Entity Framework Core

Troelsen Kapitel 22 (och 23)

Grundläggande applikationsutveckling med C#

Obs! Kan tyvärr innehålla inaktuell information från tidigare versioner av Entity Framework som har ändrats för EF Core

ADO.NET – det gamla ramverket för databasaccess

- ADO.NET kan användas på 3 olika sätt: ***connected***, ***disconnected***, ***Entity Framework***
  - När man använder ***the connected layer*** (Troelsen kapitel 21), kopplar man explicit upp/ner sig till en DBMS, och använder ***connection***, ***command*** och ***data reader*** objekt för att kommunicera med den.
  - När man använder ***the disconnected layer*** (Troelsen kapitel 11) manipulerar man en mängd ***DataTable*** objekt (som finns i en ***DataSet***), vilka representerar en kopia av databasens tabeller. Man erhåller en ***DataSet*** från databasen, manipulerar dess ***DataTables***, och skickar sedan tillbaka dessa till DMBSen via ***DataAdapters*** (med några SQL satser) för att uppdatera databasen.
  - ***Entity Framework (EF) Core*** (Troelsen kapitel 22) är en ***ORM*** (***Object Relational Mapper***) som använder C#-objekt för att representera relationell data, där dataaccesskoden abstraheras bort från programmeraren. ***Entity Framework (EF) Core*** tillåter även programmeraren att använda ***LINQ queries*** som dynamiskt skapar underliggade SQL kommandon som skickas till databasen.

Entity Framework - historik

- ADO.NET har gjort det möjligt för .NET-programmerare att arbeta med relationsdata (på ett relativt enkelt sätt) sedan den första utgåvan av .NET-plattformen. Microsoft introducerade dock en ny komponent i api:et ADO.NET som kallas Entity Framework (eller helt enkelt EF) i .NET 3.5 Service Pack 1.
- Det övergripande målet med EF är att du ska kunna interagera med data från relationsdatabaser med hjälp av en objektmodell som mappar direkt till affärsobjekten (eller domänobjekten) i ditt program. I stället för att till exempel behandla en batch med data som en samling rader och kolumner, kan du använda en samling starkt skrivna objekt som kallas ***entiteter***. Dessa entiteter är också infödda LINQ-medvetna, och du kan fråga mot dem med LINQ-syntax. EF-körningsmotorn översätter dina LINQ-frågor automatiskt till rätt SQL-frågor för din räkning!
- Alla versioner av Entitetsramverket (upp till och med EF 6.x) stöder användning av en entitetsdesigner för att skapa en XML-fil (entity data model XML).
- Från och med version 4.1 lade EF till stöd för vanliga gamla CLR-objekt (POCO) med hjälp av en teknik som kallas ***Code First***.
- Entity Framework Core stöder bara Code First-paradigmet och släpper allt EDMX-stöd!

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.
- 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, but 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. Rather, **when you apply LINQ queries to your strongly typed classes, the EF runtime generates proper SQL statements on your behalf**.
- ***LINQ to Entities*** is the term that describes the act of **applying LINQ queries to ADO.NET EF entity objects**.
- ***LINQ to SQL*** is a database programming API introduced with .NET 3.5. This API is close in concept to EF, but is deprecated (i.e. don’t use LINQ to SQL).

The Role of Entities

- The strongly typed model classes 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.
- 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***.
- For example, take the simple **Inventory** table in the **AutoLot** database and the **Car** model class. 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.
- 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**.

The Role of Entities

- Consider the following **Program** class, which uses the **Car** model class and a related **context** class named **AutoLotEntities** to add a new row to the **Inventory** table of AutoLot.
- **EF** examines the configuration of your **models** and your **context** class to take the client-side representation of the **Inventory** table (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.

```
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();
      }
   }
}
```

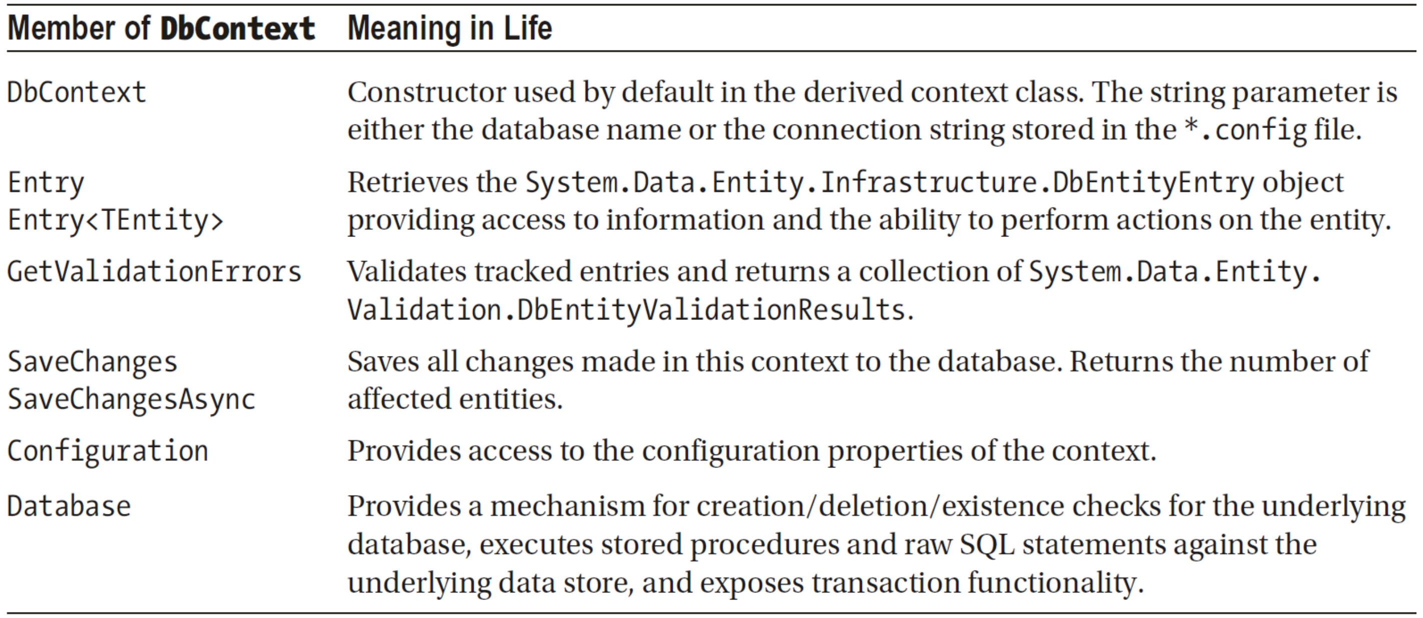
Under the covers, a connection to the database is made and opened, a proper SQL statement is generated and executed, and the connection is released and closed. These details are handled on your behalf by the framework. Thank you, Entity Framework!

The Building Blocks of the Entity Framework

- **EF** is built on top of **ADO.NET**. 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 that it supports a new set of services before it can interact with the **EF API**. 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.
- Many third-party databases (e.g., Oracle and MySQL) provide EF-aware data providers. https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/ado-net-overview contains 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

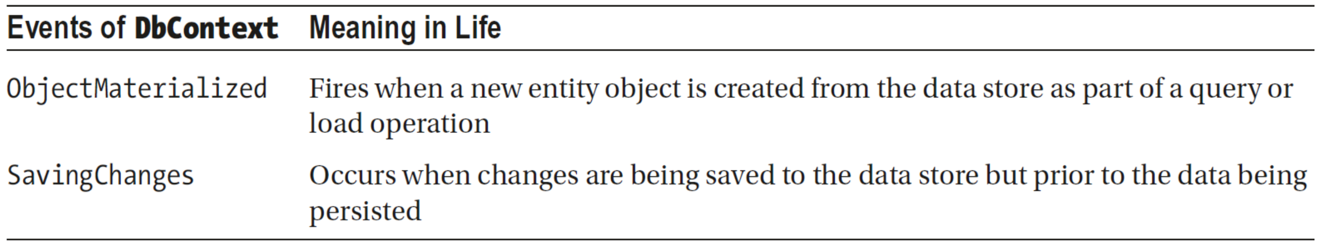
- The ***DbContext*** class represents a combination of the ***Unit of Work*** and ***Repository*** patterns that can be used to query from a database and group together changes that will be written back as a single unit of work.
- ***DbContext*** provides a number of core services to child classes, 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.
- ***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.



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.



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

- 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**:

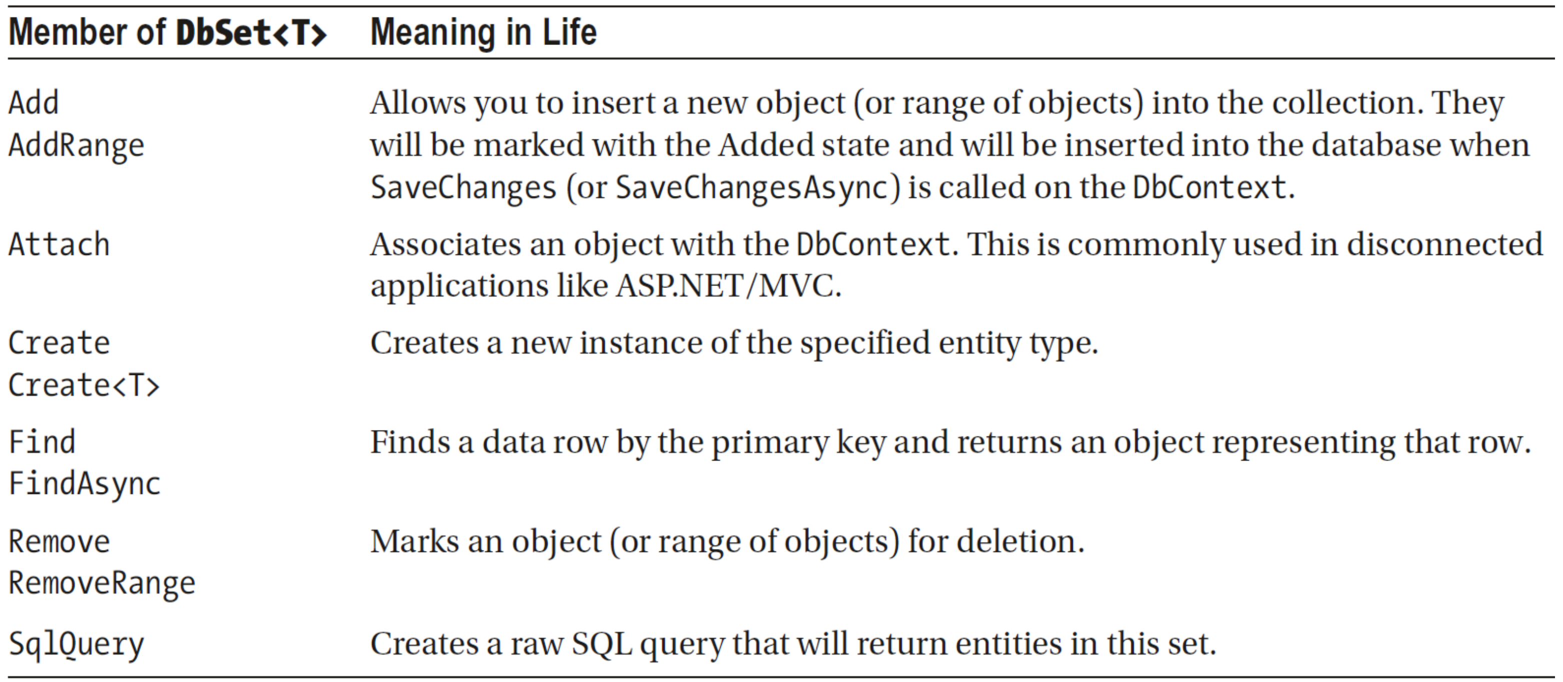
```
public class AutoLotEntities : DbContext
{
   public AutoLotEntities() : base("name=AutoLotConnection")
   {
   }

   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**:

```
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**.



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.

The Role of DbSet<T>

- Once you drill into the correct **property** of the object **context**, you can call any **member** of **DbSet<T>**.

```
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 LINQ extension methods, 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

- **Code First** doesn’t mean you can’t use **EF** with an existing database. It really just means **no EDMX model**.

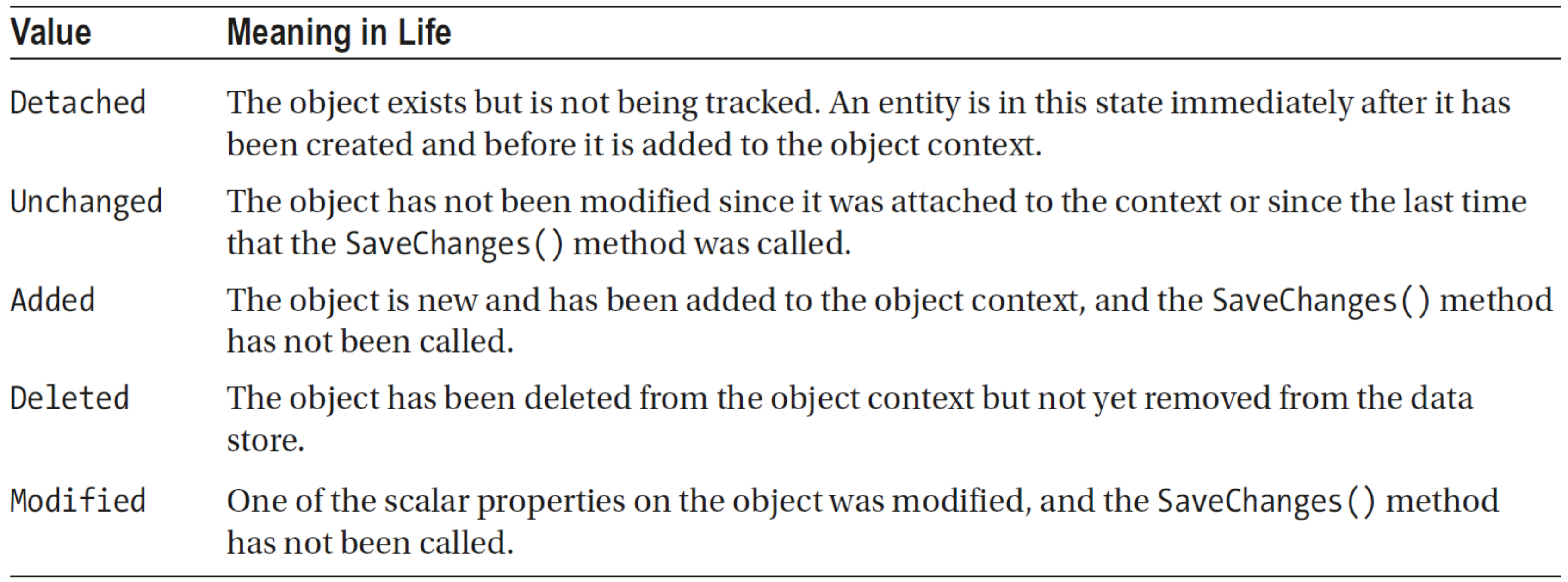
- You can use **Code First**:
  - From an **existing database**, or
  - Create a **new database from the entities** using **EF migrations**.

Transaction Support

- All versions of **EF** wrap each call to **SaveChanges** and **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: https://docs.microsoft.com/sv-se/ef/ef6/saving/transactions
- SQL statements executed using the **ExecuteSqlCommand()** from the **DbContext** database object are also wrapped in an implicit transaction (EF6).

Entity State and Change Tracking

- The **DbChangeTracker** automatically tracks the **state** for any object loaded into a **DbSet<T>** within a **DbContext**.
- While inside the **using** statement of the **DbContext**, any changes to the data will be tracked and saved when **SaveChanges** is called on the ***DbContext*** class (**AutoLotEntities** class).



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.

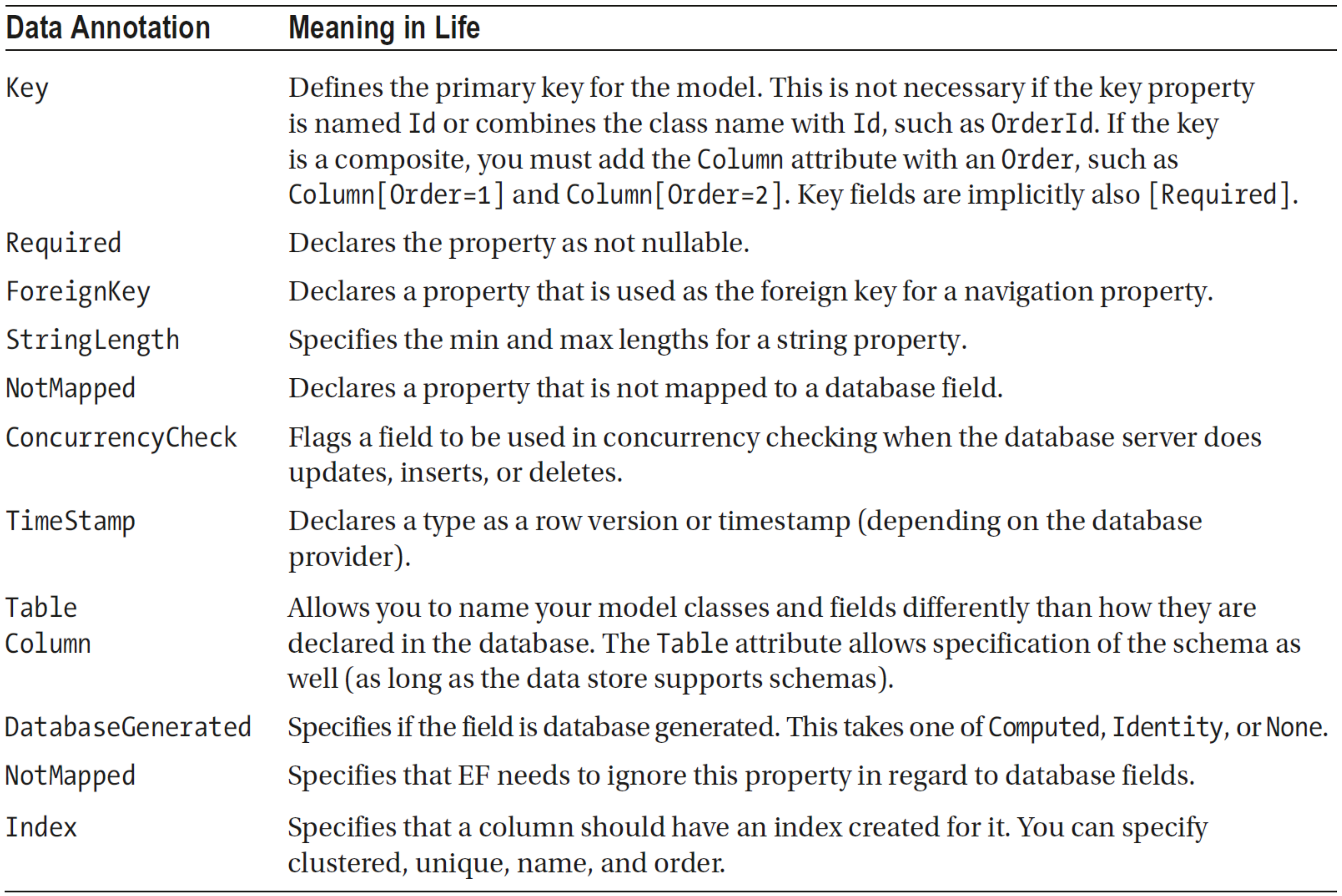
- If you need to check the state of an object, use the following code:

```
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.

Entity Framework Data Annotations

- **Data annotations** are **C# attributes** that are used to **shape entities** (e.g. for defining how entity classes and properties map to database tables and fields, etc.).



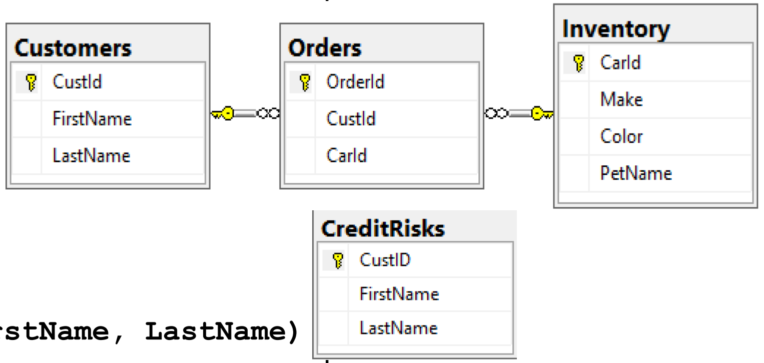
Data Annotation	Meaning in Life
Key	Defines the primary key for the model. This is not necessary if the key property is named Id or combines the class name with Id, such as OrderId. If the key is a composite, you must add the Column attribute with an Order, such as Column[Order=1] and Column[Order=2]. 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).
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.

Example: Code First from an existing database

- Let’s build a simple console app that uses **Code First** from an **existing database** to create the **model classes** representing the **AutoLot** database.

```
CREATE TABLE Inventory (
    CarId INT IDENTITY(1,1) NOT NULL,
    Make NVARCHAR(50) NULL,
    Color NVARCHAR(50) NULL,
    PetName NVARCHAR(50) NULL,
    PRIMARY KEY CLUSTERED (CarId ASC) )

CREATE TABLE Customers (
    CustId INT IDENTITY(1,1) NOT NULL,
    FirstName NVARCHAR(50) NULL,
    LastName NVARCHAR(50) NULL,
    PRIMARY KEY CLUSTERED (CustID ASC) )

CREATE TABLE Orders (
    OrderId INT IDENTITY(1,1) NOT NULL,
    CustId INT NOT NULL,
    CarId INT NOT NULL,
    PRIMARY KEY CLUSTERED (OrderId ASC),
    FOREIGN KEY (CustId) REFERENCES Customers(CustId),
    FOREIGN KEY (CarId) REFERENCES Inventory(CarId) )

CREATE TABLE CreditRisks (
    CustId INT IDENTITY(1,1) NOT NULL,
    FirstName NVARCHAR(50) NULL,
    LastName NVARCHAR(50) NULL,
    PRIMARY KEY CLUSTERED (CustID ASC) )
```

```
INSERT INTO Inventory (CarId, Make, Color, PetName)
VALUES
(1, 'VW', 'Black', 'Zippy'),
(2, 'Ford', 'Rust', 'Rusty'),
(3, 'Saab', 'Black', 'Mel'),
(4, 'Yugo', 'Yellow', 'Clunker'),
(5, 'BMW', 'Black', 'Bimmer'),
(6, 'BMW', 'Green', 'Hank'),
(7, 'BMW', 'Pink', 'Pinky')

INSERT INTO Customers (CustId, FirstName, LastName)
VALUES
(1, 'Dave', 'Brenner'),
(2, 'Matt', 'Walton'),
(3, 'Steve', 'Hagen'),
(4, 'Pat', 'Walton')

INSERT INTO Orders (OrderId, CustId, CarId)
VALUES
(1, 1, 5),
(2, 2, 1),
(3, 3, 4),
(4, 4, 7)
```

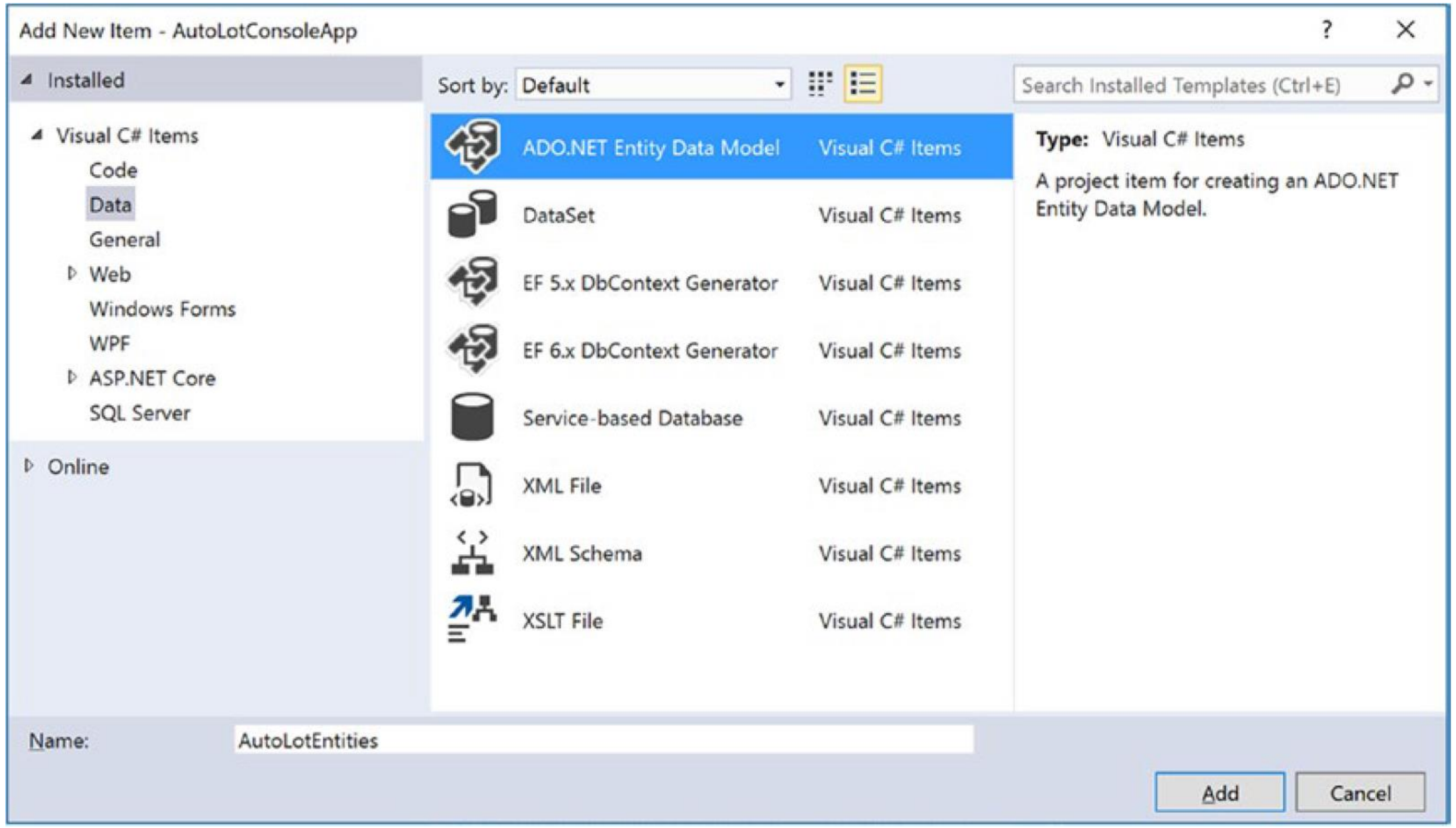


```
Innan varje INSERT INTO använder vi ...
SET IDENTITY_INSERT <Tabellnamn> ON
... och efter varje INSERT INTO ...
SET IDENTITY_INSERT <Tabellnamn> OFF
```

```
CREATE PROCEDURE GetPetName
    @carID INT,
    @petName NVARCHAR(50) OUTPUT
AS
BEGIN
    SELECT @petName = PetName
    FROM Inventory
    WHERE CarId = @carID
END
```

Generating the Model

- Create 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**.


Add New Item - AutoLotConsoleApp

Installed
- Visual C# Items
  - Code
  - Data
  - General
  - Web
  - Windows Forms
  - WPF
  - ASP.NET Core
  - SQL Server
- Online

Sort by: Default

Search Installed Templates (Ctrl+E)

ADO.NET Entity Data Model	Visual C# Items
DataSet	Visual C# Items
EF 5.x DbContext Generator	Visual C# Items
EF 6.x DbContext Generator	Visual C# Items
Service-based Database	Visual C# Items
XML File	Visual C# Items
XML Schema	Visual C# Items
XSLT File	Visual C# Items

Type: Visual C# Items

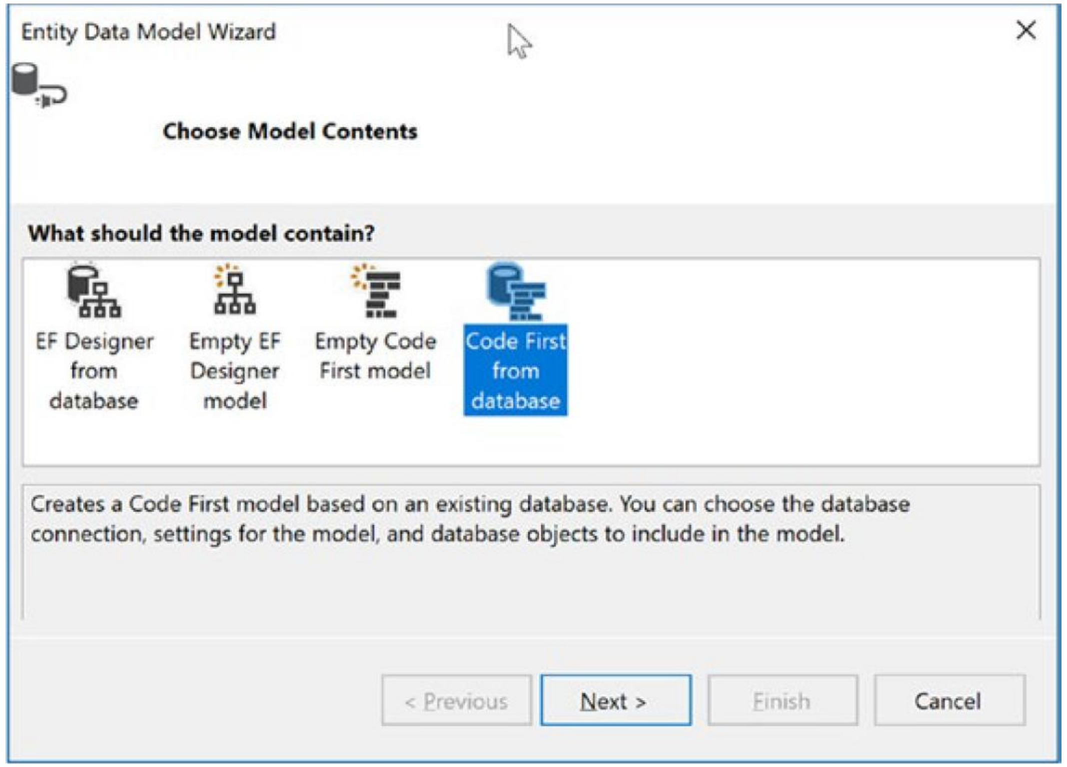
A project item for creating an ADO.NET Entity Data Model.

Name: AutoLotEntities

Add Cancel

Generating the Model

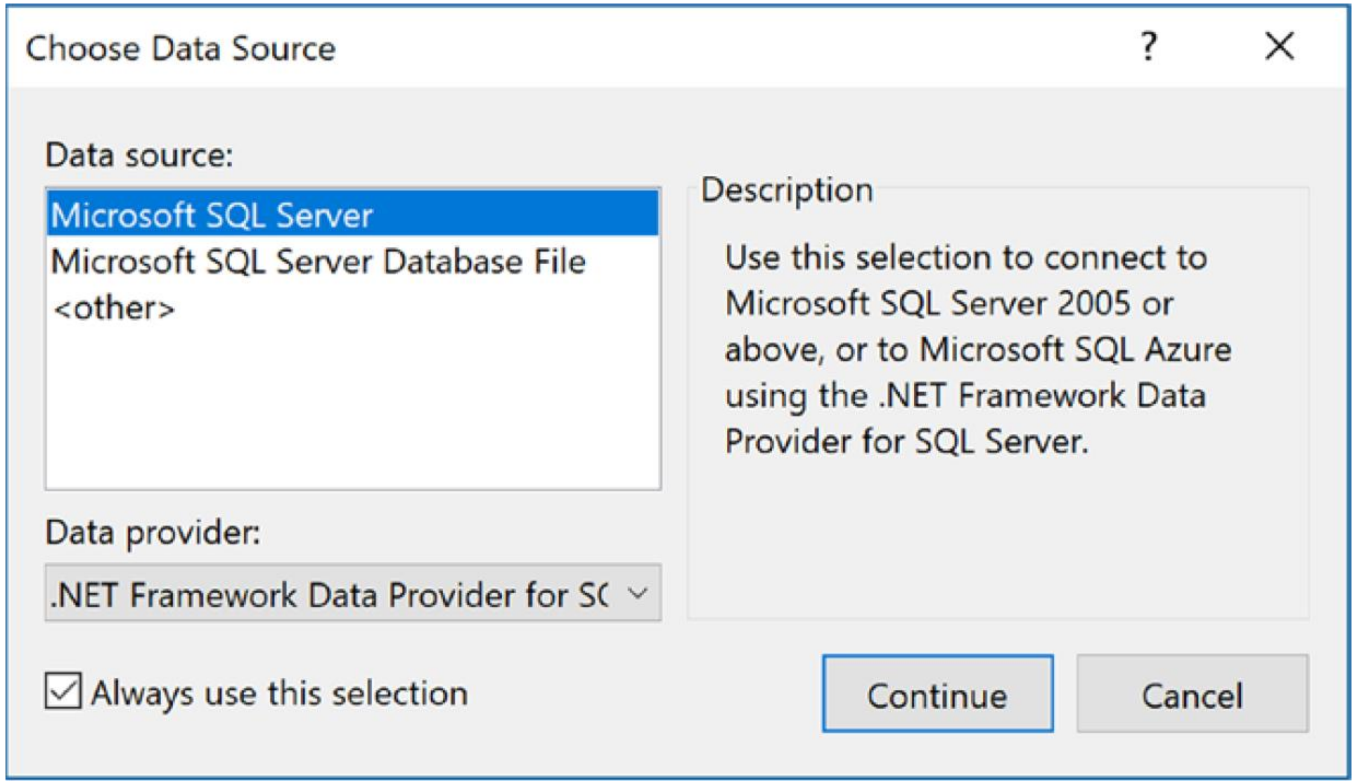
- Clicking the **Add** button launches the **Entity Model Data Wizard**.
- The wizard's first step allows you to select the option to generate an **EDM** using the **Entity Framework Designer** (from an **existing database** or by creating an **empty designer**) or using **Code First** (from an **existing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



Creates a Code First model based on an existing database. You can choose the database connection, settings for the model, and database objects to include in the model.

Generating the Model

- 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. If this is not the case, click the **New Connection** button. This loads the **Choose Data Source** screen.
- Select **Microsoft SQL Server** (as the **Data Source**) and click **Continue**.


Choose Data Source

Data source:

Microsoft SQL Server
Microsoft SQL Server Database File
<other>

Description

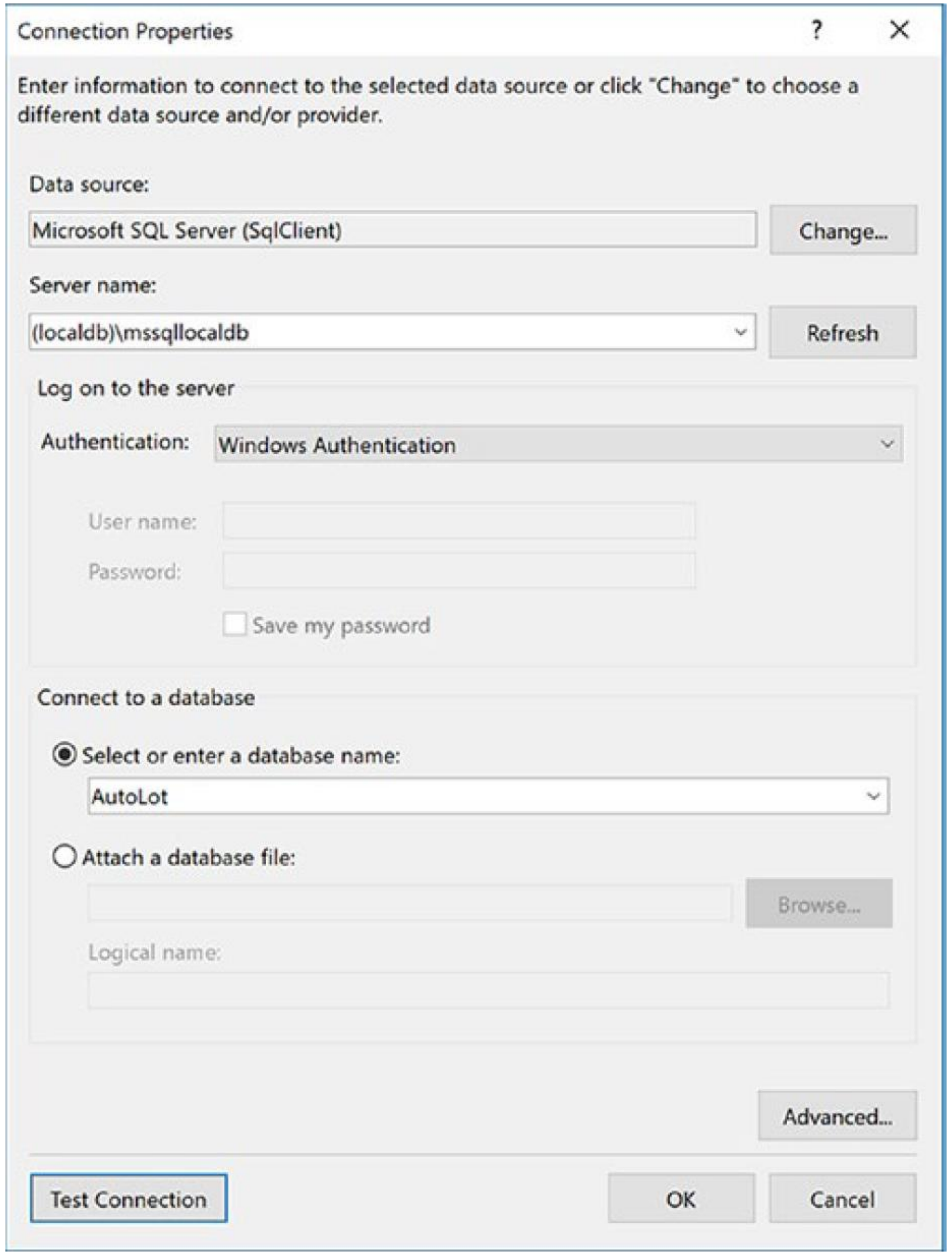
Use this selection to connect to Microsoft SQL Server 2005 or above, or to Microsoft SQL Azure using the .NET Framework Data Provider for SQL Server.

Data provider: .NET Framework Data Provider for SC

[✓] Always use this selection

Generating the Model

- On the next screen, select **(localdb)\mssqllocaldb** for the server and then select **AutoLot** for the database.



Connection Properties

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

Data source: Microsoft SQL Server (SqlClient)

Server name: (localdb)\mssqllocaldb

Log on to the server

Authentication: Windows Authentication

User name:

Password:

[ ] Save my password

Connect to a database

(●) Select or enter a database name: AutoLot

( ) Attach a database file:

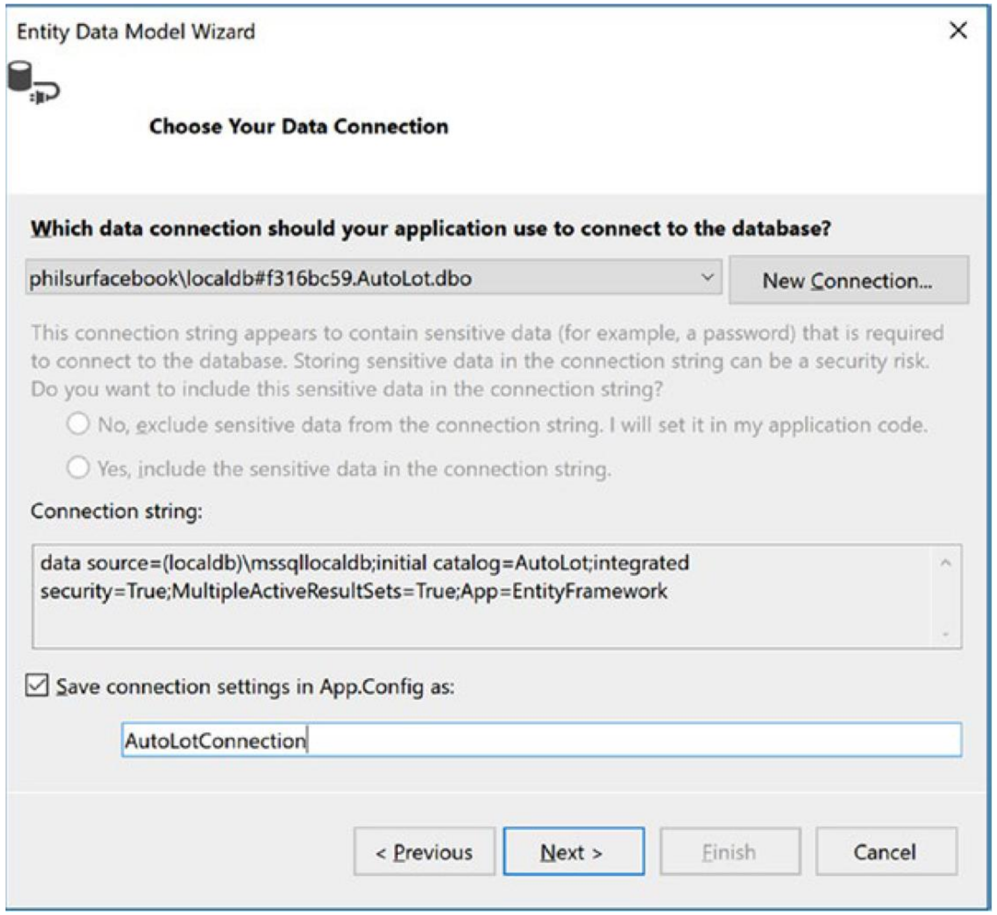
Logical name:

Generating the Model

- 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**.
- 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
```

- Notice the **App=EntityFramework** namevalue pair. **App** is short for application name, which can be used when troubleshooting SQL Server issues.



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

philsurfacebook\localdb#f316bc59.AutoLot.dbo

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.

Connection string:

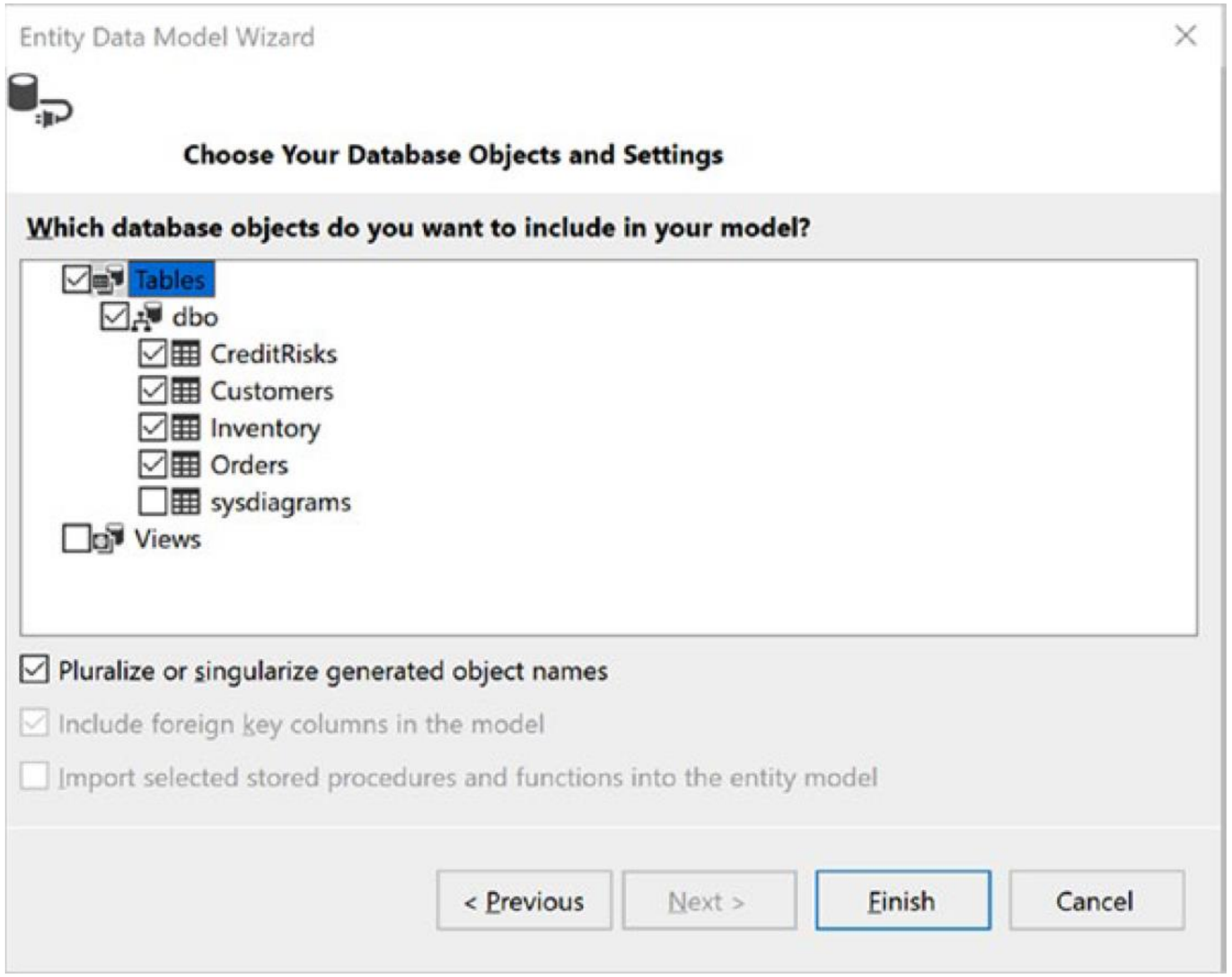
data source=(localdb)\mssqllocaldb;initial catalog=AutoLot;integrated security=True;MultipleActiveResultSets=True;App=EntityFramework

☑ Save connection settings in App.Config as:

AutoLotConnection

Generating the Model

- 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**.
- Click the **Finish** button to **generate** your **models** and the derived **DbContext**.



Entity Data Model Wizard

Choose Your Database Objects and Settings

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

- [✓] Tables
  - [✓] dbo
    - [✓] CreditRisks
    - [✓] Customers
    - [✓] Inventory
    - [✓] Orders
    - [ ] sysdiagrams
- [ ] Views

[✓] Pluralize or singularize generated object names

[✓] Include foreign key columns in the model

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

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 **data annotations**, you can change the **entity names**, as well as **property names**, to something else if needed.
- 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**.

What Did That Do?

- 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**.
  - There are also two **SuppressMessage** attributes.
- You can also see that:
  - The Inventory class has a collection of Order objects
  - The Order class contains a Inventory property.
  - These are **navigation properties**.

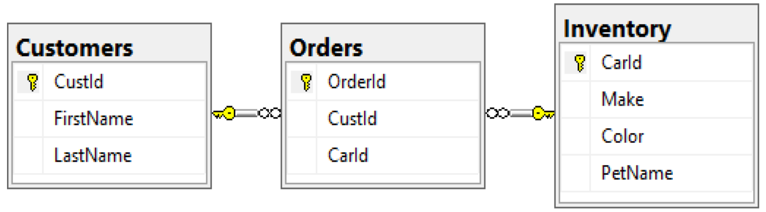
```
[Table("Inventory")]
public partial class Inventory
{
   public Inventory()
   {
      Orders = new HashSet<Order>();
   }

   [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; }
}
```

```
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; }
}
```

What Did That Do?

- 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);
   }
}
```



What Did That Do?

- Open the **App.config** file. You will see a new **configSection** (named **entityFramework**), as well as the **connection string** (**AutoLotConnection**) generated by the wizard.

```
<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, PublicKeyToken=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.
  - Change the **property name** to **CarNickName**.
- Note that you will also have to **change the type of the Inventory property to Car** in the **Order** class.

Car.cs

```
[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
}
```

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

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

Changing the Default Mappings

- 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.

```
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**:

CarPartial.cs

```
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.
- Add 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** class named **AddNewRecord()**:

```
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;
        }
    }
}
```

Inserting a Record

- 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)
{
   int carId = AddNewRecord();
   WriteLine(carId);
   ReadLine();
}
```

- The output to the console is 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.

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):

```
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 using the **DbSet<T> SqlQuery()** method, update the **PrintInventory()** method to the following:

```
private static void PrintAllInventory()
{
   using (var context = new AutoLotEntities())
   {
      string sql = "Select CarId,Make,Color,PetName as CarNickName from Inventory where Make=@p0";
      foreach (Car c in context.Cars.SqlQuery(sql, "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 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 more powerful when using **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';
```

Querying with LINQ

- A couple more **LINQ query** examples …

```
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);
      }
   }
}
```

Querying with LINQ – 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:

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

The Timing of EF Query Execution

- It's important to understand ***when* the queries execute**.
  - **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**.
- In the following code, the database call is made when the collection is iterated over with **foreach**.

```
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

- ***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** 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, the **Order** model has a **one-to-one relationship with the Inventory (Car)** record.

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

  - The **Order** model navigates back to the **Inventory** model through another **virtual** property of type **Inventory (Car)**.

- 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; }
```

- If the **CarId foreign key were a nullable int** (i.e. **int?**), then it would have a **zero-to-one relationship**.

The Role of Navigation Properties

- 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**,
  - **Explicit loading** 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**.
  - 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** to **false**:

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

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

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 use the **Include()** method as follows:

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

Eager Loading

- 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
```

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 methods below of **context** to choose what to load and then call **Load()**:
  - **Collection()** for collections.
  - **Reference()** for single objects.

```
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**.

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**.

Deleting a Single Record

- 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, 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()**:

```
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();
      }
   }
}
```

- 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**.

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:

```
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 Entity State

- A record can be deleted 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:

```
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);
      }
   }
}
```

Deleting a Record Using Entity State

- 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.
- 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:

```
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.

Creating the AutoLot Data Access Layer

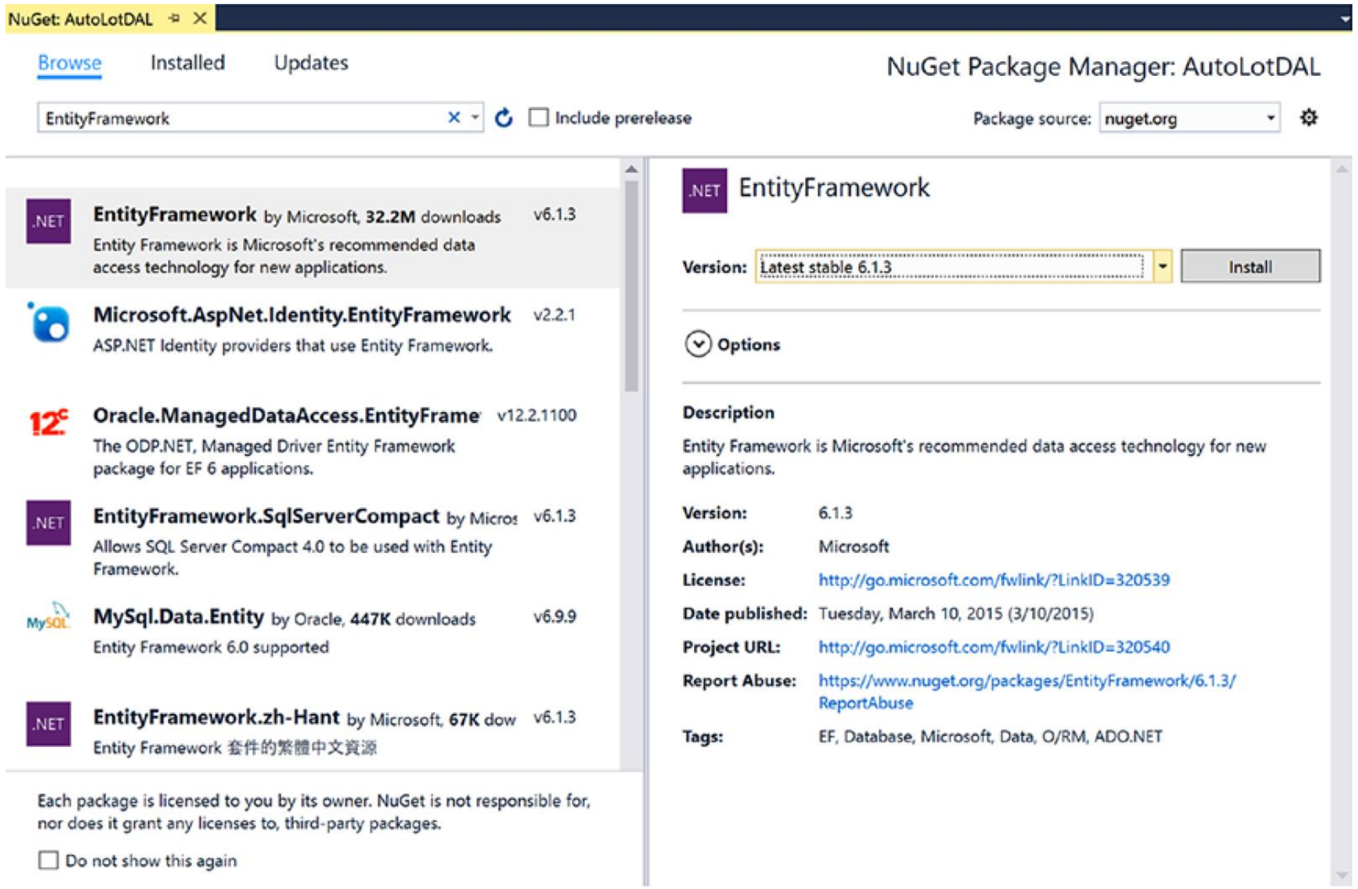
- In the previous section, you used a wizard to create the entities and context from an **existing database**.
- EF can also **create your database for you based on your model classes and derived DbContext class**.
- In addition to creating the initial database, EF enables you to use **migrations** to **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**.

Creating the AutoLot Data Access Layer

- Create a new **Class Library** project named **AutoLotDAL**.
- **Delete the default class** (Class1.cs) that was created.
- **Add two folders** named **EF** and **Models**.

Creating the AutoLot Data Access Layer

- Add the **Entity Framework** to the project using **NuGet** (you didn't need to explicitly add EF to the previous example because the wizard took care of that for you):
  - Right-click the project name and click **Manage NuGet Packages**.
  - Once the **NuGet Package Manager** loads, enter **EntityFramework** in the **Browse** search box, select the **EntityFramework package**, and click **Install**.
  - Accept the changes and the license agreement, and the **Entity Framework** will be installed into your project.



EntityFramework by Microsoft, 32.2M downloads v6.1.3
Entity Framework is Microsoft's recommended data access technology for new applications.

Microsoft.AspNet.Identity.EntityFramework v2.2.1
ASP.NET Identity providers that use Entity Framework.

Oracle.ManagedDataAccess.EntityFrame v12.2.1100
The ODP.NET, Managed Driver Entity Framework package for EF 6 applications.

EntityFramework.SqlServerCompact by Micros v6.1.3
Allows SQL Server Compact 4.0 to be used with Entity Framework.

MySql.Data.Entity by Oracle, 447K downloads v6.9.9
Entity Framework 6.0 supported

EntityFramework.zh-Hant by Microsoft, 67K dow v6.1.3
Entity Framework 套件的繁體中文資源

Each package is licensed to you by its owner. NuGet is not responsible for, nor does it grant any licenses to, third-party packages.

EntityFramework

Version: Latest stable 6.1.3

Description
Entity Framework is Microsoft's recommended data access technology for new applications.

Version:	6.1.3
Author(s):	Microsoft
License:	http://go.microsoft.com/fwlink/?LinkID=320539
Date published:	Tuesday, March 10, 2015 (3/10/2015)
Project URL:	http://go.microsoft.com/fwlink/?LinkID=320540
Report Abuse:	https://www.nuget.org/packages/EntityFramework/6.1.3/ReportAbuse
Tags:	EF, Database, Microsoft, Data, O/RM, ADO.NET

Adding the Model Classes

- Copy the models (**Car.cs**, **CarPartial.cs**, **CreditRisk.cs**, **Customer.cs** and **Order.cs**) from the **AutoLotConsoleApp** project into the **Models** folder of the project.
- Correct all of the namespaces (by changing them to **AutoLotDAL.Models**).
- Revert the changes from the last section:
  - Change the **Car class**, **constructor** and **file names** back to **Inventory**.
  - Change the **CarPartial** file name to **InventoryPartial**, and name the **class name Inventory**.
  - Change the **Car type** to **Inventory** in the **Order class**.
  - Return the **Inventory.CarNickName** property name to **PetName**.
  - Update the **ToString()** method in the **Inventory partial class** to use **PetName**.
  - Change **Foo** back to **CarId** in the **Order class**.

Update the Inventory Model Class

- Open **Inventory.cs**, and move the initialization of the **HashSet<Order>** to the **Orders property**:

```
public virtual ICollection<Order> Orders { get; set; } = new HashSet<Order>();
```

- This doesn’t change anything in how the code works; it just takes advantage of new C# features to clean up the code.

Update the Inventory Partial Class

- Import the following namespace into the **InventoryPartial.cs** file:

```
using System.ComponentModel.DataAnnotations.Schema;
```

- Add a **calculated field** that combines the **Make** and **Color** of the **Car**.
  - This is a field that is not to be stored in the database or populated when an object is materialized from the data reader.
  - The **[NotMapped]** attribute informs EF that this is a .NET-only property.

```
[NotMapped]
public string MakeColor => $"{Make} + ({Color})";
```

Update the Customer Model Class

- Open the **Customer.cs** class and move the creation of the new **HashSet<Order>** to the property like this:

```
public virtual ICollection<Order> Orders { get; set; } = new HashSet<Order>();
```

- Next, add a .NET-only field to return the **FullName** of the **Customer**:

```
[NotMapped]
public string FullName => FirstName + " " + LastName;
```

Update the DbContext

- Copy the **AutoLotEntities** class from the previous project into the **EF** folder of the current project.
- Update the namespace to **AutoLotDAL.EF**
- Import the **AutoLotDAL.Models** namespace.
- Change **DbSet<Car>** to **DbSet<Inventory>**.
- Change **Car** to **Inventory** in the **OnModelCreating()** method.

Update the App.config File

- Open the **App.config** file and look at the changes that were made by **NuGet** when you installed the **EntityFramework package**.
- Most of them should look familiar to you. What's missing is the **connection string**, so add that into the **App.Config** file like this:

```
<connectionStrings>
   <add name="AutoLotConnection" connectionString="data source=(LocalDb)\MSSQLLocalDB;
        initial catalog=AutoLot;integrated security=True;MultipleActiveResultSets=True;
        App=EntityFramework" providerName="System.Data.SqlClient" />
</connectionStrings>
```

- The situation you have just replicated is one where the **database does not yet exist**, so **everything is created in C# code first and then** (as you will soon see) **migrated to the database**.

Initializing the Database

- A powerful EF feature is the ability to make sure the **database matches the model** as well as **initializes the database with data**. This is especially handy during development and testing since the process can restore the database to a known state before each run of your code. There are two classes that you can derive from to turn this feature on: **DropCreateDatabaseIfModelChanges<TContext>** and **DropCreateDatabaseAlways<TContext>**
- Create a new class in the **EF** directory named **MyDataInitializer**. Make the class **public** and inherit **DropCreateDatabaseAlways<AutoLotEntities>**. Add the following **using** statement:

```
using AutoLotDAL.Models;
```

- Next, add an override for the **Seed()** method:

```
public class MyDataInitializer : DropCreateDatabaseAlways<AutoLotEntities>
{
   protected override void Seed(AutoLotEntities context)
   {
      base.Seed(context);
   }
}
```

- The **DropCreateDatabaseAlways** class is a generic class that gets typed for a **DbContext** class, in this case, the **AutoLotEntities** class. As the name implies, it will **drop and re-create the database every time the initializer is executed**. There is also a **DropCreateDatabaseIfModelChanges<TContext>** class that drops and re-creates the database when there are changes in the model.

Executing an Upsert

- ***Upsert*** is a term that is a combination of ***update*** and ***insert*** and is supported by EF with the **AddOrUpdate()** method on the **DbSet<T>**.
- The method takes a **lambda expression** to specify what defines the **uniqueness of each record** and then the **list of objects to *upsert***.
  - If the record exists (based on the uniqueness key) in the database, it will be **updated**.
  - If it doesn't exist, it will be **inserted**.
- Here is an example of an ***upsert*** for the **Inventory** class that checks for the **Make** and **Color** of each car before inserting the records:

```
context.Cars.AddOrUpdate(x => new {x.Make, x.Color}, car);
```

Seeding the Database

- The **Seed()** method in both of the initialization classes is used to populate the database.
- By using the **AddOrUpdate()** method, you can make sure that your database is restored to the same state without duplicating data.

Seeding the Database

- Here is an example of how to seed the database (continued on next slide).

```
protected override void Seed(AutoLotEntities context)
{
   var customers = new List<Customer>
   {
      new Customer {FirstName = "Dave", LastName = "Brenner"},
      new Customer {FirstName = "Matt", LastName = "Walton"},
      new Customer {FirstName = "Steve", LastName = "Hagen"},
      new Customer {FirstName = "Pat", LastName = "Walton"},
      new Customer {FirstName = "Bad", LastName = "Customer"},
   };
   customers.ForEach(x => context.Customers.AddOrUpdate(c=>new {c.FirstName, c.LastName},x));

   var cars = new List<Inventory>
   {
      new Inventory {Make = "VW", Color = "Black", PetName = "Zippy"},
      new Inventory {Make = "Ford", Color = "Rust", PetName = "Rusty"},
      new Inventory {Make = "Saab", Color = "Black", PetName = "Mel"},
      new Inventory {Make = "Yugo", Color = "Yellow", PetName = "Clunker"},
      new Inventory {Make = "BMW", Color = "Black", PetName = "Bimmer"},
      new Inventory {Make = "BMW", Color = "Green", PetName = "Hank"},
      new Inventory {Make = "BMW", Color = "Pink", PetName = "Pinky"},
      new Inventory {Make = "Pinto", Color = "Black", PetName = "Pete"},
      new Inventory {Make = "Yugo", Color = "Brown", PetName = "Brownie"},
   };
   context.Inventory.AddOrUpdate(x=>new {x.Make,x.Color},cars.ToArray());
```

continued on next slide

Seeding the Database

- Here is an example of how to seed the database (continued from previous slide).

continued from
previous slide

```
   var orders = new List<Order>
   {
      new Order {Inventory= cars[0], Customer = customers[0]},
      new Order {Inventory= cars[1], Customer = customers[1]},
      new Order {Inventory= cars[2], Customer = customers[2]},
      new Order {Inventory= cars[3], Customer = customers[3]},
   };
   orders.ForEach(x => context.Orders.AddOrUpdate(c=>new{c.CarId,c.CustId},x));

   context.CreditRisks.AddOrUpdate(x=>new {x.FirstName,x.LastName},
                                   new CreditRisk
                                   {
                                      CustID = customers[4].CustId,
                                      FirstName = customers[4].FirstName,
                                       LastName = customers[4].LastName
                                   });
}
```

Seeding the Database

- The last step is to **set the initializer**, with the following code (which you will add in the next section):

```
Database.SetInitializer(new MyDataInitializer());
```

Test-Driving AutoLotDAL

- To test the data initialization and seed code, add a new Console Application project named **AutoLotTestDrive** to the solution and set this project as the startup project.
- Add **EF** to the project through **NuGet** and update **connectionStrings** in **App.config**.

```
<connectionStrings>
   <add name="AutoLotConnection" connectionString="data source=(localdb)\mssqllocadb;
        initial catalog=AutoLot;integrated security=True;MultipleActiveResultSets=True;
        App=EntityFramework" providerName="System.Data.SqlClient" />
</connectionStrings>
```

- Add a reference to the **AutoLotDAL** project.
- Open **Program.cs** and add **using** statements.

```
using AutoLotDAL.EF;
using AutoLotDAL.Models;
```

- Initialize the database in the **Main()** method.
  - The **SetInitializer()** method drops and re-creates the database and then runs the **Seed()** method to populate the database.

```
static void Main(string[] args)
{
   Database.SetInitializer(new MyDataInitializer());
   using (var context = new AutoLotEntities())
   {
      foreach (Inventory c in context.Inventory)
      {
         Console.WriteLine(c);
      }
   }
   Console.ReadLine();
}
```

Entity Framework Migrations

- Once you deploy your app to production, you can’t keep dropping the database every time your users run the app. If your model changes, you need to keep your database in sync. This is where **EF migrations** come into play. Before creating your first migration, you are going to make some changes to illustrate the problem.
- Open **Program.cs** and comment out the following line:

```
Database.SetInitializer(new MyDataInitializer());
```

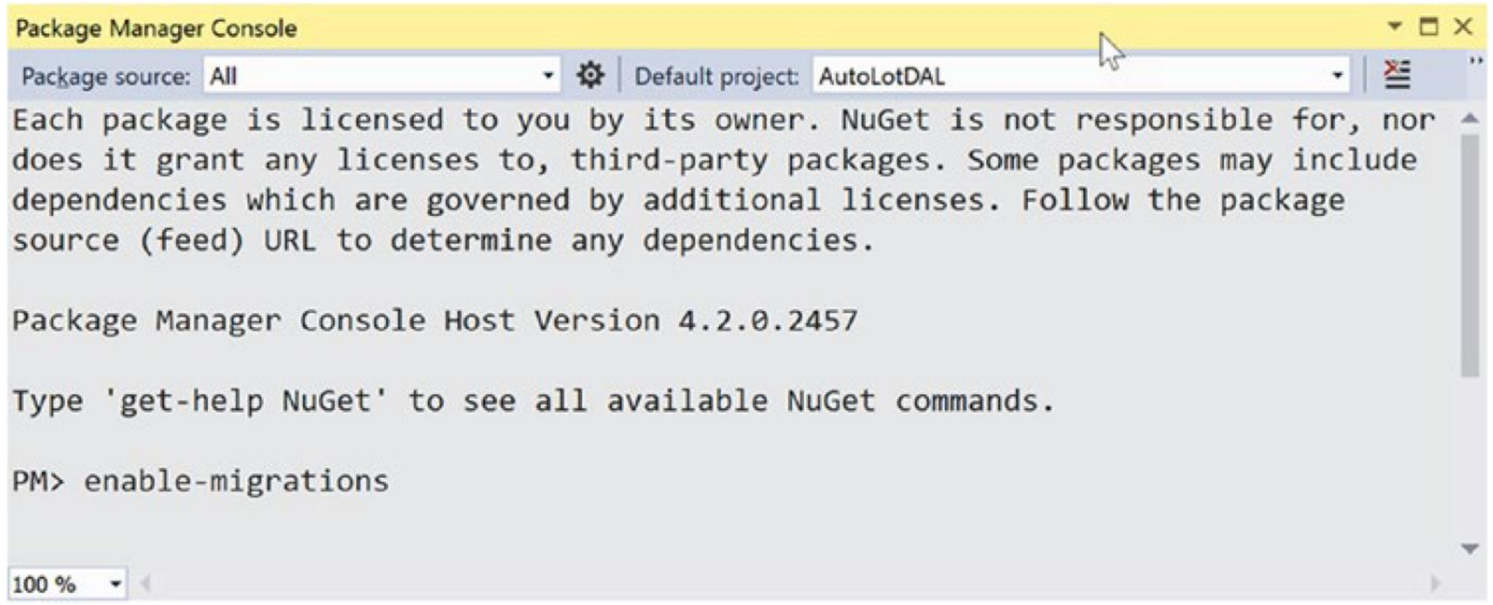
- As discussed earlier, the **data initializer** drops and re-creates the database, either each time the app runs or when the model changes.

Create the Initial Migration

- Each time a **context** is created and used to do database operations, it checks for the existence of the **__MigrationHistory** table. If the table exists, the context checks for the most recent record in the table and compares a hash of the current EF model to the most recent hash stored in the table. If the values are different, then EF will throw an exception.
- When you create your model from an existing database, the **__MigrationHistory** table does ***not*** get created. Why does it matter? When your **DbContext** class is instantiated and before the first call to the database from your custom code, EF checks the migration history. Since this table doesn't exist, there are a series of exceptions generated and swallowed by the framework. As you well know, exceptions can be expensive operations, and this can potentially cause a performance issue. Even if you don't ever plan on using migrations, you should enable migrations as covered in the next section.

Enable Migrations

- Delete the **AutoLot** database by using the **Object Explorer** in **SSMS**.
- **Enable migrations** for your project.
  - Open the **Package Manager Console** (the command-line tool for managing **NuGet** packages) by selecting ***View →Other Windows →Package Manager Console***.
  - Make sure **Default Project** is set to **AutoLotDAL**.
  - Enter **enable-migrations**.



```
Each package is licensed to you by its owner. NuGet is not responsible for, nor
does it grant any licenses to, third-party packages. Some packages may include
dependencies which are governed by additional licenses. Follow the package
source (feed) URL to determine any dependencies.

Package Manager Console Host Version 4.2.0.2457

Type 'get-help NuGet' to see all available NuGet commands.

PM> enable-migrations
```

Enable Migrations

- You will see a message that states **“Checking if the context targets an existing database..”**
- You will see that a **Migrations** folder is created with one class named **Configuration.cs**.
- Examine the **Configuration.cs** class.
  - The code in the constructor instructs **EF** to **disable automatic migrations** (which is the setting you will use most of the time since you want to have control over how migrations work).
  - The **Seed()** method enables you to add data to the database just like the **Seed()** method in the **database initializer**.

```
internal sealed class Configuration : DbMigrationsConfiguration<AutoLotDAL.EF.AutoLotEntities>
{
   public Configuration()
   {
      AutomaticMigrationsEnabled = false;
   }

   protected override void Seed(AutoLotDAL.EF.AutoLotEntities context)
   {
   }
}
```

Create the Initial Migration

Use this instead if you get a FileNotFound exception

- In the **Package Manager Console**, enter the following:

```
Add-migration Initial
```

```
EntityFramework6\Add-Migration Initial
```

- This creates an additional file in the **Migrations** folder named similar to **201707262033409_Initial.cs**. The name of the file starts with the date and CPU time, which is followed by the name of the migration. This naming format enables EF to run migrations in the correct chronological order (if more than one exists).
- Open the **201707262033409_Initial.cs** class. Take note of the two methods, one named **Up()** and the other **Down()**. The **Up()** method is for applying the changes to the database, and the **Down()** method is used to roll back the changes. When you apply a migration, any earlier migrations that have not been applied get applied by running the **Up()** methods, and any later migrations are rolled back automatically using the **Down()** methods.

Update the Database

Set the AutoLotDAL as the startup project if localdb instance not found

Use this instead if you get a FileNotFound exception

- To update the database, enter the following in the **Package Manager Console**:

```
update-database
```

```
EntityFramework6\update-database
```

- This creates the database and the tables, as well as the **__MIgrationHistory table**. It adds a row for the migration just applied.

```
MigrationId            ContextKey                           Model               ProductVersion
201707262033409_Initial AutoLotDAL.Migrations.Configuration 0x1F8B08000000... 6.1.3-40302
```

- Now that the database and the model are in sync, you can safely start updating the model.

Update the Model

- There are several changes that you want to make to the model. Once those are complete, you will create another migration to sync up the database.

Adding EntityBase

- Your models are not consistent with the names of their **primary keys**, and that is something you want to change. You are going to change all of the **primary key** fields to the name of **Id** and push the **Id** fields into a **base class**.
  - Create a new folder named **Base** in the **Models** folder.
  - Add a new class named **EntityBase**. Update the code to the following (note the **using** statements):

```
using System.ComponentModel.DataAnnotations;

namespace AutoLotDAL.Models.Base
{
   public class EntityBase
   {
      [Key]
      public int Id { get; set; }
   }
}
```

  - Update all the **model classes** to inherit from **EntityBase**, remove the **primary key** fields and update the **foreign key** names in the **Order** class. Also, update the navigation properties on the Order class using the **ForeignKey** data annotation:

```
public partial class Order : EntityBase
{
   public int CustomerId { get; set; }

   public int CarId { get; set; }

   [ForeignKey(nameof(CustomerId))]
   public virtual Customer Customer { get; set; }

   [ForeignKey(nameof(CarId))]
   public virtual Inventory Car { get; set; }
}
```

  - You will also need to fix the build errors in **MyDataInitializer**.

Adding a TimeStamp Property

- Another change to make is to add **concurrency checking** to the database. To do this, you will add a **Timestamp** property to all your **tables** using the **Timestamp** data annotation.
- Update the **EntityBase** class to the following:

```
public class EntityBase
{
    [Key]
    public int Id { get; set; }

    [Timestamp]
    public byte[] Timestamp { get; set; }
}
```

- The **Timestamp** data annotation maps the field to the SQL Server **RowVersion** data type, which in C# is represented as a **byte[]**.
- This field will now participate in **concurrency checking**.

Update the Credit Risk Class

- Finally, you are going to create a **unique index** on the **FirstName** and **LastName** properties using **data annotations**.
- Since this is a **complex key**, you also need to specify a **name for the index** and the **order for each column in the index**.
- In this example, the index name is **IDX_CreditRisk_Name**, and the column order for the index is **LastName and then FirstName**, and the index is created as **unique**.

- Add the following using statement:

```
using System.ComponentModel.DataAnnotations.Schema;
```

- Finally, update the class to the following:

```
public partial class CreditRisk : EntityBase
{
    [StringLength(50)]
    [Index("IDX_CreditRisk_Name",IsUnique = true,Order=2)]
   public string FirstName { get; set; }

    [StringLength(50)]
    [Index("IDX_CreditRisk_Name", IsUnique = true, Order = 1)]
   public string LastName { get; set; }
}
```

Create the Final Migration

- Now that the model is updated, it's time to **create the final migration** for the example.
- In the **Package Manager Console**, enter the following:

```
Add-migration Final
```

- This creates an additional file in the **Migrations** folder named **201707262025040_Final.cs**.
- If you run the migration now, you will encounter some errors. After all, the process is not perfect! You need to make a slight modification to the **Up()** and **Down()** methods (see next two slides).

Create the Final Migration

```
public override void Up()
{
   DropForeignKey("dbo.Orders", "CarId", "dbo.Inventory");
   DropForeignKey("dbo.Orders", "CustId", "dbo.Customers");
   RenameColumn(table: "dbo.Orders", name: "CustId", newName: "CustomerId");
   RenameIndex(table: "dbo.Orders", name: "IX_CustId", newName: "IX_CustomerId");
   DropPrimaryKey("dbo.Inventory");
   DropPrimaryKey("dbo.Orders");
   DropPrimaryKey("dbo.Customers");
   DropPrimaryKey("dbo.CreditRisks");
   DropColumn("dbo.Inventory", "CarId");
   DropColumn("dbo.Orders", "OrderId");
   DropColumn("dbo.Customers", "CustID");
   DropColumn("dbo.CreditRisks", "CustID");
   AddColumn("dbo.Inventory", "Id", c => c.Int(nullable: false, identity: true));
   AddColumn("dbo.Orders", "Id", c => c.Int(nullable: false, identity: true));
   AddColumn("dbo.Customers", "Id", c => c.Int(nullable: false, identity: true));
   AddColumn("dbo.CreditRisks", "Id", c => c.Int(nullable: false, identity: true));
   AddPrimaryKey("dbo.Inventory", "Id");
   AddPrimaryKey("dbo.Orders", "Id");
   AddPrimaryKey("dbo.Customers", "Id");
   AddPrimaryKey("dbo.CreditRisks", "Id");
   CreateIndex("dbo.CreditRisks", new[] { "LastName", "FirstName" }, unique: true, name: "IDX_CreditRisk_Name");
   AddForeignKey("dbo.Orders", "CarId", "dbo.Inventory", "Id");
   AddForeignKey("dbo.Orders", "CustomerId", "dbo.Customers", "Id", cascadeDelete: true);
   //DropColumn("dbo.Inventory", "CarId");
   //DropColumn("dbo.Orders", "OrderId");
   //DropColumn("dbo.Customers", "CustID");
   //DropColumn("dbo.CreditRisks", "CustID");
}
```

Move the last four lines up to the location in **red**.

Create the Final Migration

```
public override void Down()
{
   DropForeignKey("dbo.Orders", "CustomerId", "dbo.Customers");
   DropForeignKey("dbo.Orders", "CarId", "dbo.Inventory");
   DropPrimaryKey("dbo.CreditRisks");
   DropPrimaryKey("dbo.Customers");
   DropPrimaryKey("dbo.Orders");
   DropPrimaryKey("dbo.Inventory");
   DropColumn("dbo.CreditRisks", "Id");
   DropColumn("dbo.Customers", "Id");
   DropColumn("dbo.Orders", "Id");
   DropColumn("dbo.Inventory", "Id");
   AddColumn("dbo.CreditRisks", "CustID", c => c.Int(nullable: false, identity: true));
   AddColumn("dbo.Customers", "CustID", c => c.Int(nullable: false, identity: true));
   AddColumn("dbo.Orders", "OrderId", c => c.Int(nullable: false, identity: true));
   AddColumn("dbo.Inventory", "CarId", c => c.Int(nullable: false, identity: true));
   //DropForeignKey("dbo.Orders", "CustomerId", "dbo.Customers");
   //DropForeignKey("dbo.Orders", "CarId", "dbo.Inventory");
   DropIndex("dbo.CreditRisks", "IDX_CreditRisk_Name");
   //DropPrimaryKey("dbo.CreditRisks");
   //DropPrimaryKey("dbo.Customers");
   //DropPrimaryKey("dbo.Orders");
   //DropPrimaryKey("dbo.Inventory");
   //DropColumn("dbo.CreditRisks", "Id");
   //DropColumn("dbo.Customers", "Id");
   //DropColumn("dbo.Orders", "Id");
   //DropColumn("dbo.Inventory", "Id");
   AddPrimaryKey("dbo.CreditRisks", "CustID");
   AddPrimaryKey("dbo.Customers", "CustID");
   AddPrimaryKey("dbo.Orders", "OrderId");
   AddPrimaryKey("dbo.Inventory", "CarId");
   RenameIndex(table: "dbo.Orders", name: "IX_CustomerId", newName: "IX_CustId");
   RenameColumn(table: "dbo.Orders", name: "CustomerId", newName: "CustId");
   AddForeignKey("dbo.Orders", "CustId", "dbo.Customers", "CustID", cascadeDelete: true);
   AddForeignKey("dbo.Orders", "CarId", "dbo.Inventory", "CarId");
}
```

Move the commented lines up to the location in **red**.

Create the Final Migration

- The final task is to update the database. Type **update-database** in the **Package Manager Console**, and you will get a message that the migration has been applied.

Seeding the Database

- Copy the **Seed()** code from **MyDataInitializer** into the **Seed()** method of the **Configure** class.

- Use the **Package Manager Console** to **update the database** again, and the **Seed()** method executes, populating the database with data.

Adding Repositories for Code Reuse

- A common design pattern for data access is the **Repository pattern**. As described by Martin Fowler, the core of this pattern is to mediate between the **domain** and **data mapping** layers.
- This next section is not meant to be (nor does it pretend to be) a literal interpretation of Mr. Fowler's design pattern. If you are interested in the original pattern that motivated my code, you can find more information on the repository pattern on Martin Fowler's web site at www.martinfowler.com/eaaCatalog/repository.html.

Adding the IRepo Interface

- My version of the pattern starts with an interface that exposes the vast majority of the standard methods you will use in a data access library.
  - Add a new folder in the **AutoLotDAL** project named **Repos**.
  - Add a new interface into the **Repos** folder named **IRepo**.
  - Update the **using** statements to the following:

```
using System.Collections.Generic;
```

  - Add the following to the **IRepo** interface:

```
public interface IRepo<T>
{
   int Add(T entity);
   int AddRange(IList<T> entities);
   int Save(T entity);
   int Delete(int id, byte[] timeStamp);
   int Delete(T entity);
   T GetOne(int? id);
   List<T> GetAll();
   List<T> ExecuteQuery(string sql);
   List<T> ExecuteQuery(string sql, object[] sqlParametersObjects);
}
```

Adding the BaseRepo

- Next, add another class to the **Repos** directory named **BaseRepo**. This class will implement the **IRepo** interface and provide the core functionality for **type-specific repos** (coming next).
- Update the necessary **using** statements.
- Make the class public and implement **IRepo** and **IDisposable**.
  In the constructor, create a new instance of **AutoLotEntities** and set it to a **class-level variable**. The **Set<T>** method points to the **property** on the **context** that is of the same type as **T**, such as the **Cars** property, and is accessible via **Set<Inventory>**. Also add a **protected property** to expose the **context** to **derived classes**.

```
using System;
using System.Collections.Generic;
using System.Data.Entity;
using System.Data.Entity.Infrastructure;
using System.Linq;
using AutoLotDAL.EF;
using AutoLotDAL.Models.Base;
```

```
public class BaseRepo<T> : IDisposable,IRepo<T> where T:EntityBase, new()
{
   private readonly DbSet<T> _table;
   private readonly AutoLotEntities _db;
   public BaseRepo()
   {
      _db = new AutoLotEntities();
      _table = _db.Set<T>();
   }

   protected AutoLotEntities Context => _db;

   public void Dispose()
   {
      _db?.Dispose();
   }
}
```

Implement the SaveChanges() Helper Methods

- Add a method to wrap the context's **SaveChanges()** method.
- There is typically a significant amount of code and error handling code associated with calling these methods, and it is best to write that code only once and place it in the base class.
- The **exception handlers** for the **SaveChanges()** method on the **DbContext** are stubbed out. In a production application, you would need to handle any exceptions accordingly.

```
internal int SaveChanges()
{
   try
   {
      return _db.SaveChanges();
   }
   catch (DbUpdateConcurrencyException ex)
   {
      //Thrown when there is a concurrency error
      //for now, just rethrow the exception
      throw;
   }
   catch (DbUpdateException ex)
   {
      //Thrown when database update fails
      //Examine the inner exception(s) for additional
      //details and affected objects
      //for now, just rethrow the exception
      throw;
   }
   catch (CommitFailedException ex)
   {
      //handle transaction failures here
      //for now, just rethrow the exception
      throw;
   }
   catch (Exception ex)
   {
      //some other exception happened and should be handled
      throw;
   }
}
```

Implement the SaveChanges() Helper Methods

- Creating a new instance of the **DbContext** can be an expensive process from a performance perspective.
- This is sample code that creates a new instance of **AutoLotEntities** with every instance of a **repository**. If this isn't performing as well as you like (or need), then consider using just one **context** class and sharing it among **repositories**.

Retrieving Records

- The **GetOne()** method wraps the **Find()** method of the **DbSet<T>**.
- Similarly, the **GetAll()** method wraps the **ToList()** method.

```
public T GetOne(int? id) => _table.Find(id);
public virtual List<T> GetAll() => _table.ToList();
```

Retrieving Records with SQL

- The last four methods of the interface to implement are the SQL string methods. They pass through the string and parameters to the **DbSet<T>**:

```
public List<T> ExecuteQuery(string sql) => _table.SqlQuery(sql).ToList();
public List<T> ExecuteQuery(string sql, object[] sqlParametersObjects)
   => _table.SqlQuery(sql, sqlParametersObjects).ToList();
```

- You should be extremely careful running **raw SQL strings** against a data store, especially if the string accepts input from a user. Doing so makes your application ripe for **SQL injection attacks**.

Adding Records

- The **Add()** methods are wrappers for the related **Add()** methods on the context. The advantage is encapsulating the **SaveChanges()** method and related error handling.

```
public int Add(T entity)
{
   _table.Add(entity);
   return SaveChanges();
}

public int AddRange(IList<T> entities)
{
   _table.AddRange(entities);
   return SaveChanges();
}
```

- The example code persists the changes to the database every time they are executed. You can certainly modify this code to batch up the changes.

Updating Records

- For the **Save()** method, first set the **EntityState** of the entity to **EntityState.Modified** and then call **SaveChanges()**.
- Setting the **state** ensures that the **context** will propagate the changes to the server.

```
public int Save(T entity)
{
   _db.Entry(entity).State = EntityState.Modified;
   return SaveChanges();
}
```

Deleting Records

- You will add similar code for the **Delete()** method.
  - If the calling code passes in an object, the generic methods in the **BaseRepo** set the state to **EntityState.Deleted** and then call **SaveChanges()**.
  - If the calling code passes in a **key** value and a **timestamp**, a **new object** is created, the **EntityState** is changed to **Deleted**, and then **SaveChanges()** is called.

```
public int Delete(int id, byte[] timeStamp)
{
   _db.Entry(new T() {Id = id, Timestamp = timeStamp}).State = EntityState.Deleted;
   return SaveChanges();
}

public int Delete(T entity)
{
   _db.Entry(entity).State = EntityState.Deleted;
   return SaveChanges();
}
```

Entity-Specific Repos

- This design allows for **entity-specific repos** that provide additional functionality, such as sorted searches or eager fetching. You currently don’t need any, but you will create them for future use. They all follow the same pattern: inherit from **BaseRepo<T>** and add **custom data access methods**.
- Here is the **InventoryRepo** with a customized **GetAll()** method as an example:

```
public class InventoryRepo : BaseRepo<Inventory>
{
   public override List<Inventory> GetAll() => Context.Inventory.OrderBy(x=>x.PetName).ToList();
}
```

Test-Driving AutoLotDAL Take 2

- Now that you have the data access layer completed, it’s time to return to the **AutoLotTestDrive** project and write some more code.

Printing Inventory Records

- Add the following code to the **Main()** function in **Program.cs**:

```
using (var repo = new InventoryRepo())
{
   foreach (Inventory c in repo.GetAll())
   {
      Console.WriteLine(c);
   }
}
```

- You will see the list of cars, sorted by **PetName**.

Adding Inventory Records

- Adding new records shows the simplicity of calling EF using a repository.
- Add a new method, create a new **Inventory** record, and call **Add()** on the repo.

```
private static void AddNewRecord(Inventory car)
{
   // Add record to the Inventory table of the AutoLot database.
   using (var repo = new InventoryRepo())
   {
      repo.Add(car);
   }
}
```

Editing Records

- Saving changes to records is just as simple. Get an **Inventory** object, make some changes, and call **Save()** on the **InventoryRepo** class.

```
private static void UpdateRecord(int carId)
{
   using (var repo = new InventoryRepo())
   {
      // Grab the car, change it, save!
      var carToUpdate = repo.GetOne(carId);
      if (carToUpdate == null) return;
      carToUpdate.Color = "Blue";
      repo.Save(carToUpdate);
   }
}
```

Deleting Records

- Deleting records can be done with an **entity** or with the **key properties** of an entity, in this case the **Id** and **Timestamp**.
- The **Timestamp** is required to uniquely identify a record.

```
private static void RemoveRecordByCar(Inventory carToDelete)
{
   using (var repo = new InventoryRepo())
   {
      repo.Delete(carToDelete);
   }
}

private static void RemoveRecordById(int carId, byte[] timeStamp)
{
   using (var repo = new InventoryRepo())
   {
      repo.Delete(carId, timeStamp);
   }
}
```

Concurrency

- A common problem in multiuser applications is **concurrency** issues. Unless you program with concurrency in mind, if two users update the same record, the last one in wins. This might be perfectly fine for your application, but if not, **EF** and **SQL Server** provide a convenient mechanism for checking for concurrency clashes.
- The **Timestamp data annotation** triggers SQL Server to create a **RowVersion column**. The value in this column is maintained by SQL Server and is updated any time a record is added or updated. It also changes how **EF** builds and runs queries that **update** or **delete** data from the database. For example, a call to delete is no longer just looks for the **primary key** but also includes the **Timestamp** field in the **where** clause. For example, if deleting an **Inventory** record, the generated SQL is updated to something like this:

```
Execute NonQuery "DELETE [dbo].[Inventory] WHERE (([CarId] = @0) AND ([Timestamp] = @1))"
```

Concurrency

- If **two users** retrieve the **same record**, update that record, and then attempt to save, the **first user** will successfully update since the **Timestamp** in their object matches the **Rowversion** of the record (presuming no other action happened on that record). When the **second user** tries to save the record, nothing will happen because the **where** clause fails to locate a record. This triggers a **DbUpdateConcurrencyException** since the number of records to be modified (or deleted) in the **DbChangetracker** (in this example, one record) doesn't match the number of records actually affected (in this example, zero).
- **DbUpdateConcurrencyException** exposes an **Entries** collection that holds all of the entities that **failed** to be processed. Each entry exposes the **original properties**, the **current properties**, and (via a database call) the **current database properties**.

Concurrency

- The following code demonstrates a **DbUpdateConcurrencyException**.
- The two instances of the **InventoryRepo** make sure a concurrency exception is raised.
- What you do with this information is up to the needs of your application. EF and SQL Server provide the tools needed to detect concurrency conflicts.

```
private static void TestConcurrency()
{
   var repo1 = new InventoryRepo();

   //Use a second repo to make sure using a different context
   var repo2 = new InventoryRepo();
   var car1 = repo1.GetOne(1);
   var car2 = repo2.GetOne(1);
   car1.PetName = "NewName";
   repo1.Save(car1);
   car2.PetName = "OtherName";

   try
   {
      repo2.Save(car2);
   }
   catch (DbUpdateConcurrencyException ex)
   {
      var entry = ex.Entries.Single();
      var currentValues = entry.CurrentValues;
      var originalValues = entry.OriginalValues;
      var dbValues = entry.GetDatabaseValues();

      Console.WriteLine(" ******** Concurrency ***********");
      Console.WriteLine("Type\tPetName");
      Console.WriteLine($"Current:\t{currentValues[nameof(Inventory.PetName)]}");
      Console.WriteLine($"Orig:\t{originalValues[nameof(Inventory.PetName)]}");
      Console.WriteLine($"db:\t{dbValues[nameof(Inventory.PetName)]}");
   }
}
```

```
***** Concurrency *****
Type: PetName
Current: OtherName
Orig: Zippy
Db: NewName
```

Interception

- The final topic in this chapter regarding **EF** covers **interception**.
- As you have seen in the previous examples, a lot of “magic” happens behind the scenes for the data to move from the data store into your object model, and vice versa.
- **Interception** is the process of running code at different phases of the process.

The IDbCommandInterceptor Interface

- It all starts with the **IDbCommandInterceptor** interface, listed here:

```
public interface IDbCommandInterceptor : IDbInterceptor
{
   void NonQueryExecuted(DbCommand command, DbCommandInterceptionContext<int> interceptionContext);
   void NonQueryExecuting(DbCommand command, DbCommandInterceptionContext<int> interceptionContext);
   void ReaderExecuted(DbCommand command, DbCommandInterceptionContext<DbDataReader> interceptionContext);
   void ReaderExecuting(DbCommand command, DbCommandInterceptionContext<DbDataReader> interceptionContext);
   void ScalarExecuted(DbCommand command, DbCommandInterceptionContext<object> interceptionContext);
   void ScalarExecuting(DbCommand command, DbCommandInterceptionContext<object> interceptionContext);
}
```

- As you can probably infer from the names, this interface contains methods that are called by **EF** just **prior** and just **after** certain **events**.
- For example, the **ReaderExecuting()** method is called just ***before*** a reader is executed, and **ReaderExecuted()** is called just ***after*** a reader is executed.
- For this example, you will simply write to the console in each of these methods. In a production system, the logic will be more appropriate to your requirements.

Adding Interception to AutoLotDAL

- Add a new folder named **Interception** to the **AutoLotDAL** project and a new class to the folder named **ConsoleWriterInterceptor**. Make the class **public**, add **System.Data.Entity.Infrastructure.Interception** as a **using** statement, and inherit from **IDbCommandInterceptor**.

```
public class ConsoleWriterInterceptor : IDbCommandInterceptor
{
   public void NonQueryExecuting(DbCommand command, DbCommandInterceptionContext<int> interceptionContext)
   {
   }
   public void NonQueryExecuted(DbCommand command, DbCommandInterceptionContext<int> interceptionContext)
   {
   }
   public void ReaderExecuting(DbCommand command, DbCommandInterceptionContext<DbDataReader> interceptionContext)
   {
   }
   public void ReaderExecuted(DbCommand command, DbCommandInterceptionContext<DbDataReader> interceptionContext)
   {
   }
   public void ScalarExecuting(DbCommand command, DbCommandInterceptionContext<object> interceptionContext)
   {
   }
   public void ScalarExecuted(DbCommand command, DbCommandInterceptionContext<object> interceptionContext)
   {
   }
}
```

Adding Interception to AutoLotDAL

- To keep the example simple, you are just going to write to the console whether the call is asynchronous and the text of the command.
- Add a **using** for **static System.Console** and add a private method named **WriteInfo()** that takes a bool and a string.

```
private void WriteInfo(bool isAsync, string commandText)
{
   WriteLine($"IsAsync: {isAsync}, Command Text: {commandText}");
}
```

- In each of the methods from the interface, add a call to the **WriteInfo()** method like this:

```
WriteInfo(interceptionContext.IsAsync,command.CommandText);
```

Registering the Interceptor

- **Interceptors** can be **registered** through code or in the application configuration file.
    - Registering them in code isolates them from changes to the configuration file and therefore ensures that they are always registered.
    - If you need more flexibility, the configuration file might be the better choice.
- Open the **AutoLotEntities** class and add the following using statements:

```
using System.Data.Entity.Infrastructure.Interception;
```

- Next, in the constructor, add the following line of code:

```
DbInterception.Add(new ConsoleWriterInterceptor());
```

- Execute one of the test methods from earlier in this chapter, and you will see the additional output from the logger written to the console. This is a simple example but illustrates the capabilities of the **interceptor** class.
- The **DbCommandInterceptionContext<T>** contains much more than you have explored here. Please consult the .NET Framework 4.7 SDK documentation for more information.

Adding the DatabaseLogger Interceptor

- **EF** now ships with a **built-in logging interceptor** if all you want to do is simple logging. To add this capability, start by opening the **AutoLotEntities** class and **comment out your console logger**. Add a **static readonly** member of type **DatabaseLogger** (from the **System.Data.Entity.Infrastructure.Interception** namespace). The constructor takes two parameters; the first is the **file name** for the log file, and the second is optional and indicates whether the log should be **appended** to (the default is **false**). In the constructor, call **StartLogging()** on the interceptor and add the instance to the list of interceptors.

```
static readonly DatabaseLogger DatabaseLogger = new DatabaseLogger("sqllog.txt", true);
public AutoLotEntities() : base("name=AutoLotConnection")
{
   //DbInterception.Add(new ConsoleWriterInterceptor());
   DatabaseLogger.StartLogging();
   DbInterception.Add(DatabaseLogger);
}
```

- The last change is to leverage the **DbContext** implementation of the **IDisposable** pattern to stop logging and remove the interceptor.

```
protected override void Dispose(bool disposing)
{
   DbInterception.Remove(DatabaseLogger);
   DatabaseLogger.StopLogging();
   base.Dispose(disposing);
}
```

ObjectMaterialized and SavingChanges Events

- Creating a **custom interceptor** can provide a significant amount of functionality but can also involve a lot of work.
- Two of the most common scenarios are handled by two of the events on the **ObjectContext** class, **ObjectMaterialized** and **SavingChanges**.
  - The **ObjectMaterialized** event fires when an **object is reconstituted from the data store**, just before the instance is returned to the calling code.
  - The **SavingChanges** event occurs when the **object's data is about to be propagated to the data store**, just after the **SaveChanges()** method is called on the **context**.

Accessing the Object Context

- The **DbContext** doesn't expose the **ObjectContext** directly, but it does implement the **IObjectContextAdapter** interface, which grants access to the **ObjectContext**.
- To get to the **ObjectContext**, you need to cast **AutoLotEntities** to **IObjectContextAdapter**.
- Update the **using** statements.
- Update the **constructor** and add the two event handlers.

```
using System;
using System.Data.Entity;
using System.Data.Entity.Core.Objects;
using System.Data.Entity.Infrastructure;
using System.Data.Entity.Infrastructure.Interception;
using AutoLotDAL.Interception;
using AutoLotDAL.Models;
```

```
public AutoLotEntities(): base("name=AutoLotConnection")
{
   //Interceptor code
   var context = (this as IObjectContextAdapter).ObjectContext;
   context.ObjectMaterialized += OnObjectMaterialized;
   context.SavingChanges += OnSavingChanges;
}

private void OnSavingChanges(object sender, EventArgs eventArgs)
{
}

private void OnObjectMaterialized(object sender, System.Data.Entity.Core.Objects.ObjectMaterializedEventArgs e)
{
}
```

ObjectMaterialized

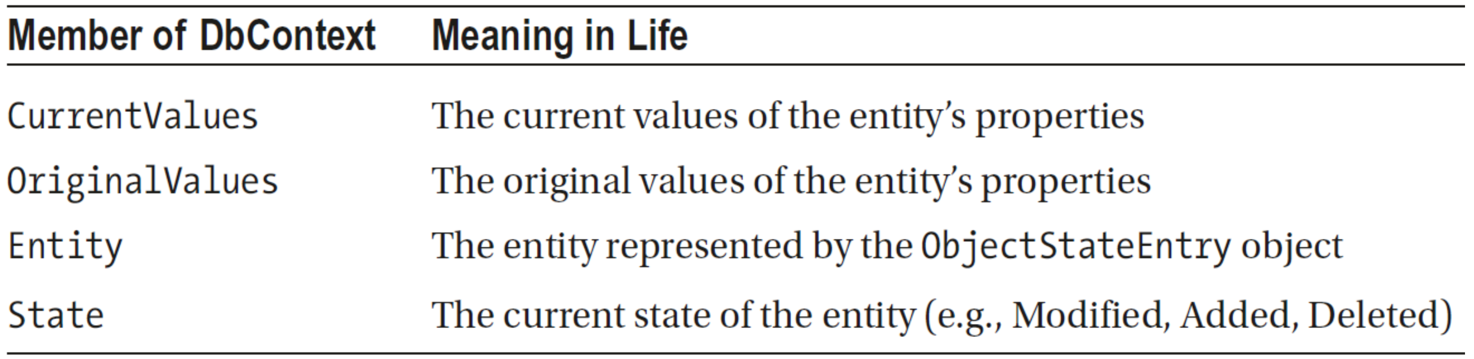
- The **ObjectMaterialized** event's arguments provide access to the entity being reconstituted.
- This event fires immediately after a model's properties are populated by **EF** and just before the **context** serves it up to the calling code.
- This event is invaluable when working with **WPF**.

SavingChanges

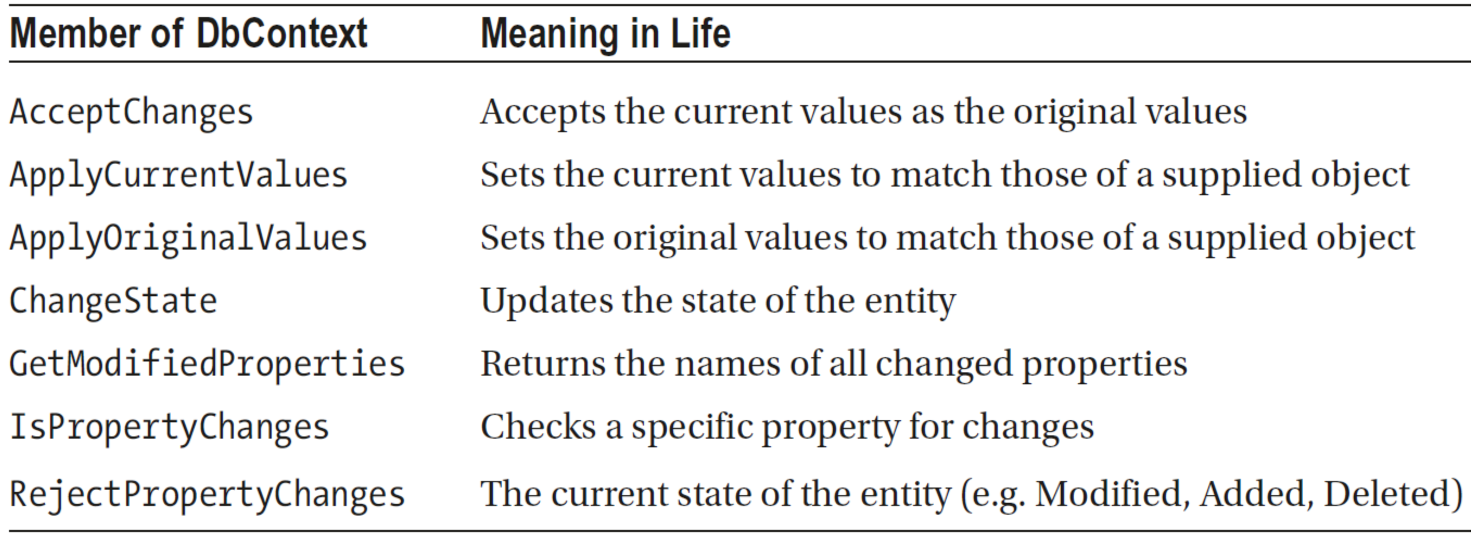
- The **SavingChanges** event fires just after the **SaveChanges()** method is called (on the **DbContext**) but before the database is updated.

- By accessing the **ObjectContext** passed into the event handler, all the entities in the transaction are accessible through the **ObjectStateEntry** property on the **DbContext**.

- The **ObjectStateEntry** also exposes a set of **methods** that can be used on the **entity**.



Member of DbContext	Meaning in Life
`CurrentValues`	The current values of the entity's properties
`OriginalValues`	The original values of the entity's properties
`Entity`	The entity represented by the `ObjectStateEntry` object
`State`	The current state of the entity (e.g., Modified, Added, Deleted)



Member of DbContext	Meaning in Life
`AcceptChanges`	Accepts the current values as the original values
`ApplyCurrentValues`	Sets the current values to match those of a supplied object
`ApplyOriginalValues`	Sets the original values to match those of a supplied object
`ChangeState`	Updates the state of the entity
`GetModifiedProperties`	Returns the names of all changed properties
`IsPropertyChanges`	Checks a specific property for changes
`RejectPropertyChanges`	The current state of the entity (e.g. Modified, Added, Deleted)

SavingChanges

- This permits you to write code that rejects any changes to a vehicle’s color if the color is red, like this:

```
private void OnSavingChanges(object sender, EventArgs eventArgs)
{
   // Sender is of type ObjectContext. Can get current and original values,
   // and cancel/modify the save operation as desired.
   var context = sender as ObjectContext;
   if (context == null) return;

   foreach (ObjectStateEntry item in
      context.ObjectStateManager.GetObjectStateEntries(EntityState.Modified | EntityState.Added))
   {
      // Do something important here
      if ((item.Entity as Inventory) != null)
      {
         var entity = (Inventory) item.Entity;
         if (entity.Color == "Red")
         {
            item.RejectPropertyChanges(nameof(entity.Color));
         }
      }
   }
}
```

Splitting the Models From the Data Access Layer

- Currently, the **data access layer** is all in one project, with all the **EF** code and the **models** grouped together. While that is sufficient for many applications, sometimes (as WPF MMVM and ASP.NET), it is better to split out the **models** from the **EF code**.
- Fortunately, this is extremely simple.
  - Add a new **class library project** named **AutoLotDAL.Models** to your solution and delete the generated **Class1.cs** class.
  - Add a reference to **System.ComponentModel.DataAnnotations** and then use **NuGet Package Manager** to add **EntityFramework** to the project.
  - Move all the **files and folders** from the **Models** directory of the **AutoLotDAL** project into the **new project** and **adjust the namespaces**.
  - Add a reference from **AutoLotDAL** to **AutoLotDAL.Models** and add a reference from **AutoLotTestDrive** to **AutoLotDAL.Models**.

Deploying to SQL Server Express

- SQL Server's **LocalDb** is great for development and local testing, but at some point, you will need to move your database to another instance.
- In this example, you will deploy your database to **SQL Server Express**, which is useful for development but can also be **used by more than one person** and supports **local services** (such as **IIS** and **WCF**). There are two easy ways to do this, and you will explore them both next.
- If you have not already installed **SQL Server Express 2019**, you can download the installer from here: https://www.microsoft.com/en-us/sql-server/sql-server-editions-express.

Deploying to SQL Server Express using Migrations

- Once you have your database ready to deploy to another instance of SQL Server, the first mechanism (if you have access to the instance that you are deploying to) is as simple as **changing the connection string and executing update-database**!
- Open the **App.config** file in **AutoLotDAL** and update the **connection string** to point to **SQL Server Express**. The exact string will depend on how you installed **SQL Server Express**, but it should look something like this:

```
<connectionStrings>
   <add name="AutoLotConnection" connectionString="data source=.\SQLEXPRESS2016;
        initial catalog=AutoLot;integrated security=True;MultipleActiveResultSets=True;App=EntityFramework"
        providerName="System.Data.SqlClient" />
</connectionStrings>
```

- When you run **update-database**, you will see the updates getting applied. You can also specify a different **connection string** in the **update-database** command.
- Instead of changing the **connection string** in the **App.config** file, you could run the following command in the **Package Manager Console**:

```
Update-Database -ProjectName AutolotDAL -ConnectionString "data source=.\SQLEXPRESS2016;
initial catalog=AutoLot;integrated security=True;MultipleActiveResultSets=True;
App=EntityFramework" -ConnectionProviderName "System.Data.SqlClient"
```

- There are many more options available for **update-database**. To get the full list of them (and any other EF commands), use **get-help update-database**.

Creating a Migration Script

- Changing the **connection string** is certainly simple, but what if you don't have access to the database and everything must go through a **DBA**?
- **EF** has you covered for that as well. The **update-database** command can also create a **SQL script** to do all of the changes for you. Adding the **-script** parameter will examine the target database and **create a script** to any migrations that have not yet been applied.
- Enter the following command into the **Package Manager Console** (note I changed the **catalog name** so that a file will be produced):

```
Update-Database -ProjectName AutolotDAL -ConnectionString "data source=.\SQLEXPRESS2016;
initial catalog=AutoLot2;integrated security=True;MultipleActiveResultSets=True;
App=EntityFramework" -ConnectionProviderName "System.Data.SqlClient" -script
```

- This will create a **script** that you can send off to your **DBA** to execute for you. The one downside is that when you create a script like this, the **Seed()** method of the **Configuration** class is **not executed or included in the SQL script**. You will have to seed the database with initial data in some other manner.