CHAPTER 22

Introducing Entity Framework 6

The previous chapter examined the fundamentals of ADO.NET. ADO.NET has enabled .NET programmers to work with relational data (in a relatively straightforward manner) since the initial release of the .NET platform. However, Microsoft introduced a new component of the ADO.NET API called the *Entity Framework* (or simply, *EF*) in .NET 3.5 Service Pack 1.

■ **Note** While this first version of EF was widely criticized, the EF team at Microsoft has been hard at work releasing new versions. The current version of EF for the full .NET Framework is currently (at the time of this writing) 6.1.3, which is packed full of features and performance enhancements over earlier versions. Entity Framework Core (formerly called EF 7) is also available and is covered in Part IX along with .NET Core.

The overarching goal of EF is to allow you to interact with data from relational databases using an object model that maps directly to the business objects (or domain objects) in your application. For example, rather than treating a batch of data as a collection of rows and columns, you can operate on a collection of strongly typed objects termed *entities*. These entities are also natively LINQ aware, and you can query against them using the same LINQ grammar you learned about in Chapter 12. The EF runtime engine translates your LINQ queries into proper SQL queries on your behalf.

This chapter will introduce you to data access using the Entity Framework. You will learn about creating a domain model, mapping model classes to the database, and the role of the `DbContext` class. You will also learn about navigation properties, transactions, and concurrency checking.

By the time you complete this chapter, you will have the final version of `AutoLotDAL.dll`. You will use this version of `AutoLotDAL.dll` for the rest of the book (until you rebuild it using EF .NET Core later in this book).

■ **Note** All the versions of the Entity Framework (up to and including EF 6.*x*) support using an entity designer to create an entity data model XML (EDMX) file. Starting with version 4.1, EF added support for plain old CLR objects (POCO) using a technique referred to as Code First. EF Core will support only the Code First paradigm, dropping all EDMX support. For this reason (and a host of other issues with the EDMX paradigm), this chapter uses only the Code First paradigm. The Code First name is actually terrible, since it gives the impression that you can't use it with an existing database. I prefer the term Code Centric, but Microsoft didn't ask my opinion!

A. Troelsen and P. Japikse, *Pro C# 7*, https://doi.org/10.1007/978-1-4842-3018-3_22

Understanding the Role of the Entity Framework

ADO.NET provides you with a fabric that lets you select, insert, update, and delete data with connections, commands, and data readers. While this is all well and good, these aspects of ADO.NET force you to treat the fetched data in a manner that is tightly coupled to the physical database schema. Recall, for example, that when you use the connected layer, you typically iterate over each record by specifying column names to a data reader.

When you use ADO.NET, you must always be mindful of the physical structure of the back-end database. You must know the schema of each data table, author potentially complex SQL queries to interact with said data table(s), and so forth. This can force you to author some fairly verbose C# code because C# itself does not speak the language of the database schema directly.

To make matters worse, the way in which a physical database is usually constructed is squarely focused on database constructs such as foreign keys, views, stored procedures, and data normalization, not object-oriented programming.

The Entity Framework lessens the gap between the goals and optimization of the database and the goals and optimization of object-oriented programming. Using EF, you can interact with a relational database without ever seeing a line of SQL code (if you so choose). Rather, when you apply LINQ queries to your strongly typed classes, the EF runtime generates proper SQL statements on your behalf.

■ **Note** *LINQ to Entities* is the term that describes the act of applying LINQ queries to ADO.NET EF entity objects.

EF is simply another approach to the data-access APIs and is not necessarily intended to completely replace using ADO.NET directly from C# code. However, once you spend some time working with EF, you will quickly find yourself preferring this rich object model over creating all of your data access code from scratch. EF is another tool in your toolbox, and only you can decide what approach works best for your project.

■ **Note** You might recall a database programming API introduced with .NET 3.5 called LINQ to SQL. This API is close in concept (and fairly close in terms of programming constructs) to EF. LINQ to SQL is in maintenance mode, meaning it will receive only critical bug fixes. If you have an application using LINQ to SQL, know that Microsoft's official policy is to support all software for at least ten years after its "end of life." So while it won't be removed from your machine by the software guardians, the official word from those kind folks in Redmond is that you should put your efforts into EF, not LINQ to SQL. They certainly are focusing on EF.

The Role of Entities

The strongly typed classes mentioned previously (and demonstrated with the `Car` class in the previous chapter) are officially called *entities*. Entities are a conceptual model of a physical database that maps to your business domain. Formally speaking, this model is termed an *entity data model* (EDM). The EDM is a client-side set of classes that are mapped to a physical database by Entity Framework convention and configuration. You should understand that the entities need not map directly to the database schema, and typically they don't. You are free to structure your entity classes to fit your application needs and then map your unique entities to your database schema.

■ **Note** In the Code First world, most people refer to the POCO classes as *models* and the collection of these classes as the *object graph*. When the model classes are instantiated with data from the data store, they are then referred to as *entities*. In reality, the terms are pretty much used interchangeably, and both will be used throughout this text.

For example, take the simple `Inventory` table in the `AutoLot` database and the `Car` model class from the previous chapter. Through model configuration, you inform EF that the `Car` model represents the `Inventory` table. This loose coupling means you can shape the entities so they closely model your business domain.

■ **Note** In many cases, the model classes will be identically named to the related database tables. However, remember that you can always reshape the model to match your business situation.

You will build a full example with EF in just a bit. However, for the time being, consider the following `Program` class, which uses the `Car` model class and a related context class (covered soon) named `AutoLotEntities` to add a new row to the `Inventory` table of `AutoLot`.

```
class Program
{
  static void Main(string[] args)
  {
    // Connection string automatically read from config file.
    using (AutoLotEntities context = new AutoLotEntities())
    {
        // Add a new record to Inventory table, using our model.
    context.Cars.Add(new Car() { Color = "Black",
                                 Make = "Pinto",
                                 PetName = "Pete" });
    context.SaveChanges();
    }
  }
}
```

EF examines the configuration of your models and your context class to take the client-side representation of the `Inventory` table (here, a class named `Car`) and map it back to the correct columns of the `Inventory` table. Notice that you see no trace of any sort of SQL `INSERT` statement. You simply add a new `Car` object to the collection maintained by the aptly named `Cars` property of the context object and then save your changes.

There is no magic in the preceding example. Under the covers, a connection to the database is made and opened, a proper SQL statement is generated and executed, and the connection is released and closed. These details are handled on your behalf by the framework. Now let's look at the core services of EF that make this possible.

The Building Blocks of the Entity Framework

EF (at its core) uses the ADO.NET infrastructure you have already examined in the previous chapter. Like any ADO.NET interaction, EF uses an ADO.NET data provider to communicate with the data store. However, the data provider must be updated so it supports a new set of services before it can interact with the EF API. As you might expect, the Microsoft SQL Server data provider has been updated with the necessary infrastructure, which is accounted for when using the `System.Data.Entity.dll` assembly.

■ **Note** Many third-party databases (e.g., Oracle and MySQL) provide EF-aware data providers. If you are not using SQL Server, consult your database vendor for details or navigate to https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/ado-net-overview for a list of known ADO.NET data providers.

In addition to adding the necessary bits to the Microsoft SQL Server data provider, the `System.Data.Entity.dll` assembly contains various namespaces that account for the EF services themselves. The first key piece of the EF API to concentrate on is the `DbContext` class. You will create a derived, model-specific context when you use EF for your data access libraries.

The Role of the DbContext Class

The `DbContext` class represents a combination of the Unit of Work and Repository patterns that can be used to query from a database and group together changes that will be written back as a single unit of work. `DbContext` provides a number of core services to child classes, including the ability to save all changes (which results in a database update), tweak the connection string, delete objects, call stored procedures, and handle other fundamental details. Table 22-1 shows some of the more commonly used members of the `DbContext`.

Table 22-1. *Common Members of* `DbContext`

Member of **DbContext**	Meaning in Life
`DbContext`	Constructor used by default in the derived context class. The string parameter is either the database name or the connection string stored in the `*.config` file.
`Entry` `Entry<TEntity>`	Retrieves the `System.Data.Entity.Infrastructure.DbEntityEntry` object providing access to information and the ability to perform actions on the entity.
`GetValidationErrors`	Validates tracked entries and returns a collection of `System.Data.Entity.Validation.DbEntityValidationResults`.
`SaveChanges` `SaveChangesAsync`	Saves all changes made in this context to the database. Returns the number of affected entities.
`Configuration`	Provides access to the configuration properties of the context.
`Database`	Provides a mechanism for creation/deletion/existence checks for the underlying database, executes stored procedures and raw SQL statements against the underlying data store, and exposes transaction functionality.

`DbContext` also implements `IObjectContextAdapter`, so any of the functionality available in the `ObjectContext` class is also available. While `DbContext` takes care of most of your needs, there are two events that can be extremely helpful, as you will see later in the chapter. Table 22-2 lists the events.

Table 22-2. *Events in* `DbContext`

Events of **DbContext**	Meaning in Life
`ObjectMaterialized`	Fires when a new entity object is created from the data store as part of a query or load operation
`SavingChanges`	Occurs when changes are being saved to the data store but prior to the data being persisted

The Role of the Derived Context Class

As mentioned, the DbContext class provides the core functionality when working with EF Code First. In your projects, you will create a class that derives from DbContext for your specific domain. In the constructor, you need to pass the name of the connection string for this context class to the base class, as shown here:

```
public class AutoLotEntities : DbContext
{
  public AutoLotEntities() : base("name=AutoLotConnection")
  {
  }
  protected override void Dispose(bool disposing)
  {
  }
}
```

The Role of DbSet<T>

To add tables into your context, you add a DbSet<T> for each table in your object model. To enable lazy loading, the properties in the context need to be virtual, like this:

```
  public virtual DbSet<CreditRisk> CreditRisks { get; set; }
  public virtual DbSet<Customer> Customers { get; set; }
  public virtual DbSet<Inventory> Inventory { get; set; }
  public virtual DbSet<Order> Orders { get; set; }
```

Each DbSet<T> provides a number of core services to each collection, such as creating, deleting, and finding records in the represented table. Table 22-3 describes some of the core members of the DbSet<T> class.

Table 22-3. *Common Members of DbSet<T>*

Member of **DbSet<T>**	Meaning in Life
Add AddRange	Allows you to insert a new object (or range of objects) into the collection. They will be marked with the Added state and will be inserted into the database when SaveChanges (or SaveChangesAsync) is called on the DbContext.
Attach	Associates an object with the DbContext. This is commonly used in disconnected applications like ASP.NET/MVC.
Create Create<T>	Creates a new instance of the specified entity type.
Find FindAsync	Finds a data row by the primary key and returns an object representing that row.
Remove RemoveRange	Marks an object (or range of objects) for deletion.
SqlQuery	Creates a raw SQL query that will return entities in this set.

Once you drill into the correct property of the object context, you can call any member of DbSet<T>. Consider again the sample code shown in the first few pages of this chapter:

```
using (AutoLotEntities context = new AutoLotEntities())
{
  // Add a new record to Inventory table, using our entity.
  context.Cars.Add(new Car() { ColorOfCar = "Black",
                                    MakeOfCar = "Pinto",
                                    NicknameOfCar = "Pete" });
  context.SaveChanges();
}
```

Here, AutoLotEntities *is-a* derived DbContext. The Cars property gives you access to the DbSet<Car> variable. You use this reference to insert a new Car entity object and tell the DbContext to save all changes to the database.

DbSet<T> is typically the target of LINQ to Entity queries; as such, DbSet<T> supports the same extension methods you learned about in Chapter 12, such as ForEach(), Select(), and All(). Moreover, DbSet<T> gains a good deal of functionality from its direct parent class, DbQuery<T>, which is a class that represents a strongly typed LINQ (or Entity SQL) query.

Code First Explained

As mentioned in a previous note, Code First doesn't mean you can't use EF with an existing database. It really just means no EDMX model. I prefer the term Code Centric, since that's what it really means. You can use Code First from an existing database or create a new database from the entities using EF migrations.

Transaction Support

All versions of EF wrap each call to SaveChanges/SaveChangesAsync in a transaction. The isolation level of these automatic transactions is the same as the default isolation level for the database (which is READ COMMITTED for SQL Server). You can add more control to the transactional support in EF if you need it. For more information, see https://msdn.microsoft.com/en-us/data/dn456843.aspx?f=255&MSPPError=-2147217396.

■ **Note** Although not covered in this book, executing SQL statements using ExecuteSqlCommand() from the DbContext database object is now wrapped in an implicit transaction. This is new in EF version 6.

Entity State and Change Tracking

The DbChangeTracker automatically tracks the state for any object loaded into a DbSet<T> within a DbContext. In the previous examples, while inside the using statement, any changes to the data will be tracked and saved when SaveChanges is called on the AutoLotEntities class. Table 22-4 lists the possible values for the state of an object.

Table 22-4. *Entity State Enumeration Values*

Value	Meaning in Life
`Detached`	The object exists but is not being tracked. An entity is in this state immediately after it has been created and before it is added to the object context.
`Unchanged`	The object has not been modified since it was attached to the context or since the last time that the `SaveChanges()` method was called.
`Added`	The object is new and has been added to the object context, and the `SaveChanges()` method has not been called.
`Deleted`	The object has been deleted from the object context but not yet removed from the data store.
`Modified`	One of the scalar properties on the object was modified, and the `SaveChanges()` method has not been called.

If you need to check the state of an object, use the following code:

```
EntityState state = context.Entry(entity).State;
```

You usually don't need to worry about the state of your objects. However, in the case of deleting an object, you can set the state of an object to `EntityState.Deleted` and save a round-trip to the database. You will do this later in the chapter.

Entity Framework Data Annotations

Data annotations are C# attributes that are used to shape your entities. Table 22-5 lists some of the most commonly used data annotations for defining how your entity classes and properties map to database tables and fields. There are many more annotations that you can use to refine your model and add validations, as you will see throughout the rest of this chapter and book.

Table 22-5. *Data Annotations Supported by the Entity Framework*

Data Annotation	Meaning in Life
`Key`	Defines the primary key for the model. This is not necessary if the key property is named `Id` or combines the class name with `Id`, such as `OrderId`. If the key is a composite, you must add the `Column` attribute with an `Order`, such as `Column[Order=1]` and `Column[Order=2]`. Key fields are implicitly also `[Required]`.
`Required`	Declares the property as not nullable.
`ForeignKey`	Declares a property that is used as the foreign key for a navigation property.
`StringLength`	Specifies the min and max lengths for a string property.
`NotMapped`	Declares a property that is not mapped to a database field.
`ConcurrencyCheck`	Flags a field to be used in concurrency checking when the database server does updates, inserts, or deletes.
`TimeStamp`	Declares a type as a row version or timestamp (depending on the database provider).

(*continued*)

Table 22-5. *(continued)*

Data Annotation	Meaning in Life
`Table` `Column`	Allows you to name your model classes and fields differently than how they are declared in the database. The `Table` attribute allows specification of the schema as well (as long as the data store supports schemas).
`DatabaseGenerated`	Specifies if the field is database generated. This takes one of `Computed`, `Identity`, or `None`.
`NotMapped`	Specifies that EF needs to ignore this property in regard to database fields.
`Index`	Specifies that a column should have an index created for it. You can specify clustered, unique, name, and order.

Code First from an Existing Database

Now that you have a better understanding of what the ADO.NET Entity Framework is and how it works from a high level, it's time to look at your first full example. You will build a simple console app that uses Code First from an existing database to create the model classes representing the existing `AutoLot` database you built in Chapter 21.

Generating the Model

Begin by creating a new Console Application project named AutoLotConsoleApp. Add a folder to the project through the Project ➤ New Folder menu option and name it `EF`. Select the new `EF` folder and then select Project ➤ Add New Item (be sure to highlight the Data node) to insert a new ADO.NET Entity Data Model item named `AutoLotEntities` (as in Figure 22-1).

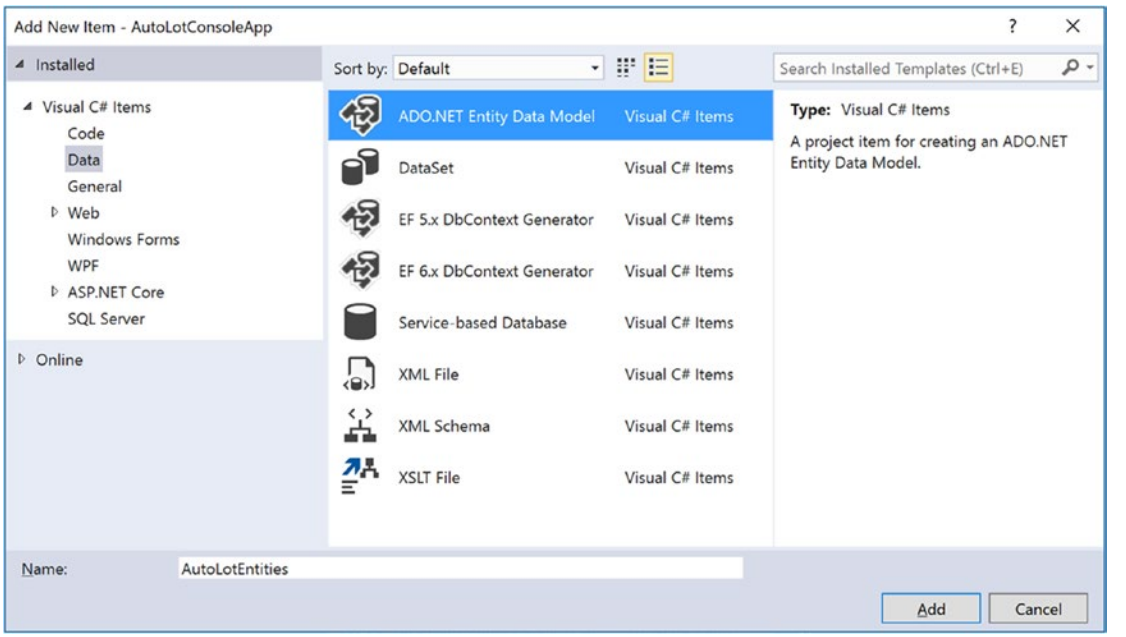

Figure 22-1. *Inserting a new ADO.NET EDM project item*

Clicking the Add button launches the Entity Model Data Wizard. The wizard's first step allows you to select the option to generate an EDM using the Entity Framework Designer (from an existing database or by creating an empty designer) or using Code First (from an existing database or by creating an empty `DbContext`). Select the "Code First from database" option and click the Next button (see Figure 22-2).

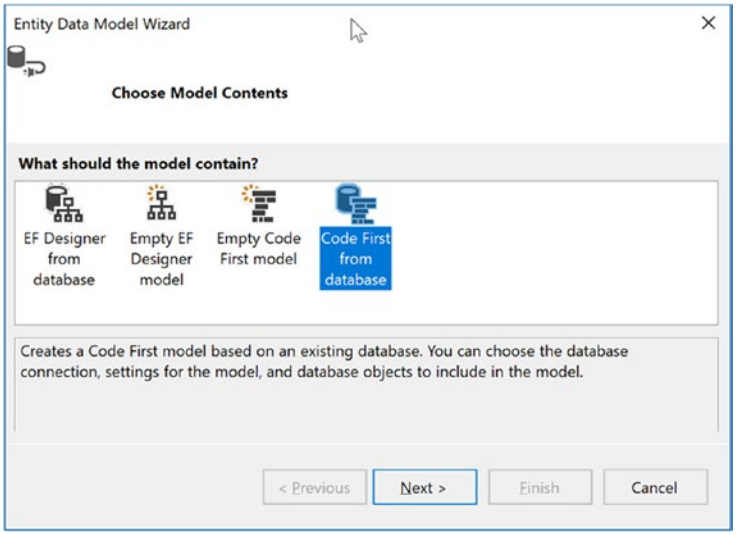

Figure 22-2. *Generating an EDM from an existing database*

The Choose Your Database Connection screen will autopopulate with any connection strings stored in Visual Studio. If you already have a connection to your database within the Visual Studio Server Explorer, you will see it listed in the drop-down combo box (as shown in Figure 22-5). If this is not the case, click the New Connection button.

This loads the Choose Data Source screen. Select Microsoft SQL Server and click Continue, as shown in Figure 22-3.

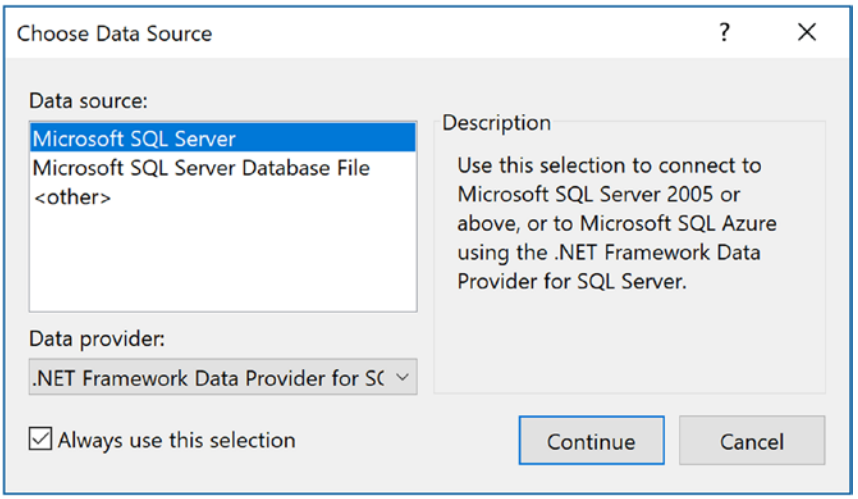

Figure 22-3. *Select SQL Server for the new connection string*

On the next screen, select `(localdb)\mssqllocaldb` for the server and then select `AutoLot` for the database, as shown in Figure 22-4.

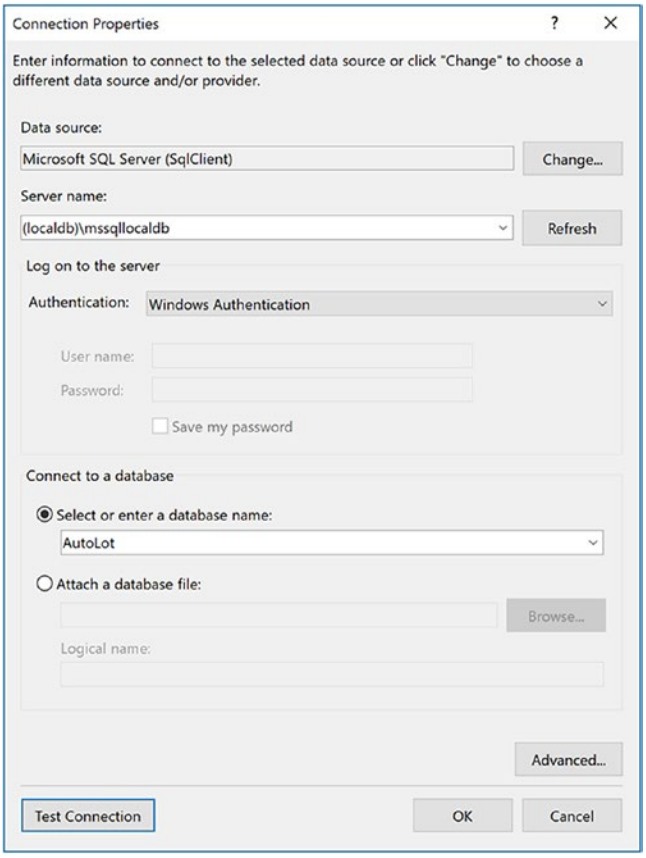

Figure 22-4. *Creating the connection string to AutoLot*

After clicking OK, your new connection string will be created and selected on the Choose Your Data Connection screen. Make sure the box to save the connection string is selected and set the `App.config` setting to `AutoLotConnection`, as shown in Figure 22-5.

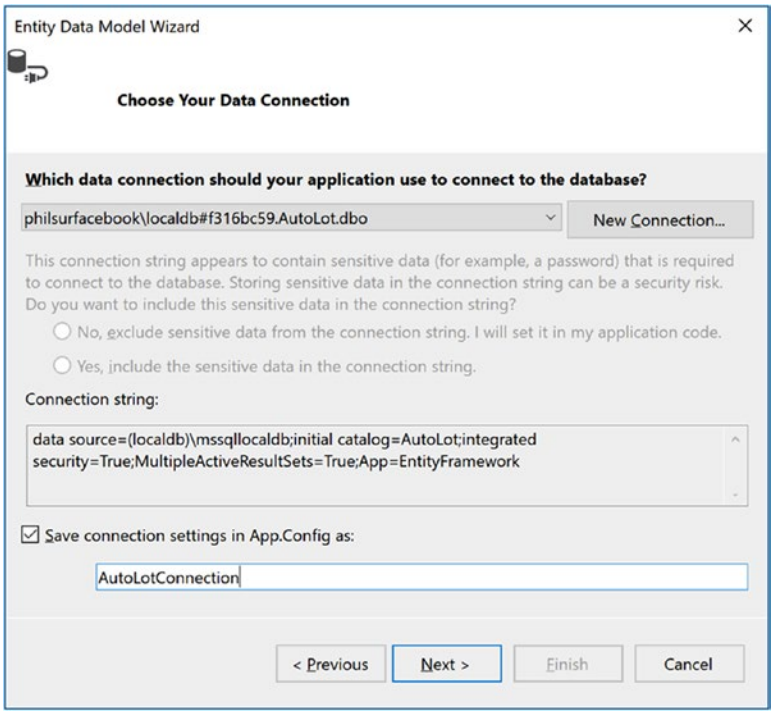

Figure 22-5. *Selecting the database used to generate the model*

Before you click the Next button, take a moment to examine the format of your connection string.

```
Data source= (localdb)\mssqllocaldb;Initial Catalog=AutoLot;Integrated Security=True;
MultipleActiveResultSets=true;App=EntityFramework
```

This is similar to what you used in Chapter 21, with the addition of the `App=EntityFramework` name-value pair. `App` is short for application name, which can be used when troubleshooting SQL Server issues.

In the wizard's final step, you select the items from the database you want generated into the EDM. Select all the application tables, making sure you don't select `sysdiagrams` (if it exists in your database), as shown in Figure 22-6. Click the Finish button to generate your models and the derived `DbContext`.

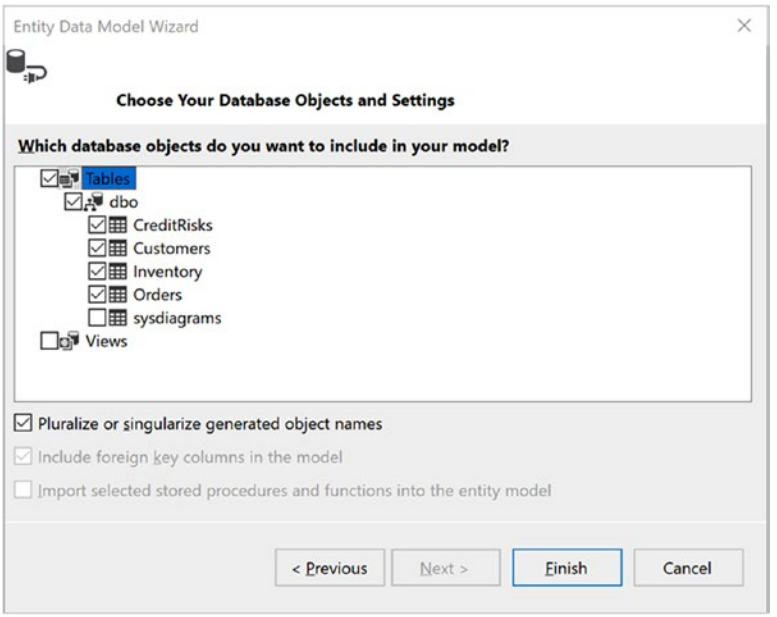

Figure 22-6. *Selecting the database items*

What Did That Do?

After you complete the wizard, you will see several new classes in your project: one for each table that you selected in the wizard and another one named `AutoLotEntities` (the same name you entered in the first step of the wizard). By default, the names of your entity classes and properties will match the original database object names. Using the data annotations listed in Table 21-5, you can change the entity names, as well as property names, to something else if needed (or wanted). Later in this chapter you will use data annotations to make some modifications to your entity classes.

■ **Note** The Fluent API is another way to configure your model classes and properties to map them to the database. Everything you can do with data annotations, you can also do with code through the Fluent API. Because of space and time constraints, I focus on covering data annotations in this chapter with only a brief mention of the Fluent API.

Open the `Inventory` class and examine the attributes on the class and the properties. At the class level, the `Table` attribute specifies what database table the class maps to. At the property level, there are two attributes in use. The first you see is the `Key` attribute, which specifies the primary key for the table. The other attribute is `StringLength`, which specifies the string length for the database field.

■ **Note** There are also two `SuppressMessage` attributes. This instructs static analyzers such as FXCop and the new Roslyn code analyzers to turn off the specific rules listed in the constructor. These are not covered in this chapter, and for clarity, I've removed them from the sample code.

```
[Table("Inventory")]
public partial class Inventory
{
  public Inventory()
  {
    Orders = new HashSet<Order>();
  }

  [Key]
  public int CarId { get; set; }

  [StringLength(50)]
  public string Make { get; set; }

  [StringLength(50)]
  public string Color { get; set; }

  [StringLength(50)]
  public string PetName { get; set; }

  public virtual ICollection<Order> Orders { get; set; }
}
```

You can also see that the `Inventory` class has a collection of `Order` objects and that the `Order` class contains a `Car` property. These are navigation properties and will be covered shortly.

```
public partial class Order
{
  public int OrderId { get; set; }

  public int CustId { get; set; }

  public int CarId { get; set; }

  public virtual Customer Customer { get; set; }

  public virtual Inventory Inventory { get; set; }
}
```

Next, open the `AutoLotEntities` class. This class derives from the `DbContext` class and contains a `DbSet<TEntity>` property for each table that you specified in the wizard. It also overrides `OnModelCreating()` to use `FluentAPI` to define the relationships between the `Orders` and `Inventory` tables.

```
public partial class AutoLotEntities : DbContext
{
  public AutoLotEntities()
    : base("name=AutoLotConnection")
  {
  }

  public virtual DbSet<CreditRisk> CreditRisks { get; set; }
  public virtual DbSet<Customer> Customers { get; set; }
  public virtual DbSet<Inventory> Inventories { get; set; }
  public virtual DbSet<Order> Orders { get; set; }
```

```
  protected override void OnModelCreating(DbModelBuilder modelBuilder)
  {
    modelBuilder.Entity<Inventory>()
      .HasMany(e => e.Orders)
      .WithRequired(e => e.Inventory)
      .WillCascadeOnDelete(false);
  }
}
```

Finally, open the App.config file. You will see a new configSection (named entityFramework), as well as the connection string generated by the wizard. Most of this you can ignore, but if you change the database location, you will need to modify the connection string to the new database.

```
<configuration>
  <configSections>
    <!-- For more information on Entity Framework configuration, visit http://go.microsoft.
    com/fwlink/?LinkID=237468 -->
    <section name="entityFramework" type="System.Data.Entity.Internal.ConfigFile.
    EntityFrameworkSection, EntityFramework, Version=6.0.0.0, Culture=neutral, PublicKeyToke
    n=b77a5c561934e089" requirePermission="false" />
  </configSections>
  <startup>
    <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.6" />
  </startup>
  <entityFramework>
    <defaultConnectionFactory type="System.Data.Entity.Infrastructure.SqlConnectionFactory,
    EntityFramework" />
    <providers>
      <provider invariantName="System.Data.SqlClient" type="System.Data.Entity.SqlServer.
      SqlProviderServices, EntityFramework.SqlServer" />
    </providers>
  </entityFramework>
  <connectionStrings>
    <add name="AutoLotConnection" connectionString="data source=(localdb)\mssqllocaldb;
    initial catalog=AutoLot;integrated security=True;MultipleActiveResultSets=True;
    App=EntityFramework" providerName="System.Data.SqlClient" />
  </connectionStrings>
</configuration>
```

Changing the Default Mappings

The [Table("Inventory")] class-level data annotation maps the class to the Inventory table in the database, regardless of what the actual name of the class. Change the file name and class name (and the constructor) to Car. The [Column("PetName")] attribute maps the decorated C# property to the PetName field on the table, allowing you to change the C# property name to anything you want, for example to the name CarNickName. The updated class looks like this:

```
[Table("Inventory")]
public partial class Car
{
  public Car()
  {
```

```
    Orders = new HashSet<Order>();
  }

  [StringLength(50), Column("PetName")]
  public string CarNickName { get; set; }

  //remainder of the class not shown for brevity
}
```

Note that you will also have to change the type of the `Inventory` property to `Car` in the `Order` class. You can also change the property name, as shown here:

```
public partial class Order
{
  public virtual Car Car { get; set; }

  //remainder of the class not shown for brevity
}
```

The final change to make is to the `AutoLotEntities` class. Open the file and change the two occurrences of `Inventory` to `Car` and `DbSet<Car>` to `Cars`. The updated code is shown here:

```
public partial class AutoLotEntities : DbContext
{
  public AutoLotEntities()
    : base("name=AutoLotConnection")
  {
  }

  // Additional code removed for brevity

  public virtual DbSet<Car> Cars { get; set; }

  protected override void OnModelCreating(DbModelBuilder modelBuilder)
  {
    modelBuilder.Entity<Car>()
      .HasMany(e => e.Orders)
      .WithRequired(e => e.Car
      .WillCascadeOnDelete(false);

    // Additional code removed for brevity
  }
}
```

Adding Features to the Generated Model Classes

All the designer-generated classes have been declared with the `partial` keyword. This is especially useful when working with the EF programming model because you can add additional methods to your entity classes that help you model your business domain better.

For example, you can override the `ToString()` method of the `Car` entity class to return the state of the entity with a well-formatted `string`. If you add this to the generated class, you risk losing that custom code

each time you regenerate your model classes. Instead, define the following partial class declaration in a new file named CarPartial.cs. The new class is listed here:

```
public partial class Car
{
  public override string ToString()
  {
    // Since the PetName column could be empty, supply
    // the default name of **No Name**.
    return $"{this.CarNickName ?? "**No Name**"} is a {this.Color} {this.Make} with ID
{this.CarId}.";
  }
}
```

Using the Model Classes in Code

Now that you have your model classes, you can author some code that interacts with them and therefore the database. Begin by adding the following `using` statements to your `Program` class:

```
using AutoLotConsoleApp.EF;
using AutoLotConsoleApp.Models;
using static System.Console;
```

Inserting Data

Once the entity classes and properties are mapped to the database tables and fields, when changes to the data in the `DbSet<T>` collections need to be persisted, EF will generate the SQL to make the changes. Adding new records is as simple as adding records to the `DbSet<T>` and calling `SaveChanges()`. You can either add one record at a time or add a range of records.

Inserting a Record

To add a new record, create a new instance of the model class, add it to the appropriate `DbSet<T>` property from the `AutoLotEntities` context class, and then call `SaveChanges()`. Add a helper method to the `Program.cs` class named `AddNewRecord()` with the following code:

```
private static int AddNewRecord()
{
  // Add record to the Inventory table of the AutoLot database.
  using (var context = new AutoLotEntities())
  {
    try
    {
      // Hard-code data for a new record, for testing.
      var car = new Car() { Make = "Yugo", Color = "Brown", CarNickName="Brownie"};
      context.Cars.Add(car);
      context.SaveChanges();
      // On a successful save, EF populates the database generated identity field.
      return car.CarId;
    }
```

```
      catch(Exception ex)
      {
        WriteLine(ex.InnerException?.Message);
        return 0;
      }
    }
}
```

This code uses the `Add()` method on the `DbSet<Car>` class. The `Add()` method takes an object of type `Car` and adds it to the `Cars` collection on the `AutoLotEntities` context class. When an object is added to a `DbSet<T>`, the `DbChangeTracker` marks the state of the new object as `EntityState.Added`. When `SaveChanges` is called on the `DbContext`, SQL statements are generated for all pending changes tracked in the `DbChangeTracker` (in this case an `insert` statement) and executed against the database. If there was more than one change, they would be executed within a transaction. If no errors occur, then the changes are persisted to the database, and any database-generated properties (in this case the `CarId` property) get updated with the values from the database.

To see this in action, update the `Main()` method like this:

```
static void Main(string[] args)
{
  WriteLine("***** Fun with ADO.NET EF *****\n");
  int carId = AddNewRecord();
  WriteLine(carId);
  ReadLine();
}
```

The output to the console is indeed the `CarId` of the new record. When new records are added (or existing records are updated), EF executes a `SELECT` statement for the values of database-computed columns on your behalf to get the value and populate your entity's properties.

Inserting Multiple Records

In addition to adding a single record at a time, you add multiple records in one call with the `AddRange()` method. This method takes an `IEnumerable<T>` and adds all the items in the list to the `DbSet<T>`.

```
private static void AddNewRecords(IEnumerable<Car> carsToAdd)
{
  using (var context = new AutoLotEntities())
  {
    context.Cars.AddRange(carsToAdd);
    context.SaveChanges();
  }
}
```

Even though all the records are now in the `DbSet<T>`, just like when adding one record, nothing happens to the database until `SaveChanges()` is called. Consider `AddRange()` as a convenience method. You could iterate through the collection, calling `Add()` for each item in the collection, and then call `SaveChanges()`, which ends in the same result as calling `AddRange()` and then `SaveChanges()`.

Selecting Records

There are several ways to get records out of the database using EF. The simplest way to get the data from the database is to iterate over a `DbSet<Car>`. This is equivalent to executing the following SQL command:

```
Select * from Inventory
```

To see this in action, add a new method named `PrintAllInventory()` and loop through the `Cars` property of the `DbContext`, printing each car (using the overridden `ToString()` method), as follows:

```
private static void PrintAllInventory()
{
    // Select all items from the Inventory table of AutoLot,
    // and print out the data using our custom ToString() of the Car entity class.
    using (var context = new AutoLotEntities())
    {
    foreach (Car c in context.Cars)
      {
        WriteLine(c);
    }
  }
}
```

Behind the scenes, EF retrieves all the `Inventory` records from the database and, using a `DataReader`, creates a new instance of the `Car` class for each row returned from the database.

Querying with SQL from DbSet<T>

You can also use inline SQL or stored procedures to retrieve entities from the database. The caveat is that the query fields must match the properties of the class being populated. To test this, update the `PrintInventory()` method to the following:

```
private static void PrintAllInventory()
{
  using (var context = new AutoLotEntities())
  {
    foreach (Car c in context.Cars.SqlQuery("Select CarId,Make,Color,PetName as CarNickName
from Inventory where Make=@p0", "BMW"))
    {
      WriteLine(c);
    }
  }
}
```

The good news is that when called on a `DbSet<T>`, this method fills the list with tracked entities, which means that any changes or deletions will get propagated to the database when `SaveChanges` is called. The bad news (as you can see from the SQL text) is that `SqlQuery` doesn't understand the mapping changes that you made earlier. Not only do you have to use the database table and field names, but any field name changes (such as the change to `PetName`) must be aliased from the database field name to the model property name.

Querying with SQL from DbContext.Database

The Database property of the DbContext also has a SqlQuery method that can be used to populate DbSet<T> entities as well as objects that are not part of your model (such as a view model). Suppose you have another class (named ShortCar) that is defined as follows:

```
public class ShortCar
{
  public int CarId { get; set; }
  public string Make { get; set; }
  public override string ToString() => $"{this.Make} with ID {this.CarId}.";
}
```

You can create and populate a list of ShortCar objects with the following code:

```
foreach (ShortCar c in context.Database.SqlQuery(typeof(ShortCar),"Select CarId,Make from
Inventory"))
{
  WriteLine(c);
}
```

It's important to understand that when calling SqlQuery on the Database object, the returned classes are *never* tracked, even if the objects are defined as a DbSet<T> on your context.

Querying with LINQ

EF becomes much more powerful when you incorporate LINQ queries. Consider this update to the PrintInventory() method that uses LINQ to get the records from the database:

```
private static void PrintAllInventory()
{
    foreach (Car c in context.Cars.Where(c => c.Make == "BMW"))
    {
      WriteLine(c);
    }
}
```

The LINQ statement is translated into a SQL query similar to the following:

```
Select * from Inventory where Make = 'BMW';
```

Given that you have already worked with many LINQ expressions in Chapter 13, a few more examples will suffice for the time being.

```
private static void FunWithLinqQueries()
{
  using (var context = new AutoLotEntities())
  {
    // Get a projection of new data.
    var colorsMakes = from item in context.Cars select new { item.Color, item.Make };
    foreach (var item in colorsMakes)
```

```
    {
      WriteLine(item);
    }

    // Get only items where Color == "Black"
    var blackCars = from item in context.Cars where item.Color == "Black" select item;
    foreach (var item in blackCars)
    {
      WriteLine(item);
    }
  }
}
```

Searching with Find()

`Find()` is a special LINQ method, in that it first looks in the `DbChangeTracker` for the entity being requested and, if found, returns that instance to the calling method. If it isn't found, EF does a database call to create the requested instance.

`Find()` only searches by primary key (simple or complex), so you must keep that in mind when using it. An example of `Find()` in action looks like this:

```
WriteLine(context.Cars.Find(5));
```

The Timing of EF Query Execution

Each of the previous methods returns data from the database. It's important to understand *when* the queries execute. One of the beauties of EF is that select queries don't execute until the resulting collection is iterated over. The query will also execute once `ToList()` or `ToArray()` is called on the query. This enables chaining of calls and building up a query as well as controlling exactly when the first database call is made. However, with lazy loading, you give up control over when the queries are made on navigation properties that weren't loaded with the initial call.

Take the following updated version of the `FunWithLinqQueries()` method. The first two lines are setting up the query. Not until the collection is iterated over with the `foreach` is the database call made.

```
private static void ChainingLinqQueries()
{
  using (var context = new AutoLotEntities())
  {
    //Not executed
    DbSet<Car> allData = context.Cars;

    //Not Executed.
    var colorsMakes = from item in allData select new { item.Color, item.Make };

    //Now it's executed
    foreach (var item in colorsMakes)
    {
      WriteLine(item);
    }
  }
}
```

The Role of Navigation Properties

As the name suggests, *navigation properties* allow you to find related data in other entities without having to author complex JOIN queries. In SQL Server, navigation properties are represented by foreign key relationships. To account for these foreign key relationships in EF, each class in your model contains virtual properties that connect your classes together. For example, in the `Inventory.cs` class, the `Orders` property is defined as `virtual ICollection<Order>`.

```
public virtual ICollection<Order> Orders { get; set; }
```

This tells EF that each `Inventory` database record (renamed to the `Car` class in the examples) can have zero to many `Order` records.

On the other side of the equation, the `Order` model has a one-to-one relationship with the `Inventory` (`Car`) record. The `Order` model navigates back to the `Inventory` model through another `virtual` property of type `Inventory`.

```
public virtual Car Car { get; set; }
```

In SQL Server, foreign keys are properties that tie tables together. In this example, the `Orders` table has a foreign key named `CarId`. In the `Orders` model, this is represented by the following property:

```
public int CarId { get; set; }
```

■ **Note** If the `CarId` foreign key were a nullable `int`, then it would have a zero-to-one relationship.

By convention, if EF finds a property named `<Class>Id`, then it will be used as the foreign key for a navigation property of type `<Class>`. As with any other name of classes and properties, this can be changed. For example, if you wanted to name the property `Foo`, you would update the class to this:

```
public partial class Order
{
  [Column("CarId")]
  public int Foo { get; set; }
  [ForeignKey(nameof(Foo))]
//rest of the class omitted for brevity
}
```

Lazy, Eager, and Explicit Loading

Loading data from the database into entity classes can happen three different ways: lazy, eager, and explicit. Lazy and eager loading are based on settings on the context, and the third, explicit, is developer controlled.

Lazy Loading

Lazy loading means that EF loads the direct object(s) requested but replaces any navigation properties with proxies. The `virtual` modifier allows the proxy assignment to the navigation properties. If any of the properties on a navigation property are requested in code, EF creates a new database call, executes it, and populates the object's details. For example, if you had the following code, EF would call one query to get all the `Cars` and then for each `Car` execute another query to get all the `Orders` for each car:

```
using (var context = new AutoLotEntities())
{
  foreach (Car c in context.Cars)
  {
    foreach (Order o in c.Orders)
    {
      WriteLine(o.OrderId);
    }
  }
}
```

While it is a plus to only load the data you need, you can see from the previous example that it might become a performance nightmare if you aren't careful about how lazy loading is utilized. You can turn off lazy loading by setting the `LazyLoadingEnabled` property on the `DbContext` configuration, like this:

```
context.Configuration.LazyLoadingEnabled = false;
```

If you know that you need to orders for each car, then you should consider using eager loading, covered next.

Eager Loading

When you know you want to load related records for an object, writing multiple queries against a relational database is inefficient. The prudent database developer would write one query utilizing SQL joins to get the related data. Eager loading does just that for C# developers using EF. For example, if you knew you needed all `Cars` and all of their related `Orders`, you would change the previous code to this:

```
using (var context = new AutoLotEntities())
{
  foreach (Car c in context.Cars.Include(c=>c.Orders))
  {
    foreach (Order o in c.Orders)
    {
      WriteLine(o.OrderId);
    }
  }
}
```

The `Include()` LINQ method informs EF to create a SQL statement that joins the tables together, just as you would if you were writing the SQL yourself. The resulting query executed against the database now resembles this:

```
SELECT
    [Project1].[CarId] AS [CarId],
    [Project1].[Make] AS [Make],
    [Project1].[Color] AS [Color],
    [Project1].[PetName] AS [PetName],
    [Project1].[C1] AS [C1],
    [Project1].[OrderId] AS [OrderId],
    [Project1].[CustId] AS [CustId],
    [Project1].[CarId1] AS [CarId1]
    FROM ( SELECT
```

```
        [Extent1].[CarId] AS [CarId],
        [Extent1].[Make] AS [Make],
        [Extent1].[Color] AS [Color],
        [Extent1].[PetName] AS [PetName],
        [Extent2].[OrderId] AS [OrderId],
        [Extent2].[CustId] AS [CustId],
        [Extent2].[CarId] AS [CarId1],
        CASE WHEN ([Extent2].[OrderId] IS NULL) THEN CAST(NULL AS int) ELSE 1 END AS [C1]
        FROM  [dbo].[Inventory] AS [Extent1]
        LEFT OUTER JOIN [dbo].[Orders] AS [Extent2] ON [Extent1].[CarId] = [Extent2].[CarId]
    )  AS [Project1]
    ORDER BY [Project1].[CarId] ASC, [Project1].[C1] ASC
```

The exact syntax of the query doesn't really matter; I've shown it to demonstrate that all `Cars` and their related `Orders` are getting retrieved in one call to the database.

Explicit Loading

The final way to load records is through explicit loading. Explicit loading allows you to explicitly load a collection or class at the end of a navigation property. To get the related object(s), you use the `Collection()` (for collections) or `Reference()` (for single objects) methods of `context` to choose what to load and then call `Load()`. The following code shows both methods in action:

```
context.Configuration.LazyLoadingEnabled = false;
foreach (Car c in context.Cars)
{
    context.Entry(c).Collection(x => x.Orders).Load();
    foreach (Order o in c.Orders)
    {
        WriteLine(o.OrderId);
    }
}
foreach (Order o in context.Orders)
{
  context.Entry(o).Reference(x => x.Car).Load();
}
```

Which data access model you use depends on the needs of your project. If you leave lazy loading enabled (the default setting), then you have to be careful that your application doesn't become too chatty. If you turn it off, you need to make sure you load related data before trying to use it.

Deleting Data

Deleting records from the database can be done using the `DbSet<T>` (conceptually the same as adding records) but can also be done using `EntityState`. You will see both of these approaches in action in the next sections.

Deleting a Single Record

One way to delete a single record is to locate the correct item in the `DbSet<T>` and then call `DbSet<T>.Remove()`, passing in that instance. Calling the `Remove` method removes the item from the collection and sets the `EntityState` to `EntityState.Deleted`. Even though it's removed from the `DbSet<T>`, it's not removed from the context (or the database) until `SaveChanges()` is called.

One catch in this process is that the instance to be removed must already be tracked (in other words, loaded into the DbSet<T>). One common pattern is used in MVC (covered later in this book), where the primary key of the item to be deleted is passed into the Delete action. This item is retrieved using Find(), then removed from the DbSet<T> with Remove(), and then persisted to the database with SaveChanges(), as in the following example:

```
private static void RemoveRecord(int carId)
{
  // Find a car to delete by primary key.
  using (var context = new AutoLotEntities())
  {
    // See if we have it.
    Car carToDelete = context.Cars.Find(carId);
    if (carToDelete != null)
    {
      context.Cars.Remove(carToDelete);
      //This code is purely demonstrative to show the entity state changed to Deleted
      if (context.Entry(carToDelete).State != EntityState.Deleted)
      {
        throw new Exception("Unable to delete the record");
      }
      context.SaveChanges();
    }
  }
 }
```

■ **Note** Calling Find() before deleting a record requires an extra round-trip to the database. First you pull the record back and then delete it. Deleting data can also be accomplished by changing the EntityState, which will be covered shortly.

Deleting Multiple Records

You can also remove multiple records at once by using RemoveRange() on the DbSet<T>. Just like the Remove() method, the items to be removed must be tracked.

The RemoveRange() method takes an IEnumerable<T> as a parameter, as shown in the following example:

```
private static void RemoveMultipleRecords(IEnumerable<Car> carsToRemove)
{
  using (var context = new AutoLotEntities())
  {
    //Each record must be loaded in the DbChangeTracker
    context.Cars.RemoveRange(carsToRemove);
    context.SaveChanges();
  }
}
```

Remember, though, that nothing happens to the database until SaveChanges() is called.

Deleting a Record Using EntityState

The final way to delete a record is by using `EntityState`. You start by creating a new instance of the item to be deleted, assigning the primary key to the new instance, and setting the `EntityState` to `EntityState.Deleted`. This adds the item to the `DbChangeTracker` for you, so when `SaveChanges()` is called, the record is deleted. Note that the record didn't have to be queried from the database first. The following code demonstrates this:

```
private static void RemoveRecordUsingEntityState(int carId)
{
  using (var context = new AutoLotEntities())
  {
    Car carToDelete = new Car() { CarId = carId };
    context.Entry(carToDelete).State = EntityState.Deleted;
    try
    {
      context.SaveChanges();
    }
    catch (DbUpdateConcurrencyException ex)
    {
      WriteLine(ex);
    }
  }
}
```

You gain performance (since you are not making an extra call to the database), but you lose the validation that the object exists in the database (if that matters to your scenario). If the `CarId` does not exist in the database, EF will throw a `DbUpdateConcurrencyException` in the `System.Data.Entity.Infrastructure` namespace. The `DbUpdateConcurrencyException` is covered in detail later in this chapter.

■ **Note** If an instance with the same primary key is already being tracked, this method will fail, since you can't have two of the same entities with the same primary key being tracked by the `DbChangeTracker`.

Updating a Record

Updating a record pretty much follows the same pattern. Locate the object you want to change, set new property values on the returned entity, and save the changes, like so:

```
private static void UpdateRecord(int carId)
{
  // Find a car to delete by primary key.
  using (var context = new AutoLotEntities())
  {
    // Grab the car, change it, save!
    Car carToUpdate = context.Cars.Find(carId);
    if (carToUpdate != null)
    {
      WriteLine(context.Entry(carToUpdate).State);
      carToUpdate.Color = "Blue";
      WriteLine(context.Entry(carToUpdate).State);
```

```
      context.SaveChanges();
    }
  }
}
```

Handling Database Changes

In this section, you created an EF solution that started with an existing database. This works great, for example, when your organization has dedicated DBAs and you are provided with a database that you don't control. As your database changes over time, all you need to do is run the wizard again and re-create your `AutoLotEntities` class; the model classes will be rebuilt for you as well. Make sure that any additions to the model classes are done using partial classes; otherwise, you will lose your work when you rerun the wizard.

This initial example should go a long way toward helping you understand the nuts and bolts of working with the Entity Framework.

■ **Source Code** You can find the AutoLotConsoleApp example in the `Chapter 22` subdirectory.

Creating the AutoLot Data Access Layer

In the previous section, you used a wizard to create the entities and context from an existing database. EF can also create your database for you based on your model classes and derived `DbContext` class. In addition to creating the initial database, EF enables you to use migrations to update your database to match any model changes. Even better, you can use the wizard to create your initial models and context and then switch to a C#-centric approach and use migrations to keep your database in sync.

■ **Note** This is the version of `AutoLotDAL.dll` that will carry forward for the rest of the book.

To get started, create a new Class Library project named AutoLotDAL. Delete the default class that was created and add two folders, named `EF` and `Models`. Add the Entity Framework to the project using NuGet. Right-click the project name and click Manage NuGet Packages. You didn't need to explicitly add EF to the previous example because the wizard took care of that for you. Once the NuGet Package Manager loads, enter **EntityFramework** in the search box, select the EntityFramework package, and click Install, as shown in Figure 22-7.