c;
  try
  {
    //get the result
    bindingSource1.DataSource =
      ((ObjectQuery)customerQuery).Execute(MergeOption.AppendOnly);
    //bind the result to the text property
    tbCustomerAddress.DataBindings.Add(new Binding("Text",
      bindingSource1, "Address"));
  }
  catch (Exception ex)
  {
    MessageBox.Show(ex.Message);
  }
}

private void btnSetAddress_Click(object sender, EventArgs e)
{
  if (this.BindingPosition(tbCustomerAddress, "Text") < 0)
  {
     MessageBox.Show("Get an address before trying to set it");
     return;
  }
  try
  {
    invoiceContext.SaveChanges();
  }
  catch (Exception ex)
  {
    MessageBox.Show(ex.Message);
  }
```

Navigational and Administrative Operations with the .NET Entity Framework

Most of the data access mechanisms discussed in this section of the book have supported some form of navigation. By navigation, we are specifically speaking about server-side, index-based navigation, such as seeks, filters, scopes and ranges, and navigating from one record to another.

As far as the .NET Entity Framework is concerned, these are not Advantage operations. Specifically, the .NET Entity Framework provides you with a conceptual model that represents an abstraction of your data. This model does not expose indexes from your underlying database structures, and therefore, the concept of index-based operations are absent.

On the other hand, there is the concept of record-by-record navigation in the .NET Entity Framework. For example, the returned records associated with the EMPLOYEE table were navigated in the following code segment, which was discussed earlier in this chapter:

```
var employeeQuery = from emp in invoiceContext.EMPLOYEEs
                    orderby emp.Employee_Number
                    select emp;
try
{
```

```
//Add data about each employee to the list box
foreach (var m in employeeQuery)
{
  lbEmployees.Items.Add(String.Format("{0} : {1} {2}",
    m.Employee_Number, m.First_Name, m.Last_Name));
}
```

Likewise, Entity SQL provides you with an EntityDataReader class, which can also be used to perform forward-only record-by-record navigation. In both cases, this navigation is limited, and is not really what we were referring to as navigation in the other chapters in this section.

On the other hand, the .NET Entity Framework does support a notion of navigation that is different from what we have been talking about. This navigation involves the associations found in the entity data model. For example, because of the relationship between the CUSTOMER table and the INVOICE table defined in the entity data model, the .NET Entity Framework navigation recognizes that when you are referring to a particular customer object, there is an associated object representing the invoices for that particular customer.

These relationships are particularly useful when you need to work with your data with respect to their associations. For example, they make it very easy to display a grid that contains all of the invoices for a currently selected customer, and to make changes to those invoices.

Once again, however, this is not what we mean by navigation when working with Advantage. In fact, with respect to the .NET Framework, if you really want navigation, you should use the Advantage Data Provider and its powerful AdsExtendedReader, as described in chapter 18, *ADS and ADO.NET*. The AdsExtendedReader gives you bi-directional navigation, server-side filters, and scopes, all powered by the indexes of your database.

With respect to administrative tasks, such as creating new tables, adding users, and changing the properties of the objects of your data dictionary, the limitations of the .NET Entity Framework are even more pronounced. Specifically, unless you specifically create custom stored procedures that expose administrative functionality, and import those into your entity data model, you cannot perform administrative tasks.

Here again, however, this is not really a problem. All you need to do is to use the Advantage Data Provider directly, and you have access to all of the system tables, system stored procedures, and all other features available through Advantage SQL. Refer to the section "Administrative Operations with ADS and ADO.NET" in Chapter 18 *ADS and ADO.NET* for information on performing administrative tasks using the Advantage Data Provider.