Entity
Framework
Code First

Succinctly

by Ricardo Peres

Syncfusion | Technology Resource Portal

Entity Framework Code First Succinctly

By
Ricardo Peres

Foreword by Daniel Jebaraj

Entity Framework Code First Succinctly

Copyright © 2014 by Syncfusion Inc.

2501 Aerial Center Parkway

Suite 200

Morrisville, NC 27560

USA

All rights reserved.

Important licensing information. Please read.

Table of Contents

Entity Framework Code First Succinctly

The Story Behind the Succinctly Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to "enable AJAX support with one click," or "turn the moon to cheese!"

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and "Like" us on Facebook to help us spread the word about the *Succinctly* series!

About the Author

Ricardo Peres is a Portuguese developer who has been working with .NET since 2001. He's a technology enthusiast and has worked in many areas, from games to enterprise applications. His main interests currently are enterprise application integration and web technologies. For the last 10 years he has worked for a multinational Portuguese-based company called Critical Software. He keeps a blog on technical subjects at http://weblogs.asp.net/ricardoperes and can be followed on Twitter at @rjperes75.

Introduction

Object/Relational mappers (ORMs) exist to bridge a gap between object-oriented programming (OOP) and relational databases. At the expense of being less specific, ORMs abstracts away database-specific technicalities and hides from you, the OOP developer, those scary SQL queries.

Entity Framework Code First is the latest edition of Microsoft's flagship data access technology. It sits on the "classic" Entity Framework, which has existed since 2009. Entity Framework already offered two development models:

- Database first, which generated code from an existing database.
- Model first, which defined a conceptual model from which both the database and the code were generated.

Code First picks up where "classic" left off: starting by code and generating the database from it, which is known as a domain-driven design (DDD) approach. It also offers a much simpler and streamlined API, which has gained a great deal of well-deserved attention.

Since Entity Framework was first included in Visual Studio 2008 and the .NET Framework 3.5 SP1, and certainly object/relational mapping existed long before that, then why is there all this hype around Entity Framework Code First (EFCF)? Well, it seems that EFCF is the new cool kid on the block for a number of reasons:

- Easy to set up: you just pop up NuGet's package manager and you're done.
- Simple to use: there are no XML schemas to master, no base classes to inherit from, no arcane interfaces to implement, and it has a clean, tidy API. You just focus on the actual domain model and its characteristics, and forget about the persistence details, which is pretty much what domain driven design (DDD) is about.
- It sits on an API for database access that you can expect to see more support and improvement for by Microsoft.
- Because it is not tied to the regular .NET framework releases, new versions come out much more often.
- Microsoft hit the bull's eye when it decided to release EFCF's source code and to start accepting community requests and even pull requests: bugs are fixed more quickly, you can influence the features the product will have, and you have the chance to try out the latest improved functionality.

For those coming from "classic" Entity Framework, this means that you have to code your own entities by hand. There is no fancy designer here. This actually gives you more control over how things are generated, and it is not a bad thing.

You can make your own decision. Stick with me and let's start exploring Entity Framework Code First.

Chapter 1 Setting Up

Before We Start

Before you start using EFCF, you need to have its assemblies deployed locally. The distribution model followed by Microsoft and a number of other companies does not depend on old school Windows installers, but instead relies on new technologies such as NuGet and Git. We'll try to make sense of each of these options in a moment, but before we get to that, make sure you have Visual Studio 2012 installed (any edition including Visual Web Developer Express will work), as well as SQL Server 2008 (any edition including Express) or higher. On SQL Server, create a new database called **Succinctly**.

Getting Entity Framework Code First From NuGet

NuGet is to .NET package management what Entity Framework is to data access. In a nutshell, it allows Visual Studio projects to have dependencies on software packages—assemblies, source code files, PowerShell scripts, etc.—stored in remote repositories. EFCF comes in its own assembly, which is deployed out-of-band between regular .NET releases. In order to install it to an existing project, first run the **Package Manager Console** from the **Tools** – **Library Package Manager** and enter the following command.

```
PM> Install-Package EntityFramework
```

This is by far the preferred option for deploying Entity Framework Code First.

> 💡 *Tip: This will only work with an existing project, not on an empty solution.*

Getting Entity Framework Code First From CodePlex

The second option, for advanced users, is to clone the Entity Framework Code First repository on CodePlex, build the binaries yourself, and manually add a reference to the generated assembly.

First things first, let's start by cloning the Git repository using your preferred Git client.

```
git clone https://git01.codeplex.com/entityframework.git
```

Next, build everything from the command line using the following two commands.

```
build /t:RestorePackages /t:EnableSkipStrongNames
build
```

You can also fire up Visual Studio 2012 and open the **EntityFramework.sln** solution file. This way, you can do your own experimentations with the source code, compile the debug version of the assembly, run the unit tests, etc.

Configuring the Database

Entity Framework is database-agnostic, but the standard version only includes providers for Microsoft technologies. This means only SQL Server 2005+, SQL Server Compact Edition, and SQL Server Express LocalDB are supported. The examples in this book will work on any of these editions. Make sure you have one of them installed and you have the appropriate administrative permissions.

Entity Framework decides on what connection to use by executing the following algorithm.

- If a connection string is passed in the DbContext's constructor, then it will try to use that connection string with the default connection factory.
- If the parameter-less constructor is used, it will look for a connection string in the configuration file, where its name is the same as the context's class.
- If no connection string is passed and no connection string with an appropriate name is found in the connection string, it will try to connect to a SQL Server instance named SQLEXPRESS, and a database with the same name as the context class, including namespace.

A connection factory is an implementation of IDbConnectionFactory that sits in a well-known location: Database.DefaultConnectionFactory. This instance can be explicitly set, and should be if a specific database engine requires it. This can be done either by code or by setting a value in the configuration file.

```xml
<entityFramework>
  <defaultConnectionFactory

type="System.Data.Entity.Infrastructure.SqlCeConnectionFactory, EntityFramework" />
</entityFramework>
```

SQL Server

For connecting to the SQL Server, no special action is required. The default Database.DefaultConnectionFactory is already an instance of SqlConnectionFactory.

If you want to have a connection string in the configuration file, you should use the provider name **"System.Data.SqlClient"** as per the following example.

```
<connectionStrings>
  <add name="Succinctly"
  connectionString="Data Source=.\SQLEXPRESS;Integrated Security=SSPI;
  Initial Catalog=Succinctly;MultipleActiveResultSets=true"
  providerName="System.Data.SqlClient" />
</connectionStrings>
```

SQL Server Compact Edition

SQL Server Compact Edition (SQLCE) is a small footprint, free and embedded database, which supports practically the same SQL syntax as its full featured sibling. If you want to use it, make sure you have the SQL Server Compact Edition installed; the download is available at http://www.microsoft.com/en-us/sqlserver/editions/2012-editions/compact.aspx.

> **Tip: SQLCE will only accept a single connection at a time.**

If you want to connect to SQLCE, you need to register a connection string using the **System.Data.SqlServerCe.4.0** provider.

```
<connectionStrings>
  <add name="Succinctly"
  connectionString="Data Source=Succinctly.sdf"
  providerName="System.Data.SqlServerCe.4.0" />
</connectionStrings>
```

If you want to pass the full connection string as parameter to the context, make sure you set the default connection factory to a SqlCeConnectionFactory instance, by using the following code.

```
Database.DefaultConnectionFactory = new SqlCeConnectionFactory
("System.Data.SqlServerCe.4.0");
```

Or by the following configuration.

```
<entityFramework>
  <defaultConnectionFactory
  type="System.Data.Entity.Infrastructure.SqlCeConnectionFactory, EntityFramework">
    <parameters>
      <parameter value="System.Data.SqlServerCe.4.0" />
    </parameters>
  </defaultConnectionFactory>
</entityFramework>
```

SQLCE will look for and create a file named *<database>*.sdf in the Bin directory of your project.

SQL Server 2012 Express LocalDB

The LocalDB database released with SQL Server 2012 Express and it is also a small footprint, fully featured server that doesn't use any services. If you don't have it already, you can download the installer from http://www.microsoft.com/en-us/download/details.aspx?id=29062.

Tip: LocalDB will only accept a single connection at a time.

For connecting to a LocalDB instance, you will need a connection string very similar to one you would use to connect to SQL Server, including the provider name. Instead of a SQL Server instance, you specify the version of LocalDB to use.

```
<connectionStrings>
  <add name="Succinctly"
  connectionString="Data Source=(localdb)\v11.0;Integrated Security=SSPI;
  Initial Catalog=Succinctly;MultipleActiveResultSets=true"
  providerName="System.Data.SqlClient" />
</connectionStrings>
```

There is no need to configure a default connection factory since LocalDB uses the same as SQL Server, which is the default.

LocalDB will look for and use a database file named *<database>*.mdf and a transaction log *<database>*_log.ldf, both located in folder %USERPROFILE%, unless explicitly located somewhere else, by specifying an AttachDBFilename parameter.

```
<connectionStrings>
  <add name="Succinctly"
  connectionString="Data Source=(localdb)\v11.0;Integrated Security=SSPI;
  MultipleActiveResultSets=true;AttachDBFilename=C:\Windows\Temp\Succinctly.mdf"
  providerName="System.Data.SqlClient" />
</connectionStrings>
```

Note: LocalDB files are fully compatible with SQL Server ones.

Chapter 2 Domain Model

Scenario

Let's consider the following scenario as the basis for our study.
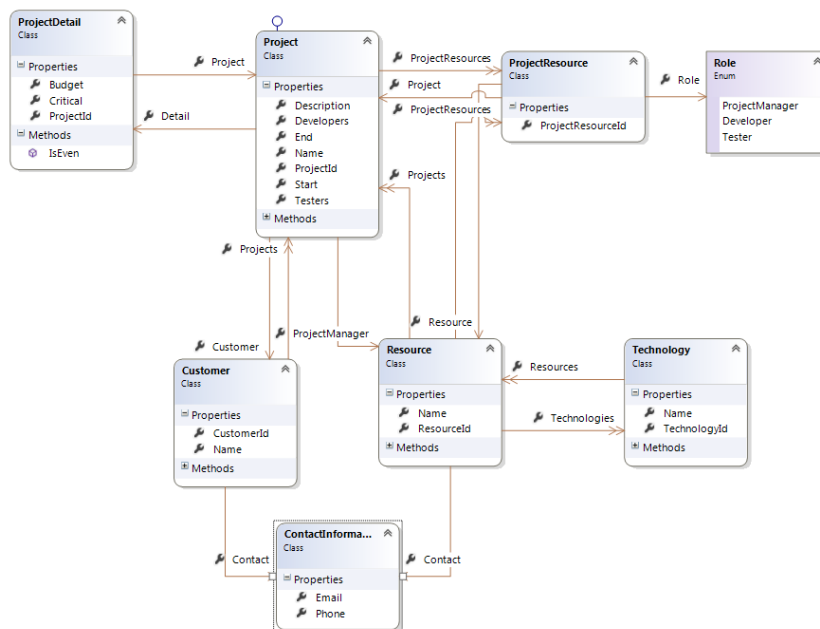
Figure 1: The domain model

You will find all these classes in the accompanying source code. Let's try to make some sense out of them:

- A Customer has a number of Projects.
- Each Project has a collection of ProjectResources, belongs to a single Customer, and has a ProjectDetail with additional information.
- A ProjectDetail refers to a single Project.
- A ProjectResource always points to an existing Resource and is assigned to a Project with a given Role.
- A Resource knows some Technologies and can be involved in several Projects.
- A Technology can be collectively shared by several Resources.
- Both Customers and Resources have Contact information.

> **Note: You can find the full source code in the following Git repository:**
> **https://bitbucket.org/syncfusiontech/entity-framework-code-first-succinctly/overview**

Core Concepts

Before a class model can be used to query a database or to insert values into it, Entity Framework needs to know how it should translate code (classes, properties, and instances) back and forth into the database (specifically, tables, columns and records). For that, it uses a mapping, for which two APIs exist. More on this later, but first, some fundamental concepts.

Contexts

A context is a class that inherits from DbContext and which exposes a number of entity collections in the form of DbSet<T> properties. Nothing prevents you from exposing all entity types, but normally you only expose aggregate roots, because these are the ones that make sense querying on their own.

An example context might be the following.

```
public class ProjectsContext : DbContext
{
  public DbSet<Tool> Tools { get; set; }
  public DbSet<Resource> Resources { get; set; }
  public DbSet<Project> Projects { get; set; }
  public DbSet<Customer> Customers { get; set; }
  public DbSet<Technology> Technologies { get; set; }
}
```

Tip: Do notice that both setters and getters for the entity collections are public.

Note: Feel free to add your own methods, business or others, to the context class.

The DbContext class offers a number of constructors, mostly for configuring the connection string:

- If the no-arguments constructor is called, the DbContext will assume that a connection string with the same name as the context's class will exist in the configuration file.
- There's also a constructor that takes a single string parameter. This parameter will either be a full connection string that is specific to the current database provider or a name of a connection string that must be present in the configuration file.
- For the sake of completeness, another constructor exists that takes an existing DbConnection; Entity Framework might not take full control of this connection, for example, it won't try to dispose of it when no longer needed.

```
public class ProjectsContext : DbContext
{
  public ProjectsContext() { }

public ProjectsContext(String nameOrConnectionString): base(nameOrConnectionString)
  { }
```

```
   public ProjectsContext(DbConnection existingConnection,
 Boolean contextOwnsConnection): base(existingConnection, contextOwnsConnection)
   { }
}
```

If we use the constructor overload that takes a connection string by its name, we must use the format "Name=Some Name".

```
public class ProjectsContext : DbContext
{
   public ProjectsContext(String name) : base("Name=AnotherName") { }
}
```

Entities

At the very heart of the mapping is the concept of entity. An entity is just a class that is mapped to an Entity Framework context and which has an identity, or a property that uniquely identifies instances of it. In DDD parlance, it is said to be an aggregate root if it is meant to be directly queried, think of a Project or a Customer, or an entity if it is loaded together with an aggregate root and not generally considerable on its own, such as project details or customer address. An entity is persisted on its own table and may have any number of business or validation methods.

```
public class Project
{
   public Project()
   {
      this.ProjectResources = new HashSet<ProjectResource>();
   }

   public Int32 ProjectId { get; set; }

   public String Name { get; set; }

   public DateTime Start { get; set; }

   public DateTime? End { get; set; }

   public virtual ProjectDetail Detail { get; set; }

   public virtual Customer Customer { get; set; }

   public void AddResource(Resource resource, Role role)
   {
      resource.ProjectResources.Add(new ProjectResource()
      { Project = this, Resource = resource, Role = role });
   }

   public Resource ProjectManager
   {
      get
```

```
      {
        return (this.ProjectResources.ToList()
        .Where(x => x.Role == Role.ProjectManager)
        .Select(x => x.Resource).SingleOrDefault());
      }
    }

    public IEnumerable<Resource> Developers
    {
      get
      {
        return (this.ProjectResources.Where(x => x.Role == Role.Developer)
        .Select(x => x.Resource).ToList());
      }
    }

    public IEnumerable<Resource> Testers
    {
      get
      {
        return (this.ProjectResources.Where(x => x.Role == Role.Tester)
        .Select(x => x.Resource)).ToList();
      }
    }

    public virtual ICollection<ProjectResource> ProjectResources { get;
 protected set;
    }

    public override String ToString()
    {
      return (this.Name);
    }
}
```

Here you can see some patterns that we will be using throughout the book:

- An entity needs to have at least a public parameter-less constructor.
- An entity always has an identifier property, which has the same name and ends with **Id**.
- Collections are always generic, have protected setters, and are given a value in the constructor in the form of an actual collection (like HashSet<T>).
- Calculated properties are used to expose filtered sets of actually persisted properties.
- Business methods are used for enforcing business rules.
- A textual representation of the entity is supplied by overriding ToString.

A domain model where its entities have only properties (data) and no methods (behavior) is sometimes called an anemic domain model. You can find a good description for this anti-pattern on Martin Fowler's web site: http://www.martinfowler.com/bliki/AnemicDomainModel.html.

Complex Types

A complex type is also a class with some properties and maybe methods, but unlike an entity, it doesn't have an identity property and doesn't have its own table for persistence. Instead, its properties are saved into the same table as its declaring type. A complex type is useful for grouping properties that conceptually should always appear together, such as the city, country, street, and zip code in an address. By reusing complex types, we can have the same logic repeated in different entities. Both a customer and a human resource might have contact information with the same structure:

```csharp
public class ContactInformation
{
  public String Email { get; set; }

  public String Phone { get; set; }
}

public class Resource
{
  public ContactInformation Contact { get; set; }
}

public class Customer
{
  public ContactInformation Contact  { get; set; }
}
```

Complex types have the following limitations:

- They cannot have navigation properties (references or collections, see next topics).
- They cannot be null, or their containing entity must initialize them.
- They cannot point to their containing entity.

Scalar Properties

Scalars are simple values, like strings, dates, and numbers. They are where actual entity data is stored, and can be of one of any of these types.

.NET Type	SQL Server Type	Description
Boolean	BIT	Single bit.
Byte	TINYINT	Single byte (8 bits).
Char	CHAR,	ASCII or UNICODE char (8 or 16 bits).

.NET Type	SQL Server Type	Description
	NCHAR	
Int16	SMALLINT	Short integer (16 bits).
fInt32	INT	Integer (32 bits).
Int64	BIGINT	Long (64 bits).
Single	REAL	Floating point number (32 bits).
Double	FLOAT	Double precision floating point number (64 bits).
Decimal	MONEY, SMALLMONEY	Currency (64 bits) or small currency (32 bits).
Guid	UNIQUEIDENTIFIER	Globally Unique Identifier (GUID).
DateTime	DATE, DATETIME, SMALLDATETIME, DATETIME2	Date with or without time.
DateTimeOffset	DATETIMEOFFSET	Date and time with timezone information.
TimeSpan	TIME	Time.
String	VARCHAR, NVARCHAR, XML	ASCII (8 bits per character), UNICODE (16 bits) or XML character string. Can also represent a Character Long Object (CLOB).
Byte[]	BINARY,	Binary Large Object (BLOB).

.NET Type	SQL Server Type	Description
	VARBINARY, ROWVERSION	
Enum	INT	Enumerated value.
DbGeography	GEOGRAPHY	Geography spatial type.
DbGeometry	GEOMETRY	Planar spatial type.

The types Byte, Char, and String can have a maximum length specified. A value of **-1** translates to MAX.

All scalar types can be made nullable, meaning they might have no value set. In the database, this is represented by a NULL column.

Scalar properties need to have both a getter and a setter, but the setter can have a more restricted visibility than the getter: internal, protected internal or protected.

Some examples of scalar properties are as follows.

```
public class Project
{
  public Int32 ProjectId { get; set; }

  public String Name { get; set; }

  public DateTime Start { get; set; }

  public DateTime? End { get; set; }
}
```

Identity Properties

One or more of the scalar properties of your entity must represent the underlying table's primary key, which can be single or composite.

Primary key properties can only be of any of the basic types, which is any type in the list above except arrays and enumerations, but no complex types or other entity's types.

References

A reference from an entity to another defines a bidirectional relation. There are two types of reference relations:

- Many-to-one: several instances of an entity can be associated with the same instance of another type (such as projects that are owned by a customer).
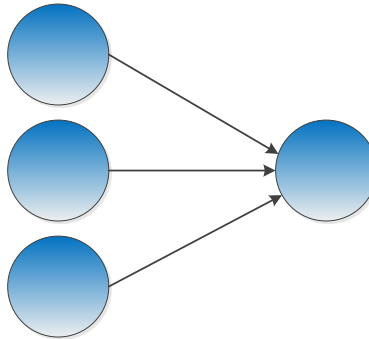
Figure 2: Many-to-one relationship

- One-to-one: an instance of an entity is associated with another instance of another entity; this other instance is only associated with the first one (such as a project and its detail).

Figure 3: One-to-one relationship

In EFCF, we represent an association by using a property of the other entity's type.
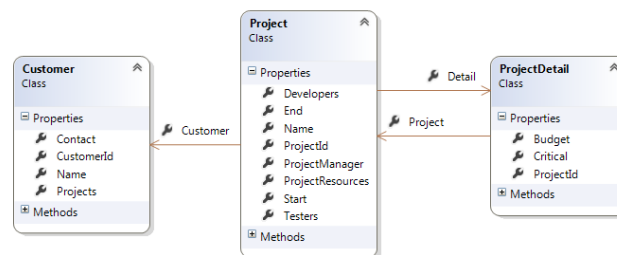
Figure 4: References: one-to-one, many-to-one

We call an entity's property that refers to another entity as an endpoint of the relation between the two entities.

```
public class Project
{
  //one endpoint of a many-to-one relation
  public virtual Customer Customer { get; set; }

  //one endpoint of a one-to-one relation
  public virtual ProjectDetail Detail { get; set; }
}
```

```
public class ProjectDetail
{
  //the other endpoint of a one-to-one relation
  public Project Project { get; set; }
}

public class Customer
{
  //the other endpoint of a many-to-one relation
  public virtual ICollection<Project> Projects { get; protected set; }
}
```

> **Note: By merely looking at one endpoint, we cannot immediately tell what its type is (one-to-one or many-to-one), we need to look at both endpoints.**

Collections

Collections of entities represent one of two possible types of bidirectional relations:

- One-to-many: a single instance of an entity is related to multiple instances of some other entity's type (such as a project and its resources).
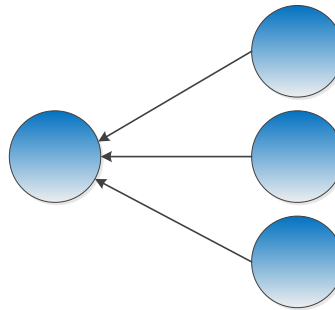
Figure 5: One-to-many relationship

- Many-to-many: possibly a number of instances of a type can be related with any number of instances of another type (such as resources and the technologies they know).
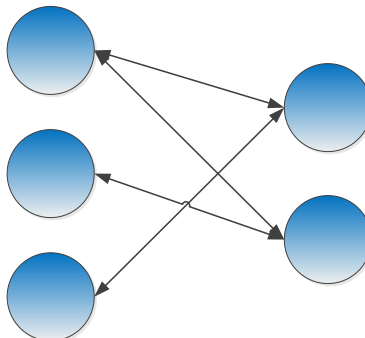
Figure 6: Many-to-many relationship
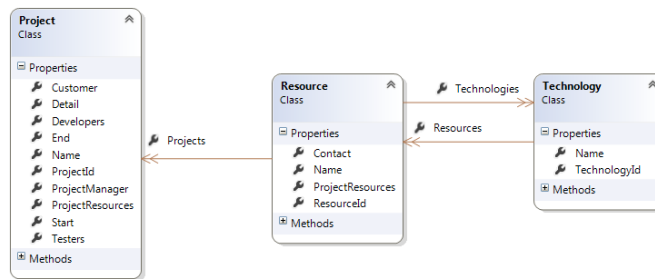
Figure 7: Collections: one-to-many, many-to-many

Entity Framework only supports declaring collections as ICollection<T> (or some derived class or interface) properties. In the entity, we should always initialize the collections properties in the constructor.

```
public class Project
{
  public Project()
  {
    this.ProjectResources = new HashSet<ProjectResource>();
  }

  public virtual ICollection<ProjectResource> ProjectResources
  { get; protected set; }
}
```

> **Note: References and collections are collectively known as navigation properties, as opposed to scalar properties.**

Mapping by Attributes

Overview

Probably the most used way to express our mapping intent is to apply attributes to properties and classes. This has the advantage that, by merely looking at a class, one can immediately infer its database structure.

Schema

Unless explicitly set, the table where an entity type is to be stored is determined by a convention (more on this later on), but it is possible to set the type explicitly by applying a TableAttribute to the entity's class.

```
[Table("MY_SILLY_TABLE", Schema = "dbo")]
public class MySillyType { }
```

The Schema property is optional and should be used to specify a schema name other than the default. A schema is a collection of database objects (tables, views, stored procedures, functions, etc.) in the same database. In SQL Server, the default schema is dbo**.**

For controlling how a property is stored (column name, physical order, and database type), we apply a ColumnAttribute.

```
[Column(Order = 2, TypeName = "VARCHAR")]
public String Surname { get; set; }
[Column(Name = "FIRST_NAME", Order = 1, TypeName = "VARCHAR")]
public String FirstName { get; set; }
```

If the TypeName is not specified, Entity Framework will use the engine's default for the property type. SQL Server will use NVARCHAR for String properties, INT for Int32, BIT for Boolean, etc. We can use it for overriding this default.

The Order applies a physical order to the generated columns that might be different from the order by which properties appear on the class. When the Order property is used, there should be no two properties with the same value in the same class.

Marking a scalar property as required requires the usage of the RequiredAttribute.

```
[Required]
public String Name { get; set; }
```

Tip: When this attribute is applied to a String property, it not only prevents the property from being null, but also from taking an empty string.

Tip: For value types, the actual property type should be chosen appropriately. If the column is non-nullable, one should not choose a property type that is nullable, such as Int32?.

For a required associated entity, it is exactly the same.

```
[Required]
public Customer Customer { get; set; }
```

Setting the maximum allowed length of a string column is achieved by means of the MaxLengthAttribute.

```
[MaxLength(50)]
public String Name { get; set; }
```

The MaxLengthAttribute can be also used to set a column as being a CLOB, a column containing a large amount of text. SQL Server uses the types NVARCHAR(MAX) and VARCHAR(MAX). For that, we pass a length of -1.

```
[MaxLength(-1)]
public String LargeText { get; set; }
```

It can also be used to set the size of a BLOB (in SQL Server, VARBINARY) column.

```
[MaxLength(-1)]
public Byte[] Picture { get; set; }
```

Like in the previous example, the **-1** size will effectively be translated to MAX.

Ignoring a property, having Entity Framework never consider it for any operations, is as easy as setting a NotMappedAttribute on the property.

```
[NotMapped]
public String MySillyProperty { get; set; }
```

Fully ignoring a type, including any properties that might refer to it, is also possible by applying the NotMappedAttribute to its class instead.

```
[NotMapped]
public class MySillyType { }
```

Primary Keys

While database tables strictly don't require a primary key, Entity Framework requires it. Both single column as well as multi-column (composite) primary keys are supported. Marking a property, or properties, as the primary key is achieved by applying a KeyAttribute.

```
[Key]
public Int32 ProductId { get; set; }
```

If we have a composite primary key, we need to apply a ColumnAttribute as well. In it we need to give an explicit order by means of the Order property so that EF knows when an entity is loaded by the Find method, which argument refers to which property.

```
[Key]
[Column(Order = 1)]
public Int32 ColumnAId { get; set; }

[Key]
[Column(Order = 2)]
public Int32 ColumnBId { get; set; }
```

Primary keys can also be decorated with an attribute that tells Entity Framework how keys are to be generated (by the database or manually). This attribute is DatabaseGeneratedAttribute, and its values are explained in further detail in the chapter *Identifier Strategies*.

Navigation Properties

We typically don't need to include foreign keys in our entities; instead, we use references to the other entity, but we can have them as well. That's what the ForeignKeyAttribute is for.

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

[ForeignKey("Customer")]
public Int32 CustomerId { get; set; }
```

The argument to ForeignKeyAttribute is the name of the navigation property that the foreign key relates to.

Now suppose we have several relations from an entity to the other. For example, a customer might have two collections of projects: one for the current and other for the past projects. It could be represented in code as this.

```
public partial class Customer
{
    //the other endpoint will be the CurrentCustomer
    [InverseProperty("CurrentCustomer")]
    public virtual ICollection<Project> CurrentProjects { get; protected set; }

    //the other endpoint will be the PastCustomer
    [InverseProperty("PastCustomer")]
    public virtual ICollection<Project> PastProjects { get; protected set; }
}

public partial class Project
{
    public virtual Customer CurrentCustomer { get; set; }

    public virtual Customer PastCustomer { get; set; }
}
```

In that case, it is impossible for EF to figure out which property should be the endpoint for each of these collections, hence the need for the InversePropertyAttribute. When applied to a collection navigation property, it tells Entity Framework what is the name of the other endpoint's reference property that will point back to it.

Note: When configuring relationships, you only need to configure one endpoint.

Computed Columns

Entity Framework Code First does not support generating computed columns—columns whose values are not physically stored in a table but instead come from SQL formulas—automatically, but you can do it manually and have them mapped to your entities. A typical case is combining a first and a last name into a full name, which can be achieved on SQL Server very easily.
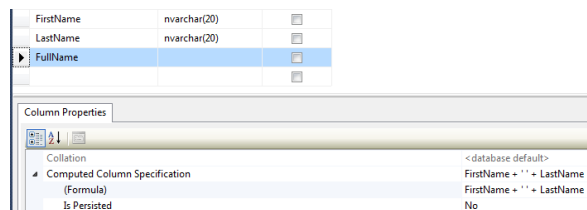
Figure 8: Computed columns

Another example of a column that is generated on the database is when we use a trigger for generating its values. You can map server-generated columns to an entity, but you must tell Entity Framework that this property is never to be inserted. For that we use the DatabaseGeneratedAttribute with the option DatabaseGeneratedOption.Computed.

```
public virtual String FirstName { get; set; }

public virtual String LastName { get; set; }

[DatabaseGenerated(DatabaseGeneratedOption.Computed)]
public virtual String FullName { get;  protected set; }
```

Since the property will never be set, we can have the setter as a protected method, and we mark it as DatabaseGeneratedOption.Computed to let Entity Framework know that it should never try to INSERT or UPDATE this column.

With this approach, you can query the FullName computed property with both LINQ to Objects as well as LINQ to Entities.

```
//this is executed by the database
var me = ctx.Resources.SingleOrDefault(x => x.FullName == "Ricardo Peres");

//this is executed by the process
var me = ctx.Resources.ToList().SingleOrDefault(x => x.FullName == "Ricardo Peres")
;
```

Complex Types

Complex types should be decorated with the ComplexTypeAttribute as in the following.

```
[ComplexType]
public class ContactInformation
{
```

```
  public String Email { get; set; }

  public String Phone { get; set; }
}
```

External Metadata

If you have a situation where you can't change an entity's code, the entity could have been generated automatically from a tool. See *Generating Code Automatically* for some examples of this. There's still something that you can do, provided the entities are generated as partial classes. The MetadataTypeAttribute allows us to have a class that contains the metadata—including, in this case, mapping attributes—for another class. Here's a quick example.

```
//the original Project class
public partial class Project
{
  //the following attributes will come from the ProjectMetadata class
  /*[Required]
  [MaxLength(50)]*/
  public String Name { get; set; }
}

//the new declaration
[MetadataType(typeof(ProjectMetadata))]
public partial class Project
{
  sealed class ProjectMetadata
  {
    [Required]
    [MaxLength(50)]
    public String Name { get; set; }
  }
}
```

This second class declaration is created in a new file and must reside in the same namespace as the original one. We don't need—in fact, we can't—to add any of its properties or methods; it will just serve as a placeholder for the MetadataTypeAttribute declaration. In the inner class ProjectMetadata, we will declare any properties for which we want to apply attributes.

> *Note: This is a good workaround for when code is generated automatically, provided all classes are generated as partial.*

Limitations

As of the current version of EFCF, there are some mapping concepts that cannot be achieved with attributes:

- Configuring cascading (see Cascading).
- Applying the Concrete Table Inheritance pattern (see Inheritance Strategies).

For these, we need to resort to code configuration, which is explained next.

Mapping by Code

Overview

Convenient as attribute mapping may be, it has some drawbacks:

- We need to add references in our domain model to the namespaces and assemblies where the attributes are defined (such as "domain pollution").
- We cannot change things dynamically; attributes are statically defined and cannot be overwritten.
- There isn't a centralized location where we can enforce our own conventions.

To help with these limitations, Entity Framework Code First offers an additional mapping API: code or fluent mapping. All functionality of the attribute-based mapping is present and more. Let's see how we implement the most common scenarios.

Fluent, or code, mapping is configured on an instance of DbModelBuilder and normally the place where we can access one is in the OnModelCreating method of the DbContext.

```csharp
public class ProjectsContext : DbContext
{
  protected override void OnModelCreating(DbModelBuilder modelBuilder)
  {
    //configuration goes here

    base.OnModelCreating(modelBuilder);
  }
}
```

This infrastructure method is called by Entity Framework when it is initializing a context, after it has automatically mapped whatever entity classes are referenced as DbSet<T> collections or referenced through them.

Schema

Here's how to configure the entity mappings by code.

```csharp
//set the table and schema
modelBuilder.Entity<Project>().ToTable("project", "dbo");

//ignoring an entity and all properties of its type
modelBuilder.Ignore<Project>();
```

This is an example of mapping individual properties. Notice how the API allows chaining multiple calls together by, in this case, setting simultaneously the column name, type, maximum length, and required flag. This is very useful and renders the code in a more readable way.

```
//ignore a property
modelBuilder.Entity<Project>().Ignore(x => x.SomeProperty);

//set a property's values (column name, type, length, nullability)
modelBuilder.Entity<Project>().Property(x => x.Name).HasColumnName("NAME")
.HasColumnType("VARCHAR").HasMaxLength(50).IsRequired();
```

Primary Keys

The primary key and the associated generation strategy are as follows.

```
//setting a property as the key
modelBuilder.Entity<Project>().HasKey(x => x.ProjectId);

//and the generation strategy
modelBuilder.Entity<Project>().Property(x => x.ProjectId)
.HasDatabaseGeneratedOption(DatabaseGeneratedOption.Identity);

//composite keys
modelBuilder.Entity<CompositeEntity>().HasKey(x => new { x.KeyColumnAId,
x.KeyColumnBId });
```

Navigation Properties

Navigation properties (references and collections) are as follows.

```
//a bidirectional many-to-one and its inverse with cascade
modelBuilder.Entity<Project>().HasRequired(x => x.Customer).WithMany(x => x.Project
s).WillCascadeOnDelete(true);

//a bidirectional one-to-many
modelBuilder.Entity<Customer>().HasMany(x => x.Projects)
.WithRequired(x => x.Customer);

//a bidirectional many-to-many
modelBuilder.Entity<Technology>().HasMany(x => x.Resources)
.WithMany(x => x.Technologies);

//a bidirectional one-to-one-or-zero with cascade
modelBuilder.Entity<Project>().HasOptional(x => x.Detail)
.WithRequired(x => x.Project).WillCascadeOnDelete(true);
//a bidirectional one-to-one (both sides required) with cascade
modelBuilder.Entity<Project>().HasRequired(x => x.Detail)
.WithRequiredPrincipal(x => x.Project).WillCascadeOnDelete(true);
```

```
//a bidirectional one-to-many with a foreign key property (CustomerId)
modelBuilder.Entity<Project>().HasRequired(x => x.Customer).WithMany(x => x.Project
s)
.HasForeignKey(x => x.CustomerId);

//a bidirectional one-to-many with a non-conventional foreign key column
modelBuilder.Entity<Project>().HasRequired(x => x.Customer).WithMany(x => x.Project
s)
.Map(x => x.MapKey("FK_Customer_Id"));
```

> *Note: When configuring relationships, you only need to configure one endpoint.*

Computed Columns

A simple column that is generated at the database by a formula, instead of being physically stored, can be done with the following.

```
modelBuilder.Entity<Resource>().Property(x => x.FullName)
.HasDatabaseGeneratedOption(DatabaseGeneratedOption.Computed);
```

Complex Types

Declaring a class as a complex type, no need to do it for individual properties, can be done with the following.

```
modelBuilder.ComplexType<ContactInformation>();

//a property in a complex type
modelBuilder.ComplexType<ContactInformation>().Property(x => x.Email).IsRequired()
.HasMaxLength(50);
```

As you can imagine, if we use fluent mapping extensively, the OnModelCreating method can get quite complex. Realizing this, EFCF offers the possibility to group entity configurations in their own class; this class must derive from EntityTypeConfiguration<T>, and here is an example of it.

```
modelBuilder.Configurations.Add(new CustomerConfiguration());

public class CustomerConfiguration : EntityTypeConfiguration<Customer>
{
  public CustomerConfiguration()
  {
    this.Table("FK_Customer_Id", "dbo");
    this.Property(x => x.Name).HasMaxLength(50).IsRequired();
  }
}
```

Identifier Strategies

Overview

Entity Framework requires that all entities have an identifier property that will map to the table's primary key. If this primary key is composite, multiple properties can be collectively designated as the identifier.

Identity

Although Entity Framework is not tied to any specific database engine, it is certainly true that out of the box it works better with SQL Server. Specifically, it knows how to work with IDENTITY columns, arguably the most common way in the SQL Server world to generate primary keys. Until recently, it was not supported by some major database engines such as Oracle.

By convention, whenever Entity Framework encounters a primary key of an integer type (Int32 or Int64), it will assume that it is an IDENTITY column. When generating the database, it will start with value 1 and use the increase step of 1 as well. It is not possible to change these parameters.

Tip: Although similar concepts exist in other database engines, Entity Framework can only use IDENTITY with SQL Server.

Manual

In the event that the identifier value is not automatically generated by the database, it must be manually set for each entity to be saved. If it is Int32 or Int64, and you want to use attributes for the mapping, then mark the identifier property with a DatabaseGeneratedAttribute and pass it the DatabaseGeneratedOption.None. This will avoid the built-in convention that will assume IDENTITY.

```
[Key]
[DatabaseGenerated(DatabaseGeneratedOption.None)]
public Int32 ProjectId { get; set; }
```

Use the following if you prefer fluent mapping.

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{

modelBuilder.Entity<Project>().HasKey(x => x.ProjectId).Property(x => x.ProjectId)
    .HasDatabaseGeneratedOption(DatabaseGeneratedOption.None);

    base.OnModelCreating(modelBuilder);
}
```

In this case, it is your responsibility to assign a valid identifier that doesn't already exist in the database. This is quite complex, mostly because of concurrent accesses and transactions; a popular alternative consists of using a Guid for the primary key column. You still have to initialize its value yourself, but the generation algorithm assures that there won't ever be two identical values.

```
public Project()
{
   //always set the id for every new instance of a Project
   this.ProjectId = Guid.NewGuid();
}

[Key]
[DatabaseGenerated(DatabaseGeneratedOption.None)]
public Guid ProjectId { get; set; }
```

> **Note: When using non-integral identifier properties, the default is to not have them generated by the database, so you can safely skip the DatabaseGeneratedAttribute.**

> **Note: Using Guids for primary keys also has the benefit that you can merge records from different databases into the same table; the records will never have conflicting keys.**

Inheritance Strategies

Consider the following class hierarchy.
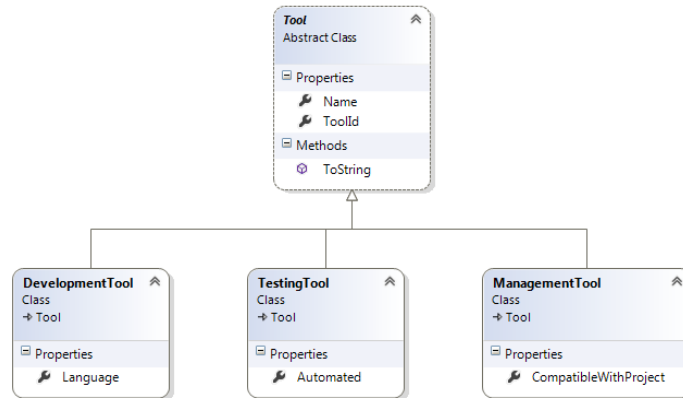
Figure 9: An inheritance model

In this example, we have an abstract concept, a Tool, and three concrete representations of it: a DevelopmentTool, a TestingTool, and a ManagementTool. Each Tool must be one of these types.

In object-oriented languages, we have class inheritance, which is something relational databases don't have. How can we store this in a relational database?

Martin Fowler, in his seminal work *Patterns of Enterprise Application Architecture*, described three patterns for persisting class hierarchies in relational databases:

1. Single Table Inheritance or Table Per Class Hierarchy: a single table is used to represent the entire hierarchy; it contains columns for all mapped properties of all classes. Many of these will be NULL because they will only exist for one particular class; one discriminating column will store a value that will tell Entity Framework what class a particular record will map to.
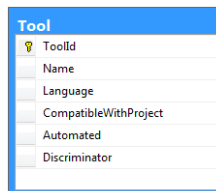
Figure 10: Single Table Inheritance data model

1. Class Table Inheritance or Table Per Class: a table will be used for the columns for all mapped base class properties, and additional tables will exist for all concrete classes; the additional tables will be linked by foreign keys to the base table.
2.
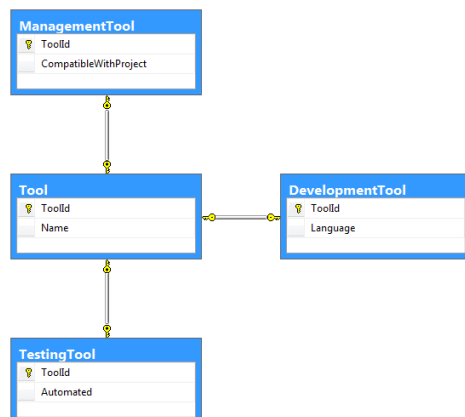
Figure 11: Class Table Inheritance data model

3. Concrete Table Inheritance or Table Per Concrete Class: one table for each concrete class, each with columns for all mapped properties, either specific or inherited by each class.
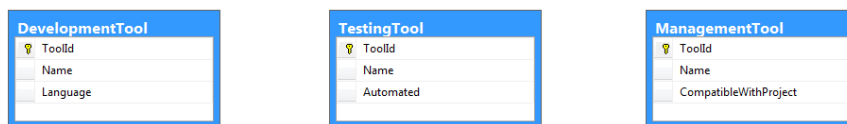
Figure 12: Concrete Table Inheritance data model

You can see a more detailed explanation of these patterns on Martin's web site at http://martinfowler.com/eaaCatalog/index.html. For now, I'll leave you with some thoughts:

- Single Table Inheritance, when it comes to querying from a base class, offers the fastest performance, because all information is contained in a single table. However, if you have lots of properties in all of the classes, it will be a difficult read, and you will have many nullable columns. In all of the concrete classes, all properties must be optional because they must allow null values. This is because different entities will be stored in the same class and not all share the same columns.
- Class Table Inheritance offers a good balance between table tidiness and performance. When querying a base class, a LEFT JOIN will be required to join each table from derived classes to the base class table. A record will exist in the base class table and in exactly one of the derived class tables.
- Concrete Table Inheritance for a query for a base class requires several UNIONs, one for each table of each derived class, because Entity Framework does not know beforehand in which table to look. This has the consequence that you cannot use IDENTITY as the identifier generation pattern or anyone that might generate identical values for any two tables. Entity Framework would be confused if it found two records with the same ID. Also, you will have the same columns, those from the base class, duplicated on all tables.

As for what Entity Framework is concerned, there really isn't any difference; classes are naturally polymorphic. See *0* for learning how we can perform queries on class hierarchies.

Here's how we can apply each of these patterns. First, here is an example for Single Table Inheritance.

```
public abstract class Tool
```

```
{
  public String Name { get; set; }

  public Int32 ToolId { get; set; }
}

public class DevelopmentTool : Tool
{
  //String is inherently nullable
  public String Language { get; set; }
}

public class ManagementTool : Tool
{
  //nullable Boolean
  public Boolean ? CompatibleWithProject { get; set; }
}

public class TestingTool : Tool
{
  //nullable Boolean
  public Boolean ? Automated { get; set; }
}
```

As you can see, there's nothing special you need to do. This is the default inheritance strategy.
One important thing, though: because all properties of each derived class will be stored in the
same table, all of them need to be nullable. It's easy to understand why. Each record in the
table will potentially correspond to any of the derived classes, and their specific properties only
have meaning to them, not to the others, so they may be undefined (NULL). In this example, I
have declared all properties in the derived classes as nullables.

Let's move on to the next pattern, Class Table Inheritance.

```
public abstract class Tool
{
  public String Name { get; set; }

  public Int32 ToolId { get; set; }
}

[Table("DevelopmentTool")]
public class DevelopmentTool : Tool
{
  public String Language { get; set; }
}

[Table("ManagementTool")]
public class ManagementTool : Tool
{
  public Boolean CompatibleWithProject { get; set; }
}
```

```csharp
[Table("TestingTool")]
public class TestingTool : Tool
{
  public Boolean Automated { get; set; }
}
```

Here, the difference is that we specify a table name for all of the derived entities. There is no need to do it for the base class, we'll just use the default. Properties of derived classes can be non-nullable, because they are stored in their own tables.

Finally, the Concrete Table Inheritance requires some more work.

```csharp
public abstract class Tool
{
  protected Tool()
  {
    //create a unique id for each instance
    this.ToolId = Guid.NewGuid();
  }

  public String Name { get; set; }

  //Guid instead of Int32
  public Guid ToolId { get; set; }
}

[Table("DevelopmentTool")]
public class DevelopmentTool : Tool
{
  public String Language { get; set; }
}

[Table("ManagementTool")]
public class ManagementTool : Tool
{
  public Boolean CompatibleWithProject { get; set; }
}

[Table("TestingTool")]
public class TestingTool : Tool
{
  public Boolean Automated { get; set; }
}

public class ProjectsContext : DbContext
{
  protected override void OnModelCreating(DbModelBuilder modelBuilder)
  {
    //map the inherited properties for each derived class
    modelBuilder.Entity<ManagementTool>().Map(m => m.MapInheritedProperties());
    modelBuilder.Entity<TestingTool>().Map(m => m.MapInheritedProperties());
    modelBuilder.Entity<DevelopmentTool>().Map(m => m.MapInheritedProperties());

    base.OnModelCreating(modelBuilder);
```

```
    }
}
```

Some notes:

- We cannot use the default, conventional, IDENTITY generation pattern for the primary key, because each table will have its own IDENTITY, and thus there would be records with the same primary key in each derived entity table. If we were going to query from the base class, for example, `ctx.Tools.Find(1)`, which record would EF choose? I chose to use a Guid as the primary key, which is a common decision, hence the constructor.
- We need to override the OnModelCreating method to complete the mapping.

Conventions

The current version of Entity Framework Code First at the time this book was written (5.0) comes along with a number of conventions. Conventions dictate how EFCF will configure some aspects of your model when they are not explicitly defined.

The full list of included conventions is comprised of the classes implementing IConvention that live in the System.Data.Entity.ModelConfiguration.Conventions namespace of the EntityFramework assembly. You can generally tell what they are supposed to do by looking at each class' description.

The most usual conventions are:

- EF will look for a connection string with the same name as the context (built in).
- All types exposed from a DbSet<T> collection in the DbContext-derived class with public setters and getters are mapped automatically (built in).
- All properties of all mapped types with a getter and a setter (any visibility) are mapped automatically, unless explicitly excluded (built in).
- All properties of nullable types are not required; those from non-nullable types (value types in .NET) are required.
- All virtual properties (references and collections) should be lazy loaded (built in).
- Single primary keys of integer types will use IDENTITY as the generation strategy (built in).
- If an entity is named with a singular noun, its corresponding table will have a pluralized name (PluralizingTableNameConvention).
- Primary key properties not explicitly set will be found automatically (IdKeyDiscoveryConvention).
- Associations to other entities are discovered automatically and the foreign key columns are built by composing the foreign entity name and its primary key (AssociationInverseDiscoveryConvention, NavigationPropertyNameForeignKeyDiscoveryConvention, PrimaryKeyNameForeignKeyDiscoveryConvention and TypeNameForeignKeyDiscoveryConvention).

- Child entities are deleted from the database whenever their parent is, if the relation is set to required (OneToManyCascadeDeleteConvention and ManyToManyCascadeDeleteConvention).

For now, there is no way to add our own custom conventions. If we want to disable a convention, just remove its class from the DbModelBuilder instance in the OnModelCreating override.

```csharp
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    //create tables with the same names as the entities, do not pluralize them
    modelBuilder.Conventions.Remove<PluralizingTableNameConvention>();

    base.OnModelCreating(modelBuilder);
}
```

We have already seen how to override the conventional table and column names.

```csharp
//change the physical table name
[Table("MY_PROJECT")]
public class Project
{
    //change the physical column name
    [Column("ID")]
    public Int32 ProjectId { get; set; }
}
```

For the connection string to use, EF lets us call a base constructor that receives a connection string.

```csharp
public class ProjectsContext : DbContext
{
    //use a different connection string
    public ProjectsContext(): base("Name=SomeConnectionString") { }
}
```

> **Note: The next version of Entity Framework, 6.0, will support custom conventions.**

Obtaining the Model Definition

If you need to import the model you generated using Code First to "classical" Entity Framework, it's just a matter of exporting an EDMX file and importing it on another project.

```csharp
var ctx = new  ProjectsContext();
XmlWriterSettings settings = new XmlWriterSettings { Indent = true };

using (XmlWriter writer = XmlWriter.Create("ProjectsContext.edmx", settings))
```

```
{
    EdmxWriter.WriteEdmx(ctx, writer);
}
```

If you open the produced file with Visual Studio, this is what you get.
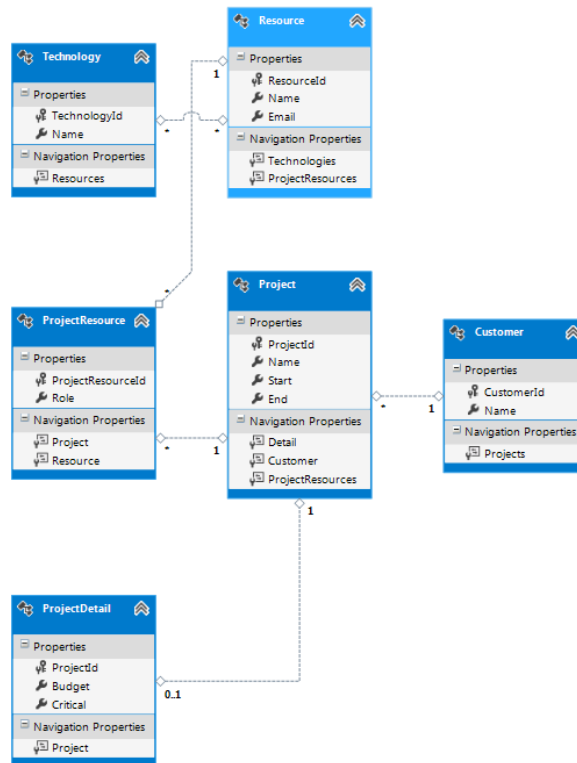
Figure 13: The Entity Data Model definition (EDMX)

This is a perfectly valid Entity Data Model definition, and you can import it into a project that uses "classic" Entity Framework and generate a class model from it.

Generating Code Automatically

One thing that is often requested is the ability to generate entity types that Entity Framework Code First can use straight away. This normally occurs when we have a large database with a big number of tables for which it would be hard to call classes manually. This is not exactly the purpose of Code First, but we have some options.

The first option is to start with "classic" Entity Framework. First, we add an item of type **ADO.NET Entity Data Model**.
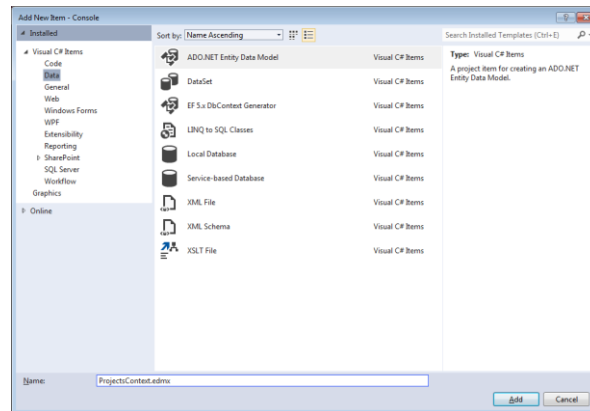
Figure 14: Add an Entity Data Model

We will want to generate entities from an existing database.
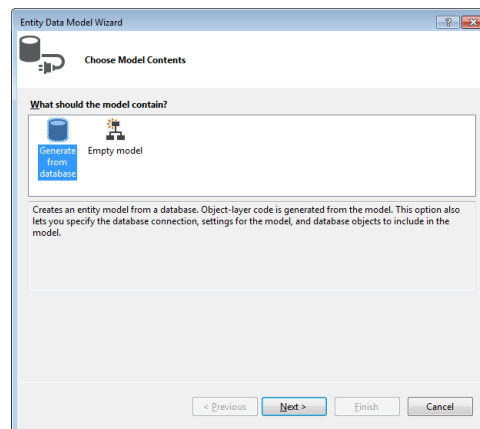
Figure 15: Generate model from database

Next, we have to create or select an existing connection string.
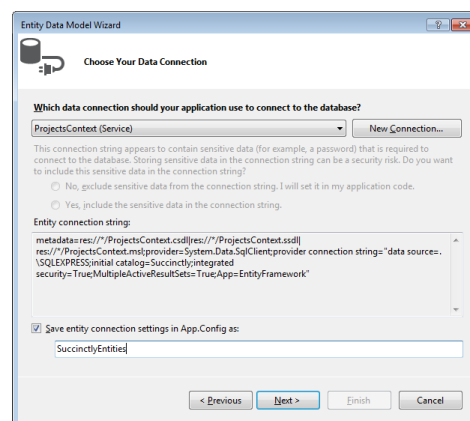
Figure 16: Set the connection string for generating the entities from the database

And choose the database objects (tables, views, stored procedures) that we want to map.
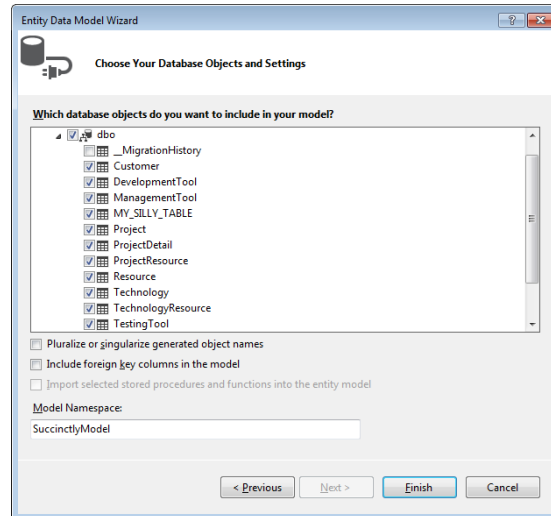
Figure 17: Choose the database objects to map

Finally, the model is generated.
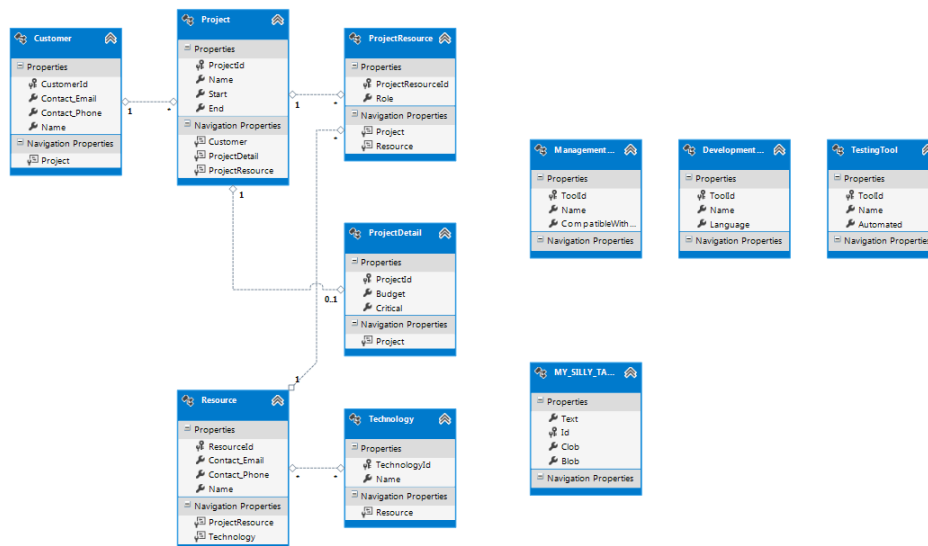
Figure 18: The generated Entity Data Model

💡 ***Tip: Inheritance is lost because EF has no way to know how we would want to
have it.***

As it is now, it won't produce a Code First context and entities, but instead regular "classic"
Entity Framework context and entities. In order to do the trick, we need to add a code
generation item, which we can do by right-clicking the design surface and selecting that option.
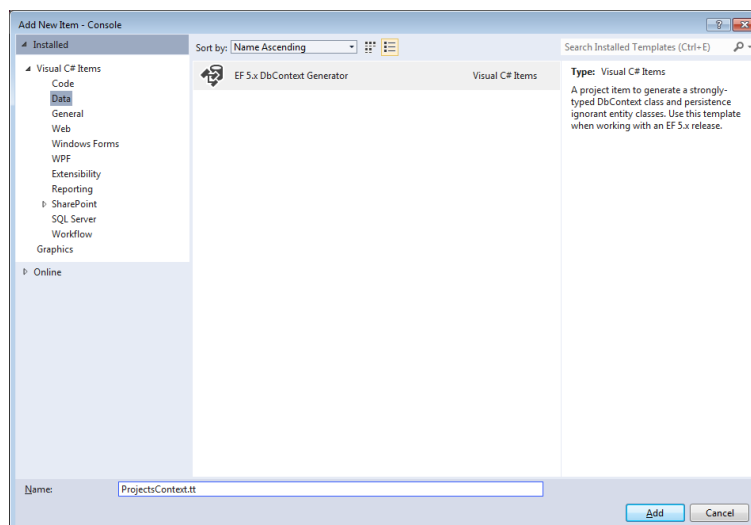
Figure 19: Adding a DbContext generator

We then select the **EF 5.x DbContext Generator** item, and that's it. As soon as you build your project, you will have both the context and the entities. Keep one thing in mind: the changes you make to the code will be lost if you update the model or have it regenerate the code!

Your other option is to use the **Entity Framework Power Tools**. This extension can be obtained from the **Extension Manager** of Visual Studio; at the time this book was written (August-September 2013), it is still in beta.
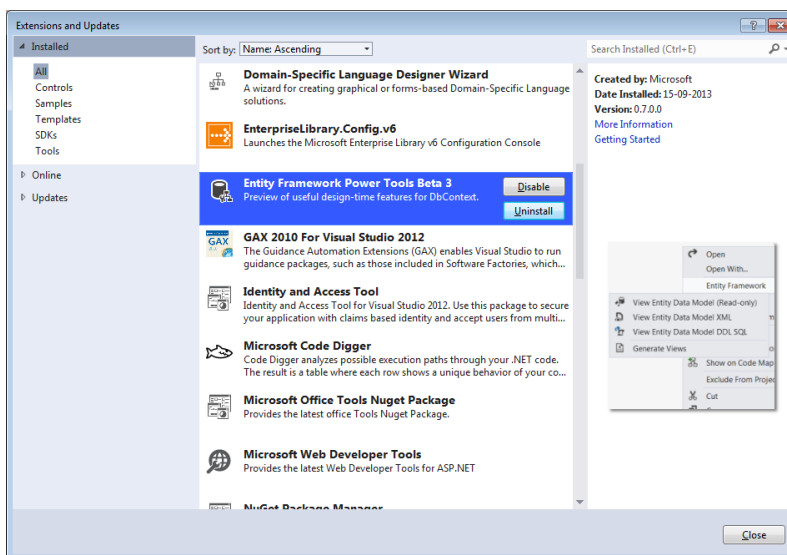
Figure 20: Entity Framework Power Tools extension

This extension adds a context menu to Visual Studio projects named **Entity Framework**. Inside of it there's a **Reverse Engineer Code First** option.
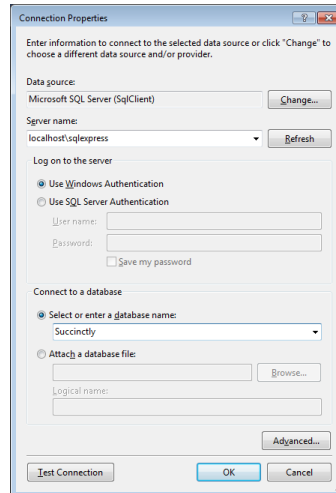
Figure 21: Reverse engineering a Code First model

As soon as you accept the connection string properties, Visual Studio will go to the database and will generate a context, its entities and associations for all the tables in the selected database, as well as its fluent mappings.
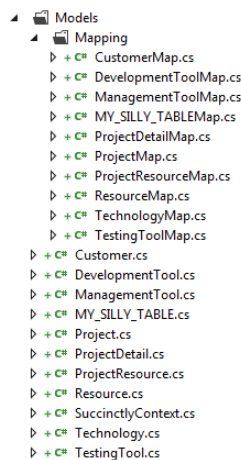
Figure 22: The generated Code First model

As you can see, this is way faster than with the "classic" approach. Some remarks on the process:

- This is a one-off generation: you cannot refresh the generated entities from the database, just replace them entirely, which means you will lose any changes you made to the code.
- The tool will generate mappings for all tables, there is no option to select just some of them.
- Inheritances are lost, which is understandable because they don't exist at the database level, only at the class level.

Tip: The Power Tools extension is still in beta, so you can expect some bugs to exist.

Chapter 3 Database

Configuring the Connection String

Like we saw in section *Contexts* of *Chapter 2 Chapter 2* Domain Model, the DbContext class has some constructors that take as parameters either the name of a connection string that should be present in the configuration file or a full connection string. If the public parameter-less constructor is used, Entity Framework by convention will look for a connection string with the same name of the context class in the configuration file. Let's see some examples.

```csharp
public class ProjectsContext : DbContext
{
  public ProjectsContext() { }

  public ProjectsContext(bool alternateConnection) :
  base(alternateConnection ? "Name=Succinctly" : "Name=ProjectsContext") { }

  public ProjectsContext(String nameOrConnectionString)
  : base(nameOrConnectionString) { }
}
```

If the first constructor is used, there must be an entry like the following in the configuration file.

```xml
<connectionStrings>
  <add name="ProjectsContext"
  connectionString="Data Source=.\SQLEXPRESS;Integrated Security=SSPI;
  Initial Catalog=Succinctly;MultipleActiveResultSets=true"
  providerName="System.Data.SqlClient"/>
</connectionStrings>
```

> *Tip: The actual connection string format and provider name depend on the database engine.*

Generating the Database

Explicit Creation

Code First, as its name implies, comes before the database. Nevertheless, we still need it and have to create it. EFCF offers a couple of ways to do just that.

We can check that a database specified by a connection string already exists. We have a method called Exists just for that purpose.

```
//check if the database identified by a named connection string exists
var existsByName = Database.Exists("Name=ProjectsContext");

//check if the database identified by a connection string exists
var existsByConnectionString = Database.Exists(@"Data
Source=.\SQLEXPRESS;Integrated Security=SSPI;Initial
Catalog=Succinctly;MultipleActiveResultSets=true");
```

If we decide that it should be created, we need to start a new context and ask it to do that for us by calling Create.

```
using (var ctx = new ProjectsContext())
{
  //create a database explicitly
  ctx.Database.Create();
}
```

It can also be done in a single step with CreateIfNotExists.

```
using (var ctx = new ProjectsContext())
{
  //will create the database if it doesn't already exist
  var wasCreated = ctx.Database.CreateIfNotExists();
}
```

> 💡 *Tip: The user specified by the connection string, which can even be the current Windows user, needs to have access right to create a database.*

> 💡 *Tip: If the database already exists, Create will fail, and CreateIfNotExists will return false.*

Database Initializers

Another way to have Entity Framework create the database for us is to use what is called a database initializer. This is a class that implements IDatabaseInitializer<T>, which can be associated with a context's class, and that will perform the initialization. Initialization is the process that creates the database and its tables. Entity Framework Code First comes with some implementations:

- CreateDatabaseIfNotExists<T>: will only create the database and tables if they don't already exist; this is the default initializer.
- DropCreateDatabaseAlways<TContext>: will drop and create the database always, beware!
- DropCreateDatabaseIfModelChanges<TContext>: if the model stored in the database does not match the current context's, the database will be dropped and recreated (see below).
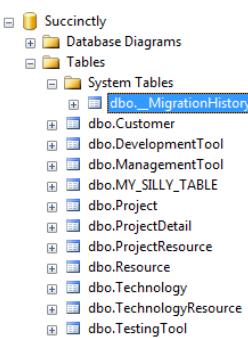
Entity Framework Code First Succinctly

- **MigrateDatabaseToLatestVersion<TContext, TMigrationsConfiguration>**: will run a custom migration for updating the current database to the current model (more on this later).

The first two initializers are self-explanatory. The last two need some explanation.

Each context has a backing model, which represents all the mapped entities, their properties and associations, and how they are tied to the database. When Entity Framework creates the database, it stores this model in a system table called __MigrationHistory.

Figure 23: The __MigrationHistory table

This table only has three columns, whose purpose is the following:

- The MigrationId column says when the database was created.
- The Model column will contain a GZipped representation of the Entity Data Model definition XML (EDMX). Whenever something changes in the model, like adding a new property or a new entity, changing a property's attributes, this definition will also change.
- The ProductVersion describes the version of Entity Framework and the .NET framework in use.

The **DropCreateDatabaseIfModelChanges<TContext>** database initializer loads the contents of this table and matches it against the model of the context. If it sees changes, probably because the class model has changed, it drops the database and recreates it.

> 💡 ***Tip: Entity Framework will never detect changes in the database, only in code.***

The **MigrateDatabaseToLatestVersion<TContext, TMigrationsConfiguration>** is quite powerful, and because it is also more complex, it will be covered in the *Migrations* section.

It is certainly possible to roll out our own initializer. For that, we have two options:

- Inherit from an existing initializer: the problem is that the included initializers have a well-defined and unchangeable behavior, and we can't really change; however, by overriding the Seed method, we can add initial data to the database whenever the initializer is run, but that's about it.

- Create our own initializer: for that we need to implement IDatabaseInitializer<T>, since there is no base class to inherit from; it's up to us to do all the work, and it might be cumbersome.

To have initial data created with the database, just override the Seed method of your initializer of choice.

```csharp
public class CreateProjectsDatabaseWithInitialData :
CreateDatabaseIfNotExists<ProjectsContext>
{
  protected override void Seed(ProjectsContext context)
  {
    var developmentTool = new DevelopmentTool() { Name = "Visual Studio 2012",
      Language = "C#" };
    var managementTool = new ManagementTool() { Name = "Project 2013",
      CompatibleWithProject = true };
    var testingTool = new TestingTool() { Name = "Selenium", Automated = true };

    context.Tools.Add(developmentTool);
    context.Tools.Add(managementTool);
    context.Tools.Add(testingTool);

    //don't forget to save changes, as this doesn't happen automatically
    context.SaveChanges();

    base.Seed(context);
  }
}
```

Note: The Seed method is not part of the IDatabaseInitializer<T> contract; if we write our own custom initializer, we need to define and call it explicitly.

While an initializer can be run explicitly on its own, just create an instance of it and call its InitializeDatabase with a context as its argument, it is often useful to associate it with a context's type, so that whenever a context is built, it will inherit the initializer without further work. This is achieved through the SetInitializer method.

```csharp
//run the initializer explicitly
new CreateDatabaseIfNotExists<ProjectsContext>().InitializeDatabase(ctx);

//set an automatic initializer for all instances of ProjectsContext
Database.SetInitializer(new CreateDatabaseIfNotExists<ProjectsContext>);

//run the initializer configured for this context type, even if it has already run
ctx.Database.Initialize(true);
```

A good place to put this initialization code is the static constructor for the context. This way, it will run a single time when the first instance of the context is created, and we are sure that it won't be forgotten.

```
public class ProjectsContext : DbContext
{
  //the static constructor runs a single time with the first instance of a class
  static ProjectsContext()
  {
    Database.SetInitializer(new CreateDatabaseIfNotExists<ProjectsContext>);
  }
}
```

This can also be done through configuration (App.config or Web.config). This has the advantage that the database initialization strategy can be changed without requiring a recompilation of the code.

```
<configuration>
  <configSections>
    <section name="entityFramework"
    type="System.Data.Entity.Internal.ConfigFile.EntityFrameworkSection,
EntityFramework" requirePermission="false" />
  </configSections>
  <entityFramework>
    <defaultConnectionFactory type="System.Data.Entity.Infrastructure.
SqlConnectionFactory, EntityFramework" />
    <contexts>
      <context type="Succinctly.Model.ProjectsContext, Succinctly.Model">
        <databaseInitializer type="System.Data.Entity.
CreateDatabaseIfNotExists`1[[Succinctly.Model.ProjectsContext, Succinctly.Model]],
EntityFramework">
          <!-- only required if the initializer constructor takes parameters -->
          <!-- must follow the order they are specified in the constructor -->
          <!--parameters>
            <parameter value="MyConstructorParameter" type="System.Int32"/>
          </parameters-->
        </databaseInitializer>
      </context>
    </contexts>
  </entityFramework>
</configuration>
```

💡 ***Tip: Notice the somewhat weird syntax that must be used when specifying a generic class' name.***

Finally, you can disable any initializer on your context. If you do so, you are on your own. You may have to create the database and all its objects by yourself using Entity Framework's functionality. The way to disable initializers is by explicitly setting it to null for a given context.

```
//disable initializers for all instances of ProjectsContext
Database.SetInitializer<ProjectsContext>(null);
```

It can also be achieved by configuration.

```
<entityFramework>
```

```
  <contexts>
    <context type="Succinctly.Model.ProjectsContext, Succinctly.Model"
disableDatabaseInitialization="true" />
  </contexts>
</entityFramework>
```

💡 ***Tip: Don't forget, the default initializer is
CreateDatabaseIfNotExists<TContext>.***

Generating Scripts

If for whatever reason you need to look at and possibly tweak the SQL script that is used to
create the database, you can export it into a file; for that, you need to access the ObjectContext
of the context.

```
var octx = (ctx as IObjectContextAdapter).ObjectContext;

File.WriteAllText("ProjectsContext.sql", octx.CreateDatabaseScript());
```

💡 ***Tip: The generated SQL will be specific to the database engine whose
provider we are using.***

Migrations

We all know that schema and data change over time, be it the addition of another column or a
modification in a base record. Entity Framework Code First offers a code-based approach for
dealing with these kinds of scenarios. It's called migrations, which are actually implemented as
database initializers and have a base class of MigrateDatabaseToLatestVersion<TContext,
TMigrationsConfiguration>.
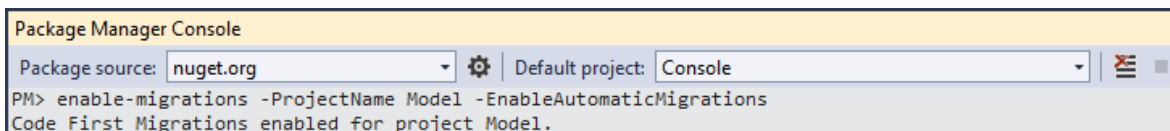
A migration can be of one of two types:

- Automatic: whenever a change is detected in the model—a property is added or
  removed, new entities are mapped—Entity Framework Code First migrations will update
  the database automatically to reflect these changes.
- Versioned or named: explicitly changes the schema—adding columns, indexes or
  constraints to existing tables, for example—and store a checkpoint, so that it can be
  rolled back.

As you can see, this is a big improvement over the other database initializers. The database
doesn't need to be dropped when the model changes.

Automatic Migrations

An automatic migration is called by executing the **Enable-Migrations** PowerShell command on the NuGet console.

> *Note: You only need to pass the ProjectName parameter if your model is defined in an assembly that is not the startup one.*

This command will create a configuration class called **Configuration** inheriting from DbMigrationsConfiguration to the **Migrations** folder.
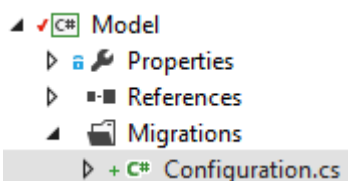
Figure 24: Automatic migration configuration class

The system table __**MigrationHistory** will be updated to reflect the fact that an automatic migration was created.

Figure 25: The __MigrationHistory table after an automatic migration was created

Now we can tell Entity Framework to start using this metadata for the database initialization.

```
Database.SetInitializer(
new MigrateDatabaseToLatestVersion<ProjectsContext, Configuration>());
```

To see this in action, we can add a new property to one of our model classes, anything will do.

```
public class Project
{
  public int? X { get; set; }
}
```

> *Tip: Do not add required properties to an existing model, because the migration will fail if the table already has records.*

When we instantiate the context and run any query, such as `ctx.Projects.ToList()`, we won't notice it, but the database was modified behind the scene.

Figure 26: The modified table after the new column was added

Did you see the new X column? It was created as the result of an automatic migration. The __MigrationHistory table will contain an evidence of it.

Figure 27: The __MigrationHistory table after an automatic migration

> **Note: Remember that the Model column contains a compressed version of the Entity Definition Model XML.**

What if we wanted to remove this new property? We would have two options:

- Remove it from the model and let automatic migrations take care of dropping its column from the database as well.
- Reverting to the initial state of the model.

If we tried option number one, we would get an AutomaticDataLossException.
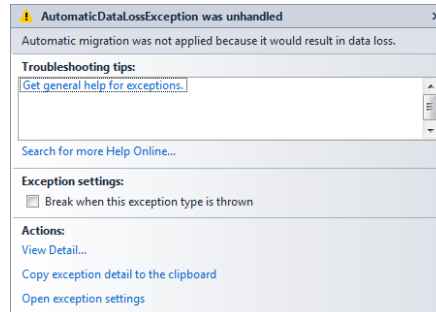
Figure 28: Exception thrown by automatic migrations

The second option is achieved by running the Update-Database command, but it would yield the same result.

```
PM> update-database -ProjectName Model -TargetMigration $InitialDatabase
Specify the '-Verbose' flag to view the SQL statements being applied to the target database.
Reverting migrations: [201309211516207_AutomaticMigration, 201309211453190_InitialCreate].
Reverting automatic migration: 201309211516207_AutomaticMigration.
Automatic migration was not applied because it would result in data loss.
```

Figure 29: Exception thrown by update-database

> **Tip: The special keyword $InitialDatabase allows us to go back to the initial version of the model, as it was stored in the __MigrationHistory table.**

This is actually the automatic migrations' way of warning us that we were probably doing something wrong. In this case, we were dropping a column on which we might have valuable data.

If that is not the case, we can tell it to forget about these warnings by going to the **Configuration** class generated when we enabled the automatic migrations and setting the AutomaticMigrationDataLossAllowed to true in its constructor, next to AutomaticMigrationsEnabled.

```
public Configuration()
{
  this.AutomaticMigrationsEnabled = true;
  this.AutomaticMigrationDataLossAllowed = true;
}
```

> 💡 ***Tip: Use AutomaticMigrationDataLossAllowed with care, or you may lose important information.***

If we run the same code again, and then look at the database, we will see that effectively the new column has been dropped.

	ProjectId	Name	Start	End	Customer_CustomerId
1	1	Big Project	2013-09-21 00:00:00.000	NULL	1
2	2	Small Project	2013-09-14 00:00:00.000	2013-09-20 00:00:00.000	2

Figure 30: The modified table after the new column was dropped

And it shows up in __MigrationHistory.

	MigrationId	Model	ProductVersion
1	201309211453190_InitialCreate	0x1F8B0800000000000400ED1D5D6FDCB8F1BD40FFC3629F...	5.0.0.net45
2	201309211516207_AutomaticMigration	0x1F8B0800000000000400ED1D5D6FDCB8F1BD40FFC3629F...	5.0.0.net45
3	201309211538380_AutomaticMigration	0x1F8B0800000000000400ED1D5D6FDCB8F1BD40FFC3629F...	5.0.0.net45

Figure 31: The __MigrationHistory table after the revert

Versioned Migrations

The other type of migration offers more fine-grained control, but involves more work. We start by creating a migration with a name that describes our purpose.

```
PM> add-migration -ProjectName Model -Name AddProjectStatus
Scaffolding migration 'AddProjectStatus'.
The Designer Code for this migration file includes a snapshot of your current Code First model. This snapshot is used to calculate the changes to your model when you scaffold the next migration. If you make additional changes to your model that you want to include in this migration, then you can re-scaffold it by running 'Add-Migration 201309211556555_AddProjectStatus' again.
```

Figure 32: Adding a versioned migration

We now have a new class in the **Migrations** folder whose name reflects the name passed to Add-Migration and the timestamp of its creation which it inherits from DbMigration.
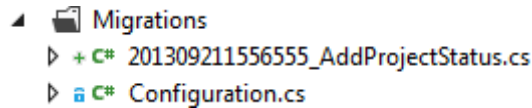
Figure 33: The versioned migration class

This new class is empty except for two method declarations, which are overridden from the base class.

- Up: will specify the changes that will be applied to the database when this migration is run.
- Down: will contain the reverse of the changes declared in the Up method, for the case when this migration is rolled back.

Note: This class is not specific to any Entity Framework Code First context; it only cares about the database.

Let's see a simple example.

```csharp
public partial class AddProjectStatus : DbMigration
{
  public override void Up()
  {
    this.AddColumn("dbo.Project", "Status", x =>
      x.Int(nullable: false, defaultValue: 0));
    this.CreateIndex("dbo.Project", "Status");
  }

  public override void Down()
  {
    this.DropIndex("dbo.Project", "Status");
    this.DropColumn("dbo.Project", "Status");
  }
}
```

As you can see, on the Up method we are doing two things:

4. Adding a new column, Status, of type INT, not-nullable, with a default value of 1, to the **dbo.Project** table.
5. Creating an index over this new column.

All operations are reversed in the Down method.

The DbMigration class contains helper methods for most typical database operations, but in case we need something different, we can always use the Sql method.

```csharp
this.Sql("-- some SQL command");
```

Now that we have this migration, we might as well execute it.

Entity Framework Code First Succinctly

```
PM> Update-Database -ProjectName Model -TargetMigration AddProjectStatus
Specify the '-Verbose' flag to view the SQL statements being applied to the target database.
Applying code-based migrations: [201309211556555_AddProjectStatus].
Applying code-based migration: 201309211556555_AddProjectStatus.
Running Seed method.
```

Figure 34: Executing a named migration

The output mentions a Seed method. This method is defined in the Configuration class in the **Migrations** folder and it is currently empty. On it, you can add any initial data that is required by your business logic.

```
protected override void Seed(ProjectsContext context)
{
  context.Projects.AddOrUpdate(p => p.Name, new Project { Name = "Big Project",
Customer = new Customer { CustomerId = 1 }, Start = DateTime.Now });
}
```

> *Tip: Beware, the Seed method might be called several times, one for each migrations is run, so you must be careful to not insert duplicate data; that's what the AddOrUpdate method is there for.*

Once again, we can see that the __MigrationHistory table was updated.

	MigrationId	Model	ProductVersion
1	201309211453190_InitialCreate	0x1F8B0800000000000400ED1D5D6FDCB8F1BD40FFC3629F...	5.0.0.net45
2	201309211516207_AutomaticMigration	0x1F8B0800000000000400ED1D5D6FDCB8F1BD40FFC3629F...	5.0.0.net45
3	201309211538380_AutomaticMigration	0x1F8B0800000000000400ED1D5D6FDCB8F1BD40FFC3629F...	5.0.0.net45
4	201309211556555_AddProjectStatus	0x1F8B0800000000000400ED1D5D6FDCB8F1BD40FFC3629F...	5.0.0.net45

Figure 35: The __MigrationHistory table after the named migration

Of course, we can always go back to the previous state, or the state identified by any named migration has been run, by using Update-Database.

```
PM> Update-Database -ProjectName Model -TargetMigration $InitialDatabase
Specify the '-Verbose' flag to view the SQL statements being applied to the target database.
Reverting migrations: [201309211556555_AddProjectStatus, 201309211538380_AutomaticMigration, 201309211516207_AutomaticMigration, 201309211453190_InitialCreate].
Reverting code-based migration: 201309211556555_AddProjectStatus.
Reverting automatic migration: 201309211538380_AutomaticMigration.
Reverting automatic migration: 201309211516207_AutomaticMigration.
Reverting automatic migration: 201309211453190_InitialCreate.
```

Figure 36: Returning to the initial database version

If you want to revert to a named migration, just pass its name as a parameter to Update-Database.

```
PM> update-database -ProjectName Model -TargetMigration SomeOldVersion
```

Figure 37: Reverting to a named version

At any time, you can see what migrations have been executed in the database by invoking the Get-Migrations command.

```
PM> get-migrations -ProjectName Model
Retrieving migrations that have been applied to the target database.
201309211858284_InitialCreate
```

Figure 38: Listing all applied migrations

Chapter 4 Getting Data from the Database

Overview

As you might expect, Entity Framework offers a number of APIs to get data from the database into objects. These are designed to cover different scenarios; we will see them one by one.

By Id

All data access layers really must support loading by primary key, and certainly EFCF does so. For that, we have the Find method.

```
//retrieving a record by a single primary key consisting of an integer
var project = ctx.Projects.Find(1);
```

This method can take any number of parameters; it supports entities having composite primary keys. The following example shows just that.

```
//retrieving a record by a composite primary key consisting of two integers
var project = ctx.SomeEntities.Find(1, 2);
```

Find will return null if no matching record is found; no exception will be thrown since this is a perfectly valid result.

> **Tip: The order and type of the parameters should be the same as the keys defined in the mapping for this entity.**

LINQ

Since its introduction with .NET 3.5, Language Integrated Querying (LINQ) has become the *de facto* standard for querying data of any kind, so it's no surprise that EF has LINQ support. It will probably be your API of choice for most scenarios. It is strongly typed, meaning you can tell at compile time that some things are not right, refactor friendly, and its syntax is easy to understand. Let's see some examples.

A LINQ query is built from the entity collection properties of DbSet<T> type exposed by the context and it consists of an IQueryable<T> implementation.

Entity Framework Code First Succinctly

> 💡 **Tip: Don't forget that LINQ is about querying. You won't find any way to change data here.**

If you don't include a terminal operator, such as ToList, ToArray, ToDictionary, Any, Count, LongCount, Single, SingleOrDefault, First, FirstOrDefault, Last, LastOrDefault, a query is not actually executed. You can pick up this query and start adding restrictions such as paging or sorting.

```
//create a base query
var projectsQuery = from p in ctx.Projects select p;

//add sorting
var projectsSortedByDateQuery = projectsQuery.OrderBy(x => x.Start);

//execute and get the sorted results
var projectsSortedByDateResults = projectsSortedByDateQuery.ToList();

//add paging and ordering (required for paging)
var projectsWithPagingQuery = projectsQuery.OrderBy(x => x.Start).Take(5).Skip(0);

//execute and get the first 5 results
var projectsWithPagingResults = projectsWithPagingQuery.ToList();

//add a restriction
var projectsStartingAWeekAgoQuery = projectsQuery.Where(x => x.Start
>= EntityFunctions.AddDays(DateTime.Today, -7));

//execute and get the projects that started a week ago
var projectsStartingAWeekAgoResults = projectsStartingAWeekAgoQuery.ToList();
```

You will get at most a single result.

```
//retrieving at most a single record with a simple filter
var project = ctx.Projects.Where(x => x.ProjectId == 1).SingleOrDefault();
```

Restricting by several properties is just as easy.

```
//retrieving multiple record with two filters
var projects = ctx.Projects.Where(x => x.Name.Contains("Something") && x.Start >=
DateTime.Today.AddDays(-7)).ToList();
```

Or having one of two conditions matched.

```
//or
var resourcesKnowingVBOrCS = ctx.Technologies.Where(t => t.Name == "VB.NET"
|| t.Name == "C#").SelectMany(x => x.Resources).Select(x => x.Name).ToList();
```

Count results.

```
//count
var numberOfClosedProjects = ctx.Projects
.Where(x => x.End != null && x.End < DateTime.Now).Count();
```

Check record existence.

```
//check existence
var existsProjectBySomeCustomer = ctx.Projects
.Any(x => x.Customer.Name == "Some Customer");
```

Perform a projection, that is, only get some parts of an entity.

```
//get only the name of the resource and the name of the associated project
var resourcesXprojects = ctx.Projects.SelectMany(x => x.ProjectResources)
.Select(x => new { Resource = x.Resource.Name, Project = x.Project.Name }).ToList()
;
```

Do aggregations.

```
//average project duration
var averageProjectDuration = ctx.Projects.Where(x => x.End != null)
.Average(x => EntityFunctions.DiffDays(x.Start, x.End));

//sum of project durations by customer
var sumProjectDurationsByCustomer = ctx.Projects.Where(x => x.End != null)
.Select(x => new { Customer = x.Customer.Name, Days = EntityFunctions
.DiffDays(x.Start, x.End) }).GroupBy(x => x.Customer)
.Select(x => new { Customer = x.Key, Sum = x.Sum(y => y.Days) }).ToList();
```

Get distinct values.

```
//distinct roles performed by a resource
var roles = ctx.Resources.SelectMany(x => x.ProjectResources)
.Where(x => x.Resource.Name == "Ricardo Peres").Select(x => x.Role).Distinct()
.ToList();
```

You can also group on a property.

```
//grouping and projecting
var resourcesGroupedByProjectRole = ctx.Projects.SelectMany(x => x.ProjectResources
)
.Select(x => new { Role = x.Role, Resource = x.Resource.Name }).GroupBy(x => x.Role
)
.Select(x => new { Role = x.Key, Resources = x }).ToList();

//grouping and counting
var projectsByCustomer = ctx.Projects.GroupBy(x => x.Customer)
.Select(x => new { Customer = x.Key.Name, Count = x.Count() }).ToList();

//top 10 customers having more projects in descending order
```

```
var top10CustomersWithMoreProjects = ctx.Projects.GroupBy(x => x.Customer.Name)
.Select(x => new { x.Key, Count = x.Count() }).OrderByDescending(x => x.Count)
.Take(10).ToList();
```

Or use the results of a subquery.

```
//subquery
var usersKnowingATechnology = (from r in ctx.Resources where r.Technologies.Any(x =
> (from t in ctx.Technologies where t.Name == "ASP.NET" select t).Contains(x)) sele
ct r).ToList();
```

Finally, check for one of a set of values.

```
//contains
var customersToFind = new String[] { "Some Customer", "Another Customer" };
var projectsOfCustomers = ctx.Projects
.Where(x => customersToFind.Contains(x.Customer.Name)).ToList();
```

> **Note: In case you are wondering, all literals present in LINQ queries –
> strings, numbers, dates, etc. – will be turned into parameters for proper
> execution plan reusing.**

You might have noticed that whenever we had date operations, we made use of some
EntityFunctions methods such as EntityFunctions.DiffDays. This class has some extension
methods for operations that LINQ does not offer. These can be grouped as:

- Date and time: add time intervals, truncate a date or time, calculate the time interval
  between two dates.
- String: get the left or right part of a string, reverse it, convert it to ASCII or UNICODE.
- Statistics: variance and variance based on the whole population.

There's also a similar class, SqlFunctions, that offers extension methods for invoking SQL
Server specific functions, namely:

- Mathematical: trigonometric, logarithmic, power, conversion to and from radians, random
  numbers.
- String: get ASCII and UNICODE code from a string, calculate the difference between
  two strings, the position of a character in a string, inserts a string into another, gets the
  SOUNDEX value for a string.
- Checksums: calculate checksums for columns.
- Date and time: calculate time intervals between dates, add time interval.
- System: get the database date and time, the current user, host name.
- Data types: checks if a column can be converted to another data type, does
  conversions.

As flexible as LINQ is, there are some limitations. It's not easy to build queries dynamically, for
example, when the filters or desired sorting property is not known at compile time. Let's look at
the other APIs to see how we can deal with these situations.

> 💡 **_Tip: Do not query over calculated columns, only mapped ones, because otherwise you will get an error, since Entity Framework knows nothing about these._**

Entity SQL

Entity SQL (or ESQL) is Entity Framework's query language. It is very similar to SQL, but it has some great advantages.

- It is very similar to SQL, so we can reuse some of our knowledge of it.
- It is database-independent, which means it works the same regardless of the actual database we are targeting.
- It is object, not set, oriented, so in some ways it is similar to what we do in code.
- It knows about entity relations, so we don't need to specify them.
- Unlike LINQ, queries can be created dynamically, at runtime.

> 💡 **_Tip: Entity SQL only supports querying, not updating._**

Entity SQL commands are strings, which means we can do the usual string manipulation operations, and will only know if the syntax is correct when we actually execute it.

While Entity Framework Code First can certainly use Entity SQL, it can't do it directly: neither DbContext or any of the other Code First classes expose methods for working with it. We first need to get its underlying ObjectContext.

```
//get the ObjectContext from the DbContext
ObjectContext octx = (ctx as IObjectContextAdapter).ObjectContext;
```

And the actual action occurs on the CreateQuery method.

```
//all values from the Projects collection
var allProjects = octx.CreateQuery<Resource>("SELECT VALUE p FROM Projects AS p")
.ToList();
```

Parameters are explicitly named with the @ prefix before the name and must be given a value.

```
//simple filter
var usersInProject = octx.CreateQuery<Resource>(
"SELECT VALUE pr.Resource FROM ProjectResources AS pr WHERE pr.Project.Name = @name
", new ObjectParameter("name", "Big Project")).ToList();
```

Using a subquery for the following.

```
//contains
var usersKnowingATechnology = octx.CreateQuery<Resource>(
"SELECT VALUE r FROM Resources AS r WHERE EXISTS (SELECT VALUE t FROM Technologies
AS t WHERE t.Name = @name AND r IN t.Resources)",
new ObjectParameter("name", "ASP.NET")).ToList();
```

Use the following for paging.

```
//paging
var pagedResources = octx.CreateQuery<Resource>(
"SELECT VALUE r FROM Resources AS r ORDER BY r.Name SKIP 5 LIMIT(5)").ToList();

//paging with parameters
var pagedResourcesWithParameters = octx.CreateQuery<Resource>(
"SELECT VALUE r FROM Resources AS r ORDER BY r.Name SKIP @skip LIMIT(@limit)",
new ObjectParameter("skip", 5), new ObjectParameter("limit", 5)).ToList();

//single first record ordered descending
var lastProject = octx.CreateQuery<Project>(
"SELECT VALUE TOP(1) p FROM Projects AS p ORDER BY p.Start DESC").SingleOrDefault()
;
```

Use the following for ranges.

```
//between with parameters
var projectsStartingInADateInterval = octx.CreateQuery<Project>(
"SELECT VALUE p FROM Projects AS P WHERE p.Start BETWEEN @start AND @end",
new ObjectParameter("start", DateTime.Today.AddDays(14)),
new ObjectParameter("end", DateTime.Today.AddDays(-7))).ToList();

//in with inline values
var projectsStartingInSetOfDates = octx.CreateQuery<Project>(
"SELECT VALUE p FROM Projects AS P WHERE p.Start IN MULTISET(DATETIME
'2013-12-25 0:0:0', DATETIME '2013-12-31 0:0:0')").ToList();
```

Use the following for count records.

```
//count records
var numberOfClosedProjects = octx.CreateQuery<Int32>(
"SELECT VALUE COUNT(p.ProjectId) FROM Projects AS p WHERE p.[End] IS NOT NULL AND p
.[End] < @now", new ObjectParameter("now", DateTime.Now)).Single();
```

Use the following for projections.

```
//projection with date difference
var projectNameAndDuration = octx.CreateQuery<Object>(
"SELECT p.Name AS Name, DIFFDAYS(p.Start, p.[End]) FROM Projects AS p WHERE p.[End]

IS NOT NULL").ToList();

//projection with count
```

```
var customersAndProjectCount = octx.CreateQuery<Object>(
"SELECT p.Customer.Name, COUNT(p.Name) FROM Projects AS p GROUP BY p.Customer")
.ToList();

//projection with case
var customersAndProjectCountIndicator = octx.CreateQuery<Object>(
"SELECT p.Customer.Name, CASE WHEN COUNT(p.Name) > 10 THEN 'Lots' ELSE 'Few' END AS
Amount FROM Projects AS p GROUP BY p.Customer").ToList();
```

💡 ***Tip: Property names with the same name as reserved keywords must be escaped inside [ ].***

💡 ***Tip: When performing projections with more than one property, the template argument type of CreateQuery must be Object.***

When projecting, we lose strong typing, so we must directly access the returned IDataRecord instances.

```
if (customersAndProjectCountIndicator.Any() == true)
{
  var r = customersAndProjectCountIndicator.OfType<IDataRecord>().First();
  var nameIndex = r.GetOrdinal("Name");
  var nameValue = r.GetString(nameIndex);
}
```

As for some built-in functions.

```
//max number of days
var maxDurationDays = octx.CreateQuery<Int32?>(
"SELECT VALUE MAX(DIFFDAYS(p.Start, p.[End])) FROM Projects AS p WHERE p.[End] IS N
OT NULL").SingleOrDefault();

//string matching (LIKE)
var technologiesContainingDotNet = octx.CreateQuery<String>(
"SELECT VALUE t.Name FROM Technologies AS T WHERE CONTAINS(t.Name, '.NET')")
.ToList();
```

SQL

Try as we might, the truth is, it's impossible to escape from using SQL. This may be because performance is typically better or because some query is difficult or even impossible to express using any of the other APIs, but that's just the way it is. Entity Framework Code First has full support for SQL, including:

- Getting entities and values.
- Executing INSERTs, UPDATEs and DELETEs.
- Calling functions and stored procedures.

Entity Framework Code First Succinctly

It does have all the disadvantages you might expect:

- Not strongly typed.
- No compile-time checking.
- If you use database-specific functions and you target a new database, you have to rewrite your SQL.
- You have to know the right syntax for even simple things, such as paging or limiting the number of records to return.

The first case I'm demonstrating is how to execute a SELECT and convert the result into an entity. For that we shall use the SqlQuery method of the DbSet<T>.

```
//simple select
var projectFromSQL = ctx.Projects.SqlQuery(
"SELECT * FROM Project WHERE Name = @p0", "Big Project").SingleOrDefault();
```

> *Tip: Notice how we pass parameters directly after the SQL; each must be named @p0, @p1, and so on.*

If we wanted to retrieve an entity from a table-valued function, we would use.

```
//table-valued function
var projectFromFunction = ctx.Projects.SqlQuery(
"SELECT * FROM dbo.GetProjectById @p0", 1).SingleOrDefault();
```

Where the GetProjectById function might be.

```
CREATE FUNCTION dbo.GetProjectById
(
        @ProjectID INT
)
RETURNS TABLE
AS
RETURN
(
        SELECT *
        FROM Project
        WHERE ProjectId = @ProjectID
)
GO
```

> *Tip: Methods that return entities from SQL cannot return complex types, only scalar and enumerated types. This is a known problem with Entity Framework and there is an open issue for it at http://entityframework.codeplex.com/workitem/118.*

> *Tip: Don't forget that if you want to return entities, your SQL must return columns that match the properties of these entities, as specified in its mapping.*

Although the SqlQuery was primarily design to work with entities, we can also use it to retrieve scalars.

```
//current date and time
var now = ctx.Database.SqlQuery<DateTime>("SELECT GETDATE()").Single();
```

If we want to execute arbitrary SQL commands (UPDATE, DELETE, INSERT), we will need the ExecuteSqlCommand like in the following example.

```
//update records
var updateCount = ctx.Database.ExecuteSqlCommand(
"UPDATE ProjectDetail SET Budget = Budget * 1.1 WHERE ProjectId = {0}", 1);
```

Finally, for really special situations, we can always resort to the underlying DbConnection.

```
//create the connection in a using block so that it is disposed at the end
using (var cmd = ctx.Database.Connection.CreateCommand())
{
  if (ctx.Database.Connection.State == ConnectionState.Closed)
  {
    ctx.Database.Connection.Open();
  }

  cmd.CommandText = "Some weird SQL command";

  //the number of affected records, if the query returns it
  var result = cmd.ExecuteNonQuery();

  //or a single scalar value
  //var result = cmd.ExecuteScalar();

  //or even a data reader
  var result = cmd.ExecuteReader();

  ctx.Database.Connection.Close();
}
```

Lazy, Explicit and Eager Loading

Lazy Loading

By default, all references (one-to-one, many-to-one) and collections (one-to-many, many-to-many) are lazy loaded, which means that Entity Framework won't actually try to load its values until someone tries to access them. For instance, consider this query where we load a Project by its id.

```
//load a project by id
var p = ctx.Projects.Find(1);
```

This query will produce this SQL.

```
SELECT TOP (2)
[Extent1].[ProjectId] AS [ProjectId],
[Extent1].[Description_Description] AS [Description_Description],
[Extent1].[Name] AS [Name],
[Extent1].[Start] AS [Start],
[Extent1].[End] AS [End],
[Extent1].[Customer_CustomerId] AS [Customer_CustomerId]
FROM [dbo].[Project] AS [Extent1]
WHERE [Extent1].[ProjectId] = @p0
-- p0 (dbtype=Int32, size=0, direction=Input) = 1
```

> 📋 *Note: You might have noticed the TOP(2) clause: this is merely for assuring that a single record is selected, as the Find method expects; if that is not the case, an exception will be thrown.*

As you can see, the only table that is touched is the **Project** one. However, when we access the **Customer** property.

```
//access the customer
var c = p.Customer;
```

The **Customer** reference property will then be loaded, For that, EF will issue another query.

```
SELECT
[Extent2].[CustomerId] AS [CustomerId],
[Extent2].[Contact_Email] AS [Contact_Email],
[Extent2].[Contact_Phone] AS [Contact_Phone],
[Extent2].[Name] AS [Name]
FROM  [dbo].[Project] AS [Extent1]
INNER JOIN [dbo].[Customer] AS [Extent2] ON [Extent1].[Customer_CustomerId] = [Extent2].[CustomerId]
WHERE [Extent1].[ProjectId] = @EntityKeyValue1
-- EntityKeyValue1 (dbtype=Int32, size=0, direction=Input) = 1
```

Or if we go the other way and first load a customer.

```
//load a customer by id
var customer = ctx.Customers.Find(1);
```

The following SQL is executed.

```sql
SELECT TOP (2)
[Extent1].[CustomerId] AS [CustomerId],
[Extent1].[Contact_Email] AS [Contact_Email],
[Extent1].[Contact_Phone] AS [Contact_Phone],
[Extent1].[Name] AS [Name]
FROM [dbo].[Customer] AS [Extent1]
WHERE [Extent1].[CustomerId] = @p0
-- p0 (dbtype=Int32, size=0, direction=Input) = 1
```

Do note that only the Customer data is retrieved, and this is consistent with the previous query. Now let's access the **Projects** collection.

```
//load all projects
var projects = customer.Projects;
```

This SQL is sent to the server.

```sql
SELECT
[Extent1].[ProjectId] AS [ProjectId],
[Extent1].[Description_Description] AS [Description_Description],
[Extent1].[Name] AS [Name],
[Extent1].[Start] AS [Start],
[Extent1].[End] AS [End],
[Extent1].[Customer_CustomerId] AS [Customer_CustomerId]
FROM [dbo].[Project] AS [Extent1]
WHERE [Extent1].[Customer_CustomerId] = @EntityKeyValue1
-- EntityKeyValue1 (dbtype=Int32, size=0, direction=Input) = 1
```

The Projects of this Customer are all loaded into memory. Entity Framework takes care of generating the appropriate SQL for us, opening and closing connections behind our back and instancing the entities from the returned resultsets. Keep in mind that navigation properties are only loaded from the database the first time they are accessed, so it is likely that you will notice a delay on the first access, but after that they are always returned from memory.

> *Note: Never mind the actual SQL details; they are generated from a generic algorithm and even if they are not exactly what you'd expect. They will work!*

This is possible because when EF returns entities from a query, it doesn't return instances of the exact declared class but from a derived, special class. This class, called a proxy class, is generated automatically by EF.

```
//load a project by id
var p = ctx.Projects.Find(1);
```
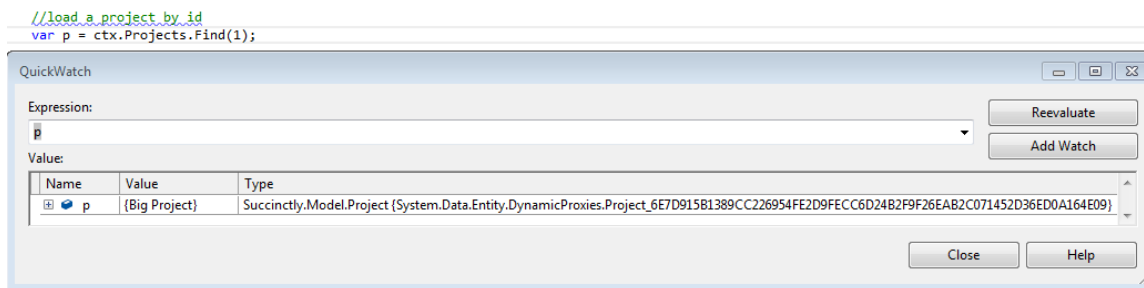

Figure 39: A proxy to an entity

See the funny class name starting with **System.Data.Entity.DynamicProxies.Project_** in the inspector? That is the class that is generated automatically. This class has a reference to the context from which it came from, and has overrides for all navigation properties so that when we try to access them by code, it will only then load the associated entities. For this to happen, you must simultaneously assure that:

- The class is not sealed.
- The class is not private or internal.
- All navigation properties (references and collections) meant to be lazy loaded are virtual.
- All navigation properties are implemented as auto properties, with no backing field and no custom logic.

Seems easy, don't you think? If any of this conditions is not verified, then lazy loading cannot be used. As an example, if we would pick the Project entity and change its Customer property so that it is not virtual, then loading a Project would always have the consequence of having a null value for its Customer property.

> ***Tip: Navigation properties that haven't been explicitly loaded always have value null when lazy loading is disabled.***

There are properties in the DbContext that allow configuring lazy loading: Database.Configuration.LazyLoadingEnabled and ProxyCreationEnabled. Please note that even though there are two properties, both must be simultaneously true to have lazy loading working, which they are by default.

To see if a reference property has been loaded, we use code such as the following.

```
//load a project
var project = ctx.Projects.Find(1);

//see if the Customer property is loaded
var customerLoaded = ctx.Entry(project).Reference(x => x.Customer).IsLoaded;
```

If you try to load a lazy navigation property after you have disposed of its originating context, you will get an ObjectDisposedException, because the entity relies on an active context for getting data from the database.

Explicit Loading

Say you disable lazy loading.

```
//disable lazy loading
ctx.Configuration.LazyLoadingEnabled = false;
```

If lazy loading is turned off, either globally or for a certain property, we can still force a navigation property to load explicitly.

```
//explicitly load the Customer property
ctx.Entry(project).Reference(x => x.Customer).Load();
```

Don't forget that this is only necessary if lazy loading is disabled. The same also applies to collections.

```
//see if the ProjectResources collection is loaded
var resourcesLoaded = ctx.Entry(project).Collection(x => x.ProjectResources)
.IsLoaded;

if (resourcesLoaded == false)
{
  //explicitly load the ProjectResources collection
  ctx.Entry(project).Collection(x => x.ProjectResources).Load();
}
```

Another interesting case is where you want to load just a part of some collection by filtering out the entities that do not match a given condition, or even count its members without actually loading them. It is possible to do it with EF, and for that you would issue queries like this.

```
//count an entity's collection entities without loading them
var countDevelopersInProject = ctx.Entry(project).Collection(x => x.ProjectResource
s).Query().Where(x => x.Role == Role.Developer).Count();

//filter an entity's collection without loading it
var developersInProject = ctx.Entry(project).Collection(x => x.ProjectResources)
.Query().Where(x => x.Role == Role.Developer).ToList();
```

The difference between lazy and eager loading is that with lazy loading, you don't need to do anything explicit, you just access the navigation properties without even thinking about it; whereas with explicit loading you have to perform some action.

Eager Loading

Even if lazy and explicit loading is good for most occasions, you only load the data you need, when you need it, there may be cases where you want all data from the main entity as well as associated entities to be loaded at the same time. This will most likely be because of one of two reasons:

- You are certain that you are going to have to access some of the navigation properties, and for performance reasons, you load them beforehand (for example, you need to go through all order's details).
- The entity's (and its associated entities') lifecycle will probably outlive the context from which it was obtained from (for example, you are going to store the entity in some cache), so it won't have access to it and thus lazy loading will not be possible.

Enter eager loading. What eager loading means is, when issuing a query, you explicit declare the expansion paths that Entity Framework will bring along with the root entities. EF will then generate a different SQL expression than it would normally would with JOINs for all the required associations.

For example, the following query brings along a Customer and all of its Projects and it introduces the Include method.

```
//explicitly eager load the Customer for each project
var projectsAndTheirCustomers = ctx.Projects.Include(x => x.Customer).ToList();
```

For the record, this will produce the following SQL statement.

```
SELECT
[Extent1].[ProjectId] AS [ProjectId],
[Extent1].[Name] AS [Name],
[Extent1].[Start] AS [Start],
[Extent1].[End] AS [End],
[Extent2].[CustomerId] AS [CustomerId],
[Extent2].[Contact_Email] AS [Contact_Email],
[Extent2].[Contact_Phone] AS [Contact_Phone],
[Extent2].[Name] AS [Name1]
FROM   [dbo].[Project] AS [Extent1]
INNER JOIN [dbo].[Customer] AS [Extent2] ON
[Extent1].[Customer_CustomerId] = [Extent2].[CustomerId]
```

The Include method can also take a String as its parameter, which must be the name of a navigation property (a reference or a collection).

```
//explicitly eager load the Customer for each project
var projectsAndTheirCustomers = ctx.Projects.Include("Customer").ToList();
```

Multiple paths can be specified.

```
//two independent include paths
var resourcesProjectResourcesAndTechnologies = ctx.Resources
.Include(x => x.ProjectResources).Include(x => x.Technologies).ToList();
```

In this case, the SQL will look like.

```
SELECT
[UnionAll1].[ResourceId] AS [C1],
[UnionAll1].[ResourceId1] AS [C2],
[UnionAll1].[ResourceId2] AS [C3],
```

```sql
[UnionAll1].[Contact_Email] AS [C4],
[UnionAll1].[Contact_Phone] AS [C5],
[UnionAll1].[Name] AS [C6],
[UnionAll1].[C1] AS [C7],
[UnionAll1].[ProjectResourceId] AS [C8],
[UnionAll1].[ProjectResourceId1] AS [C9],
[UnionAll1].[Role] AS [C10],
[UnionAll1].[Project_ProjectId] AS [C11],
[UnionAll1].[Resource_ResourceId] AS [C12],
[UnionAll1].[C2] AS [C13],
[UnionAll1].[C3] AS [C14]
FROM  (SELECT
        CASE WHEN ([Extent2].[ProjectResourceId] IS NULL) THEN CAST(NULL AS int)
 ELSE 1 END AS [C1],
        [Extent1].[ResourceId] AS [ResourceId],
        [Extent1].[ResourceId] AS [ResourceId1],
        [Extent1].[ResourceId] AS [ResourceId2],
        [Extent1].[Contact_Email] AS [Contact_Email],
        [Extent1].[Contact_Phone] AS [Contact_Phone],
        [Extent1].[Name] AS [Name],
        [Extent2].[ProjectResourceId] AS [ProjectResourceId],
        [Extent2].[ProjectResourceId] AS [ProjectResourceId1],
        [Extent2].[Role] AS [Role],
        [Extent2].[Project_ProjectId] AS [Project_ProjectId],
        [Extent2].[Resource_ResourceId] AS [Resource_ResourceId],
        CAST(NULL AS int) AS [C2],
        CAST(NULL AS varchar(1)) AS [C3]
        FROM  [dbo].[Resource] AS [Extent1]
        LEFT OUTER JOIN [dbo].[ProjectResource] AS [Extent2] ON [Extent1].[ResourceId]
= [Extent2].[Resource_ResourceId]
UNION ALL
        SELECT
        2 AS [C1],
        [Extent3].[ResourceId] AS [ResourceId],
        [Extent3].[ResourceId] AS [ResourceId1],
        [Extent3].[ResourceId] AS [ResourceId2],
        [Extent3].[Contact_Email] AS [Contact_Email],
        [Extent3].[Contact_Phone] AS [Contact_Phone],
        [Extent3].[Name] AS [Name],
        CAST(NULL AS int) AS [C2],
        CAST(NULL AS int) AS [C3],
        CAST(NULL AS int) AS [C4],
        CAST(NULL AS int) AS [C5],
        CAST(NULL AS int) AS [C6],
        [Join2].[TechnologyId] AS [TechnologyId],
        [Join2].[Name] AS [Name1]
        FROM  [dbo].[Resource] AS [Extent3]
        INNER JOIN  (SELECT [Extent4].[Resource_ResourceId] AS [Resource_ResourceId],
[Extent5].[TechnologyId] AS [TechnologyId], [Extent5].[Name] AS [Name]
                FROM  [dbo].[TechnologyResource] AS [Extent4]
                INNER JOIN [dbo].[Technology] AS [Extent5] ON [Extent5].[TechnologyId]
= [Extent4].[Technology_TechnologyId] ) AS [Join2] ON [Extent3].[ResourceId] =
[Join2].[Resource_ResourceId]) AS [UnionAll1]
ORDER BY [UnionAll1].[ResourceId1] ASC, [UnionAll1].[C1] ASC
```

A final example with multilevel inclusion:

```
//multilevel include paths
var resourcesProjectResourcesCustomers = ctx.Resources.Include(x => x.ProjectResour
ces.Select(y => y.Project.Customer)).ToList();
```

And the generated SQL will look like this.

```
SELECT
[Project1].[ResourceId] AS [ResourceId],
[Project1].[Contact_Email] AS [Contact_Email],
[Project1].[Contact_Phone] AS [Contact_Phone],
[Project1].[Name] AS [Name],
[Project1].[C1] AS [C1],
[Project1].[ProjectResourceId] AS [ProjectResourceId],
[Project1].[Role] AS [Role],
[Project1].[ProjectId] AS [ProjectId],
[Project1].[Name1] AS [Name1],
[Project1].[Start] AS [Start],
[Project1].[End] AS [End],
[Project1].[CustomerId] AS [CustomerId],
[Project1].[Contact_Email1] AS [Contact_Email1],
[Project1].[Contact_Phone1] AS [Contact_Phone1],
[Project1].[Name2] AS [Name2],
[Project1].[Resource_ResourceId] AS [Resource_ResourceId]
FROM ( SELECT
        [Extent1].[ResourceId] AS [ResourceId],
        [Extent1].[Contact_Email] AS [Contact_Email],
        [Extent1].[Contact_Phone] AS [Contact_Phone],
        [Extent1].[Name] AS [Name],
        [Join2].[ProjectResourceId] AS [ProjectResourceId],
        [Join2].[Role] AS [Role],
        [Join2].[Resource_ResourceId] AS [Resource_ResourceId],
        [Join2].[ProjectId] AS [ProjectId],
        [Join2].[Name1] AS [Name1],
        [Join2].[Start] AS [Start],
        [Join2].[End] AS [End],
        [Join2].[CustomerId] AS [CustomerId],
        [Join2].[Contact_Email] AS [Contact_Email1],
        [Join2].[Contact_Phone] AS [Contact_Phone1],
        [Join2].[Name2] AS [Name2],
        CASE WHEN ([Join2].[ProjectResourceId] IS NULL) THEN CAST(NULL AS int) ELSE 1
END AS [C1]
        FROM  [dbo].[Resource] AS [Extent1]
        LEFT OUTER JOIN  (SELECT [Extent2].[ProjectResourceId] AS [ProjectResourceId],
[Extent2].[Role] AS [Role], [Extent2].[Resource_ResourceId] AS [Resource_ResourceId],
[Extent3].[ProjectId] AS [ProjectId], [Extent3].[Name] AS [Name1], [Extent3].[Start] AS
[Start], [Extent3].[End] AS [End], [Extent4].[CustomerId] AS [CustomerId],
[Extent4].[Contact_Email] AS [Contact_Email], [Extent4].[Contact_Phone] AS
[Contact_Phone], [Extent4].[Name] AS [Name2]
                FROM   [dbo].[ProjectResource] AS [Extent2]
                INNER JOIN [dbo].[Project] AS [Extent3] ON [Extent2].[Project_ProjectId]
= [Extent3].[ProjectId]
                INNER JOIN [dbo].[Customer] AS [Extent4] ON
[Extent3].[Customer_CustomerId] = [Extent4].[CustomerId] ) AS [Join2] ON
[Extent1].[ResourceId] = [Join2].[Resource_ResourceId]
)  AS [Project1]
ORDER BY [Project1].[ResourceId] ASC, [Project1].[C1] ASC
```

As you can imagine, EF goes into a lot of work to JOIN all data that it needs to fetch at the same time, hence the quite complicated SQL.

Local Data

Entities known by an Entity Framework context – either loaded from it or marked for deletion or for insertion – are stored in what is called local or first level cache. Martin Fowler calls it Identity Map, and you can read more about the concept at http://martinfowler.com/eaaCatalog/identityMap.html. Basically, the context keeps track of all these entities, so that it doesn't need to materialize them whenever a query that returns their associated records is executed. It is possible to access this cache by means of the Local property of the DbSet<T>:

```
//all local Projects
var projectsAlreadyLoaded = ctx.Projects.Local;

//filtered local Projects
var projectsAlreadyLoadedBelongingToACustomer = ctx.Projects.Local.Where(x =>
x.Customer.Name == "Some Customer");
```

Neither of these queries goes to the database, they are all executed in memory. The Local property is actually an instance of ObservableCollection<T>, which means we have an event that notifies us whenever new items are added or removed from the local cache, CollectionChanged.

```
ctx.Projects.Local.CollectionChanged += (sender, e) =>
{
  if (e.Action == NotifyCollectionChangedAction.Add)
  {
    //an entity was added to the local cache
  }
  else if (e.Action == NotifyCollectionChangedAction.Remove)
  {
    //an entity was removed from the local cache
  }
};

//discard all known projects (stop tracking their changes)
ctx.Projects.Local.Clear();
```

It is possible to know all entities that are present in the local cache, and to see their state, as seen by the context. It's the ChangeTracker's responsibility to keep track of all these entities.

```
//get the projects in local cache that have been modified
var modifiedProjects = ctx.ChangeTracker.Entries<Project>()
.Where(x => x.State == EntityState.Modified).Select(x => x.Entity);
```

As you can guess, it's considerably faster to get an entity from the local cache than it is to load it from the database. With that thought on our mind, we can write a method such as the next one, for transparently returning a local entity or fetching it with SQL.

```
//retrieve from cache or the database
public static IQueryable<T> LocalOrDatabase<T>(this DbContext context,
Expression<Func<T, Boolean>> expression) where T : class
{
    var localResults = context.Set<T>().Local.Where(expression.Compile());

  if (localResults.Any() == true)
  {
    return (localResults.AsQueryable());
  }

  return (context.Set<T>().Where(expression));
}
```

Chapter 5 Writing Data to the Database

Saving, Updating, and Deleting Entities

Saving Entities

Because EF works with POCOs, creating a new entity is just a matter of instantiating it with the **new** operator. If we want it to eventually get to the database, we need to attach it to an existing context.

```
var developmentTool = new DevelopmentTool() { Name = "Visual Studio 2012",
Language = "C#" };

ctx.Tools.Add(developmentTool);
```

New entities must be added to the DbSet<T> property of the same type, which is also your gateway for querying.

> **Tip: We can only add entities when we have exposed the entity's type, or a base of it, in our context, as a _DbSet<T>_ property.**

However, this new entity is not immediately sent to the database. The EF context implements the Unit of Work pattern, a term coined by Martin Fowler, for which you can read more here: http://martinfowler.com/eaaCatalog/unitOfWork.html. In a nutshell, this pattern states that the Unit of Work container will keep internally a list of items in need of persistence (new, modified, or deleted entities) and will save them all in an atomic manner, taking care of any eventual dependencies between them. The moment when these entities are persisted in Entity Framework Code First happens when we call the DbContext's SaveChanges method.

```
var affectedRecords = ctx.SaveChanges();
```

At this moment, all the pending changes are sent to the database. Entity Framework employs a first level (or local) cache, which is where all the "dirty" entities – like those added to the context – sit waiting for the time to persist. The SaveChanges method returns the number of records that were successfully inserted, and will throw an exception if some error occurred in the process. In that case, all changes are rolled back, and you really should take this scenario into consideration.

Updating Entities

As for updates, Entity Framework tracks changes to loaded entities automatically. For each entity, it "knows" what their initial values were, and if they differ from the current ones, the entity is considered "dirty." A sample code follows.

```
//load some entity
var tool = ctx.Tools.FirstOrDefault();

//change something
t.Name += "_changed";

//send changes
var affectedRecords = ctx.SaveChanges(); //1
```

As you see, no separate Update method is necessary, since all type of changes (inserts, updates, and deletes) are detected automatically and performed by the SaveChanges method. The SaveChanges method still needs to be called, and it will return the combined count of all inserted and updated entities. If some sort of integrity constraint is violated, then an exception will be thrown, and this needs to be dealt with appropriately.

Upserting Entities

Sometimes you may want to either insert a record or to update it, depending on whether it already exists or not; this is sometimes called UPSERT. Entity Framework Code First offers an interesting method that, from a property of an entity, checks its existence and acts accordingly. This method is AddOrUpdate, and here is a quick example.

```
ctx.Customers.AddOrUpdate(x => x.Name, new Customer { Name = "Big Customer" });
```

Because there's a chance that you will be inserting a new record, you must supply all of its required properties, otherwise an error will occur.

Deleting Entities

When you have a reference to a loaded entity, you can mark it as deleted, so that when changes are persisted, EF will delete the corresponding record. Deleting an entity in EF consists of removing it from the DbSet<T> collection.

```
//load some entity
var tool = ctx.Tools.FirstOrDefault();

//remove the entity
ctx.Tools.Remove(tool);

//send changes
var affectedRecords = ctx.SaveChanges(); //1
```

Of course, SaveChanges still needs to be called to make the changes permanent. If any integrity constraint is violated, an exception will be thrown.

> 📝 **Note: Entity Framework will apply all the pending changes – inserts, updates, and deletes – in an appropriate order, including entities that depend on other entities.**

Inspecting the Tracked Entities

We talked about the local cache, and you may have asked yourself where this cache is and what can be done with it.

You access the local cache entry for an entity with the Entry method. This returns an instance of DbEntityEntry, which contains lots of useful information, such as the current state of the entity – as seen by the context – the initial and current values, and so on.

```
//load some entity
var tool = ctx.Tools.FirstOrDefault();

//get the cache entry
var entry = ctx.Entry(tool);

//get the entity state
var state = entry.State; //EntityState.Unchanged

//get the original value of the Name property
var originalName = entry.OriginalValues["Name"] as String; //Visual Studio 2012

//change something
t.Name += "_changed";

//get the current state
state = entry.State; //EntityState.Modified

//get the current value of the Name property
var currentName = entry.CurrentValues["Name"] as String; //Visual Studio
2012_changed
```

If you want to inspect all the entries currently being tracked, there is the ChangeTracker property.

```
//get all the added entities of type Project
var addedProjects = ctx.ChangeTracker.Entries()
.Where(x => x.State == EntityState.Added).OfType<Project>();
```

Cascading Deletes

Two related tables can be created in the database in such a way that when one record of the parent table is deleted, all corresponding records in the child table are also deleted. This is called cascading deletes.
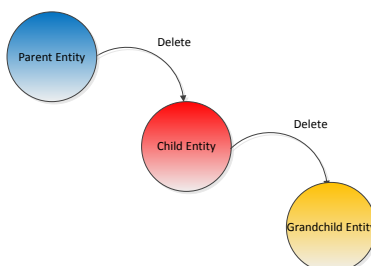

Figure 40: Cascading deletes

This is useful for automatically keeping the database integrity. If the database wouldn't do this for us, we would have to do it manually, otherwise we would end up with orphan records. This is only useful for parent-child or master-detail relationships where one endpoint cannot exist without the other; not all relationships should be created this way, for example, when the parent endpoint is optional, we typically won't cascade. Think of a customer-project relationship: it doesn't make sense to have projects without a customer. On the other hand, it does make sense to have a bug without an assigned developer.

When Entity Framework creates the database, it will create the appropriate cascading constraints depending on the mapping.
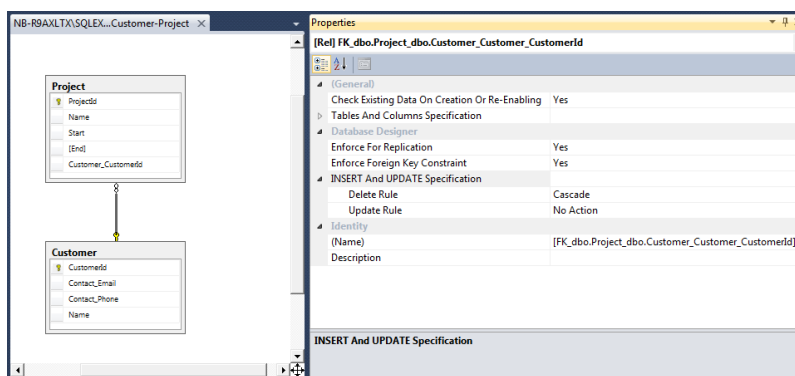

Figure 41: A cascade delete

As of now, EF applies a convention for that, but that can be overridden by fluent mapping.

Relationship	Default Cascade
One-to-One	No.
One-to-many	Only if the one endpoint is required.

Relationship	Default Cascade
Many-to-one	No.
Many-to-many	No.

We configure the cascading option in fluent configuration like this.

```
//when deleting a Customer, delete all of its Projects
modelBuilder.Entity<Project>().HasRequired(x => x.Customer)
.WillCascadeOnDelete(true);

//when deleting a ProjectResource, do not delete the Project
modelBuilder.Entity<ProjectResource>().HasRequired(x => x.Project)
.WithMany(x => x.ProjectResources).WillCascadeOnDelete(false);

//when deleting a Project, delete its ProjectResources
modelBuilder.Entity<Project>().HasMany(x => x.ProjectResources)
.WithRequired(x => x.Project).WillCascadeOnDelete(false);
```

Tip: You can have multiple levels of cascades, just make sure you don't have circular references.

Note: Cascade deletes occur at the database level; Entity Framework does not issue any SQL for that purpose.

Refreshing Entities

When a record is loaded by EF as the result of a query, an entity is created and is placed in local cache. When a new query is executed that returns records that are associated with an entity already in local cache, no new entity is created; instead, the one from the cache is returned. This has a sometimes undesirable side effect: even if something changed in the entities' record, the local entity is not updated. This is an optimization that Entity Framework does, but sometimes it can lead to unexpected results.

If we want to make sure we have the latest data, we need to force an explicit refresh.

```
var project = ctx.Projects.Find(1);

//time passes…

ctx.Entry(project).Reload();
```

Concurrency Control

Overview

Optimistic concurrency control is a method for working with databases that assumes that multiple transactions can complete without affecting each other; no locking is required. When committing a record, each transaction will check to see if the record has been modified in the database, and will fail. This is very useful for dealing with multiple accesses to data in the context of web applications.

There are two ways for dealing with the situation where data has been changed:

- First one wins: the second transaction will detect that data has been changed, and will throw an exception.
- Last one wins: while it detects that data has changed, the second transaction chooses to overwrite it.

Entity Framework supports both of these methods.

First One Wins

We have an entity instance obtained from the database, we change it, and we tell the EFCF context to persist it. Because of optimistic concurrency, the SaveChanges method will throw a DbUpdateConcurrencyException if the data was changes, so make sure you wrap it in a try…catch.

```
try
{
  ctx.SaveChanges();
}
catch (DbUpdateConcurrencyException)
{
  //the record was changed in the database, notify the user and fail
}
```

The first one wins approach is just this: fail in case a change has occurred.

Last One Wins

For this one, we will detect that a change has been made, and we overwrite it explicitly.

```
//assume the operation failed
var failed = true;

//loop until succeeded
do
{
```

```
    try
    {
      ctx.SaveChanges();

      //if succeeded, exit the loop
      failed = false;
    }
    catch (DbUpdateConcurrencyException ex)
    {
      var entity = ex.Entries.Single();

      //get the current values from the database
      var databaseValues = entity.GetDatabaseValues();

      //set the current database values as the original values
      //the original values are the ones that will be compared with the current
ones
      entity.OriginalValues.SetValues(databaseValues);
    }
}
while (failed);
```

Applying Optimistic Concurrency

Entity Framework by default does not perform the optimistic concurrency check. You can enable it by choosing the property or properties whose values will be compared with the current database values. This is done by applying a ConcurrencyCheckAttribute when mapping by attributes.

```
public class Project
{
  [Required]
  [MaxLength(50)]
  [ConcurrencyCheck]
  public String Name { get; set; }
}
```

Or in mapping by code.

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
  modelBuilder.Entity<Project>().Property(x => x.Name).IsConcurrencyToken();
}
```

What happens is this: when EF generates the SQL for an UPDATE operation, it will not only include a WHERE restriction for the primary key, but also for any properties marked for concurrency check, comparing their columns with the original values.
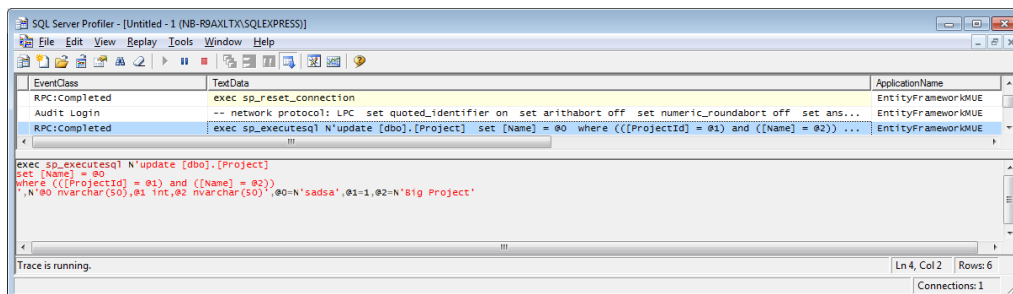
Figure 42: An update with a concurrency control check

If the number of affected records is not 1, this will likely be because the values in the database will not match the original values known by Entity Framework.

SQL Server has a data type whose values cannot be explicitly set, but instead change automatically whenever the record they belong to changes: ROWVERSION. Other databases offer similar functionality.

Because Entity Framework has a nice integration with SQL Server, columns of type ROWVERSION are supported for optimistic concurrency checks. For that, we need to map one such column into our model as a timestamp. First with attributes by applying a TimestampAttribute to a property, which needs to be of type byte array, and doesn't need a public setter.

```
[Timestamp]
public Byte [] RowVersion { get; protected set; }
```

And, for completeness, with fluent configuration.

```
modelBuilder.Entity<Project>().Property(x => x.RowVersion).IsRowVersion();
```

The behavior of TimestampAttribute is exactly identical to that of ConcurrencyCheckAttribute, but there can be only one property marked as timestamp per entity.

Detached Entities

A common scenario in web applications is this: you load some entity from the database, store it in the session, and in a subsequent request, get it from there and resume using it. This is all fine except that, if you are using Entity Framework Code First, you won't be using the same context instance on the two requests. This new context knows nothing about this instance. In this case, it is said that the entity is detached in relation to this context. This has the effect that any changes to this instance won't be tracked and any lazy loaded properties that weren't loaded when it was stored in the session won't be loaded as well.

What we need to do is to associate this instance with the new context.

```
//retrieve the instance from the ASP.NET context
var project = HttpContext.Current.Session["StoredProject"] as Project;
var ctx = new  ProjectsContext();
```

```
//attach it to the current context with a state of unchanged
ctx.Entry(project).State = EntityState.Unchanged;
```

After this, everything will work as expected.

Validation

Validation in Entity Framework Code First always occurs when an entity is about to be saved, which is normally a process triggered by the SaveChanges method. This can be prevented by setting the ValidateOnSaveEnabled property on the context.

```
//disable validation upon saving
ctx.Configuration.ValidateOnSaveEnabled = false;
```

We can see if the entries currently tracked by a context are valid by explicitly calling GetValidationErrors.

```
//all validation errors
var allErrors = ctx.GetValidationErrors();

//validation errors for a given entity
var errorsInEntity = ctx.Entry(p).GetValidationResult();
```

A validation result consists of instances of DbEntityValidationResult, of which there will be only one per invalid entity. This class offers the following properties.

Table 1: Validation result properties

Property	Purpose
Entry	The entity to which this validation refers to.
IsValid	Indicates if the entity is valid or not.
ValidationErrors	A collection of individual errors.

The ValidationErrors property contains a collection of DbValidationError entries, each exposing the following.

Table 2: Result error properties

Property	Purpose

ErrorMessage	The error message.
PropertyName	The name of the property on the entity that was considered invalid (can be empty if what was considered invalid was the entity as a whole).

If we attempt to save an entity with invalid values, a DbEntityValidationException will be thrown, and inside of it, there is the EntityValidationErrors collection which exposes all DbEntityValidationResult found.

```
try
{
  //try to save all changes
  ctx.SaveChanges();
}
catch (DbEntityValidationException ex)
{
  //validation errors were found that prevented saving changes
  var errors = ex.EntityValidationErrors.ToList();
}
```

Validation Attributes

Similar to the way we can use attributes to declare mapping options, we can also use attributes for declaring validation rules. A validation attribute must inherit from ValidationAttribute in the System.ComponentModel.DataAnnotations namespace and override one of its IsValid methods. There are some simple validation attributes that we can use out of the box and are in no way tied to Entity Framework.

Table 3: Validation attributes

Validation Attribute	Purpose
CompareAttribute	Compares two properties and fails if they are different.
CustomValidationAttribute	Executes a custom validation function and returns its value.
MaxLengthAttribute	Checks if a string property has a length greater than a given value.

Validation Attribute	Purpose
MinLengthAttribute	Checks if a string property has a length smaller than a given value.
RangeAttribute	Checks if the property's value is included in a given range.
RegularExpressionAttribute	Checks if a string matches a given regular expression.
RequiredAttribute	Checks if a property has a value; if the property is of type string, also checks if it is not empty.
StringLengthAttribute	Checks if a string property's length is contained within a given threshold.
MembershipPasswordAttribute	Checks if a string property (typically a password) matches the requirements of the default Membership Provider.

It is easy to implement a custom validation attribute. Here we can see a simple example that checks if a number is even.

```
[Serializable]
[AttributeUsage(AttributeTargets.Property, AllowMultiple = false, Inherited = true)
]
public sealed class IsEvenAttribute : ValidationAttribute
{
  protected override ValidationResult IsValid(Object value,
ValidationContext validationContext)
  {
    //check if the value is not null or empty
    if ((value != null) && (String.IsNullOrWhiteSpace(value.ToString()) == false))
    {
      TypeConverter converter = TypeDescriptor.GetConverter(value);
      //check if the value can be converted to a long
      if (converter.CanConvertTo(typeof(Int64)) == true)
      {
        Int64 number = (Int64) converter.ConvertTo(value, typeof(Int64));
        //fail if the number is even
        if ((number % 2) != 0)
        {
```

```
        return (new ValidationResult(this.ErrorMessage, new String[]
{ validationContext.MemberName }));
        }
      }
    }
    return (ValidationResult.Success);
  }
}
```

It can be applied to any property whose type can be converted to a long – it probably doesn't make sense in the case of a Budget, but let's pretend it does.

```
[IsEven(ErrorMessage = "Number must be even")]
public Int32 Number { get; set; }
```

We can also supply a custom validation method by applying a CustomValidationAttribute. Let's see how the same validation – "is even" – can be implemented using this technique. First, use the following attribute declaration.

```
[CustomValidation(typeof(CustomValidationRules), "IsEven", ErrorMessage = "Number m
ust be even")]
public Int32 Number { get; set; }
```

Next, use the following actual validation rule implementation.

```
public static ValidationResult IsEven(Object value, ValidationContext context)
{
  //check if the value is not empty
  if ((value != null) && (String.IsNullOrWhiteSpace(value.ToString()) == false))
  {
    TypeConverter converter = TypeDescriptor.GetConverter(value);
    //check if the value can be converted to a long
    if (converter.CanConvertTo(typeof(Int64)) == true)
    {
      Int64 number = (Int64) converter.ConvertTo(value, typeof(Int64));
      //fail if the number is even
      if ((number % 2) != 0)
      {
        return (new ValidationResult("Number must be even", new String[]
{ context.MemberName }));
      }
    }
  }
  return (ValidationResult.Success);
}
```

I chose to implement the validation function as static, but it is not required. In that case, the class where the function is declared must be safe to instantiate (not abstract with a public parameter-less constructor).

Implementing Self Validation

Another option for performing custom validations lies in the IValidatableObject interface. By implementing this interface, an entity can be self-validatable; that is, all validation logic is contained in itself. Let's see how.

```
public class Project : IValidatableObject
{
  //other members go here
  public IEnumerable<ValidationResult> Validate(ValidationContext context)
  {
    if (this.ProjectManager == null)
    {
      yield return (new ValidationResult("No project manager specified"));
    }
    if (this.Developers.Any() == false)
    {
      yield return (new ValidationResult("No developers specified"));
    }
    if ((this.End != null) && (this.End.Value < this.Start))
    {
      yield return (new ValidationResult("End of project is before start"));
    }
  }
}
```

Wrapping Up

You might have noticed that all these custom validation techniques – custom attributes, custom validation functions, and IValidatableObject implementation – all return ValidationResult instances, whereas Entity Framework Code First exposes validation results as collections of DbEntityValidationResult and DbValidationError. Don't worry, Entity Framework will take care of it for you!

So, what validation option is best? In my opinion, all have strong points and definitely all can be used together. I'll just leave some final remarks:

- If a validation attribute is sufficiently generic, it can be reused in many places.
- When we look at a class that uses attributes to express validation concerns, it is easy to see what we want.
- It also makes sense to have general purpose validation functions available as static methods, which may be invoked from either a validation attribute or otherwise.
- Finally, a class can self-validate itself in ways that are hard or even impossible to express using attributes. For example, think of properties whose values depend on other properties' values.

Transactions

Transactions in Entity Framework Code First come in two flavors:

- Implicit: method SaveChanges creates a transaction for wrapping all change sets that it will send to the database, if no ambient transaction exists; this is necessary for properly implementing a Unit of Work, where either all or no changes at all are applied simultaneously.
- Explicit: Entity Framework automatically enlists in ambient transactions created by TransactionScope.

Ambient transactions are nice because they can be automatically promoted to distributed transactions if any ADO.NET code tries to access more than one database server (or instance in a same physical server) while inside its scope. For that, the Distributed Transaction Coordinator service must be started.
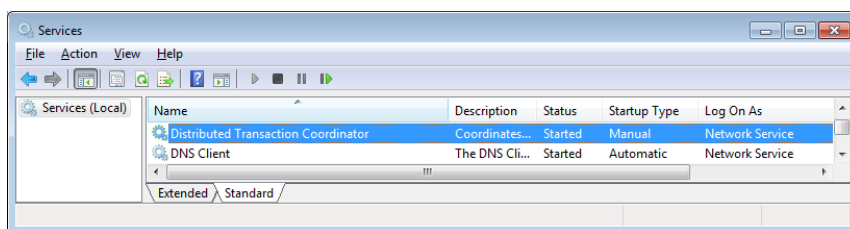

Figure 43: The Distributed Transaction Coordinator service

They are also the standard for transactions in the .NET world, with support that goes from databases to web services.

You should use transactions for primarily two reasons:

- For maintaining consistency when performing operations that must be simultaneously successful (think of a bank transfer where money leaving an account must enter another account).
- For assuring identical results for read operations where data can be simultaneously accessed and possibly changed by third parties.

That's enough theory. Transaction scopes in .NET are created like this.

```csharp
using (var tx = new TransactionScope())
{
  using (var ctx = new ProjectsContext())
  {
    //do changes

    //submit changes
    ctx.SaveChanges();
  }

  using (var ctx = new ProjectsContext())
  {
    //do more changes
```

```
    //submit changes
    ctx.SaveChanges();
  }

  //must be explicitly marked as complete
  tx.Complete();
}
```

This example shows a transaction scope wrapping two contexts. Changes submitted by any of these contexts are only committed if the Complete method is successfully executed, otherwise all changes are rolled back. Because there is no Rollback method, TransactionScope assumes a rollback when its Dispose method is called and Complete is not.

💡 ***Tip: Always wrap TransactionScope in a using block and keep it as short-lived as possible.***

Chapter 6 Spatial Data Types

Overview

SQL Server 2008 introduced two spatial data types. These offer the ability to write geographically-oriented queries in a more natural way, because the server knows about coordinates, distances, and so on. These types are:

- GEOMETRY: Euclidean (flat space) standard operations, defined by the Open Geospatial Consortium (OGC).
- GEOGRAPHY: Round-Earth operations, also defined by OGC.

They offer similar operations, which include:

- Calculating the distance between two points.
- Checking if a point is included in a polygon.
- Getting the intersection of two polygons.

Spatial types are not an exclusive of SQL Server, but not all Entity Framework providers support them; for more information, check out Entity Framework Provider Support for Spatial Types: http://msdn.microsoft.com/en-us/data/dn194325.aspx.

Entity Framework handles the spatial types pretty much like it does all other types. Namely, it can generate databases from entities containing properties of spatial types and uses LINQ for querying these types. Their .NET counterparts are DbGeography and DbGeometry.

A detailed explanation of these types is outside the scope of this chapter, so I'll just leave some querying examples.

First, let's start with a class with a geographical property.

```
public class Venue
{
  public Int32 VenueId { get; set; }

  public String Name { get; set; }

  public DbGeography Location { get; set; }
}
```

As an appetizer, two simple queries for calculating the distance between a fixed location point and some venues and for checking the points that fall inside an area using LINQ are as follows.

```
//a fixed location in Well-known Text (WKT) format
var location = DbGeography.FromText(string.Format("POINT({0} {1})", 41, 8));

//an area in WKT and Spatial Reference System Identifier (SRID) 4326
```

```
var area = DbGeography.MultiPointFromText("MULTIPOINT(53.095124 -
0.864716, 53.021255 -1.337128, 52.808019 -1.345367, 52.86153 -1.018524)", 4326);

//the distance from all stored locations to the fixed location
var venuesAndDistanceToLocation = ctx.Venues
.OrderBy(v = v.Location.Distance(location))
.Select(v => new { Venue = v, Distance = v.Location.Distance(location) }).ToList();

//venues inside the area
var pointInsideArea = ctx.Venues.Where(x => area.Intersects(x.Location)).ToList();
```

Chapter 7 Handling Events

Saving and Loading Events

Events are .NET's implementation of the Observer design pattern, used to decouple a class from other classes that are interested in changes that may occur in it. While Entity Framework Code First does not directly expose any events, because it sits on top of the "classic" Entity Framework, it is very easy to make use of the events it exposes.

The Entity Framework's ObjectContext class exposes two events.

Table 4: ObjectContext events

Event	Purpose
ObjectMaterialized	Raised when an entity is materialized, as the result of the execution of a query.
SavingChanges	Raised when the context is about to save its attached entities.

Why do we need this? Well, we may want to perform some additional tasks just after an entity is loaded from the database or just before it is about to be saved or deleted.

One way to bring these events to Code First land is as follows.

```
public class ProjectsContext : DbContext
{
  public ProjectsContext()
  {
    this.AddEventHandlers();
  }

  //raised when the context is about to save dirty entities
  public event EventHandler<EventArgs> SavingChanges;
  //raised when the context instantiates an entity as the result of a query
  public event EventHandler<ObjectMaterializedEventArgs> ObjectMaterialized;

  public void AddEventHandlers()
  {
    //access the underlying ObjectContext
    var octx = (this as IObjectContextAdapter).ObjectContext;
    //add local event handlers
    octx.SavingChanges += (s, e) => this.OnSavingChanges(e);
    octx.ObjectMaterialized += (s, e) => this.OnObjectMaterialized(e);
  }
```

```csharp
    protected virtual void OnObjectMaterialized(ObjectMaterializedEventArgs e)
    {
      var handler = this.ObjectMaterialized;

      if (handler != null)
      {
        //raise the ObjectMaterialized event
        handler(this, e);
      }
    }

    protected virtual void OnSavingChanges(EventArgs e)
    {
      var handler = this.SavingChanges;

      if (handler != null)
      {
        //raise the SavingChanges event
        handler(this, e);
      }
    }
}
```

So, we have two options for handling the ObjectMaterialized and the SavingChanges events:

- Inheriting classes can override the OnObjectMaterialized and OnSavingChanges methods.
- Interested parties can subscribe to the SavingChanges and ObjectMaterialized events.

Now imagine this: you want to define a marked interface such as IImmutable that, when implemented by an entity, will prevent it from ever being tracked by Entity Framework. Here's a possible solution for this scenario.

```csharp
//a simple marker interface
public interface IImmutable { }

public class Project : IImmutable { /* … */ }

public class ProjectsContext : DbContext
{
  protected virtual void OnObjectMaterialized(ObjectMaterializedEventArgs e)
  {
    var handler = this.ObjectMaterialized;

    if (handler != null)
    {
      handler(this, e);
    }

    //check if the entity is meant to be immutable
    if (e.Entity is IImmutable)
```

```
    {
      //if so, detach it from the context
      this.Entry(e.Entity).State = EntityState.Detached;
    }
  }

  protected virtual void OnSavingChanges(EventArgs e)
  {
    var handler = this.SavingChanges;

    if (handler != null)
    {
      handler(this, e);
    }

    //get all entities that are not unchanged (added, deleted or modified)
    foreach (var immutable in this.ChangeTracker.Entries()
    .Where(x => x.State != EntityState.Unchanged && x.Entity is
IImmutable).Select(x => x.Entity).ToList())
    {
      //set the entity as detached
      this.Entry(e.Entity).State = EntityState.Detached;
    }
  }
}
```

Very quickly, what it does is:

- All IImmutable entities loaded from queries (ObjectMaterialized) are immediately detached from the context.
- Dirty IImmutable entities about to be saved (SavingChanges) are set to detached, so they aren't saved.

In another case, there is auditing changes made to an entity. For that we want to record:

- The user who first created the entity.
- When it was created.
- The user who last modified the entity.
- When it was last updated.

We will start by defining a common auditing interface, IAuditable, where these auditing properties are defined, and then we will provide an appropriate implementation of OnSavingChanges.

```
//an interface for the auditing properties
public interface IAuditable
{
  String CreatedBy { get; set; }

  DateTime CreatedAt { get; set; }

  String UpdatedBy { get; set; }
```

```
   DateTime UpdatedAt { get; set; }
}

public class Project : IAuditable { /* … */ }

public class ProjectsContext : DbContext
{
  protected virtual void OnSavingChanges(EventArgs e)
  {
    var handler = this.SavingChanges;

    if (handler != null)
    {
      handler(this, e);
    }
    foreach (var auditable in this.ChangeTracker.Entries()
.Where(x => x.State == EntityState.Added).Select(x => x.Entity).OfType<IAuditable>(
))
    {
      auditable.CreatedAt = DateTime.Now;
      auditable.CreatedBy = Thread.CurrentPrincipal.Identity.Name;
    }

    foreach (var auditable in this.ChangeTracker.Entries()
.Where(x => x.State == EntityState.Modified).Select(x => x.Entity)
.OfType<IAuditable>())
    {
      auditable.UpdatedAt = DateTime.Now;
      auditable.UpdatedBy = Thread.CurrentPrincipal.Identity.Name;
    }
  }
}
```

On the OnSavingChanges method we:

- List all IAuditable entities that are new and set their CreatedAt and CreatedBy properties.
- At the same time, all modified entities have their UpdatedAt and UpdatedBy properties set.

Another typical use for the SavingChanges event is to generate a value for the primary key when IDENTITY cannot be used. In this case, we need to fetch this next value from somewhere, such as a database sequence, function, or table, and assign it to the identifier property.

```
//an interface for accessing the identifier property of entities that require
explicit identifier assignments
public interface IHasGeneratedIdentifier
{
  Int32 Identifier { get; set; }
}

public class ProjectsContext : DbContext
{
  protected virtual void OnSavingChanges(EventArgs e)
```

```
  {
    var handler = this.SavingChanges;

    if (handler != null)
    {
      handler(this, e);
    }

    foreach (var entity in this.ChangeTracker.Entries()
.Where(x => x.State == EntityState.Added).Select(x => x.Entity)
.OfType<IHasGeneratedIdentifier>())
    {
      //call some function that returns a valid identifier
      entity.Identifier = this.Database.SqlQuery<Int32>("EXEC GetNextId()");
    }
  }
}
```

💡 *Tip: How you implement the GetNextId function is up to you.*

Chapter 8 Extending Entity Framework

Calling Database Functions

LINQ offers operators for some common database functions, but what happens if you want to call another operator that is not supported?

For example, in SQL Server, we could define a function called CalculateValue that would take a string as its only parameter and return an integer based on it.

```sql
CREATE FUNCTION CalculateValue
(
        @parameter NVARCHAR
)
RETURNS INT
AS
BEGIN
        -- roll out your own implementation

        RETURN 0
END
```

It is possible to call this function on a LINQ query; for that, we need to declare an extension method and decorate it with the EdmFunctionAttribute. This tells Entity Framework that it should call a database function with the given name and signature, and pass it the extension method parameters. Let's see an example.

```csharp
public static class StringExtensions
{
  [EdmFunction("SqlServer", "CalculateValue")]
  public static Int32 CalculateValue(this String phrase)
  {
    throw (new NotImplementedException());
  }
}
```

The first parameter to EdmFunctionAttribute is a namespace, but just use whatever you like. The second is the function name, in case it is different from the .NET method name. This implementation throws an exception because it is not meant to be called directly, but on the database.

```csharp
var value = ctx.Projects.Select(x => x.Name.CalculateValue()).ToList();
```

You can use this technique for calling other functions, including ones defined by you, but this won't work with Table-Valued functions (TVL), only with scalar functions. Also, you can only pass it arguments of basic types, not entities or complex types.

> 📝 ***Note: This is the exact same technique that the methods on [SqlFunctions](#) and [EntityFunctions](#) classes use.***

Implementing LINQ Extension Methods

Another technique consists of leveraging LINQ expressions to build complex queries from extension methods.

The [BETWEEN](#) SQL operator does not have a corresponding LINQ expression. We can use two simultaneous conditions in our LINQ expression, one for the low end of the range (> X) and for the high end (< Y). We can also implement a LINQ extension method to provide us this functionality with a single expression.

```csharp
public static class QueryableExtensions
{
  public static IQueryable<TSource> Between<TSource, TKey>(
    this IQueryable<TSource> source,
    Expression<Func<TSource, TKey>> property, TKey low, TKey high
  ) where TKey : IComparable<TKey>
  {
    var sourceParameter = Expression.Parameter(typeof(TSource));
    var body = property.Body;
    var parameter = property.Parameters[0];
    var compareMethod = typeof(TKey).GetMethod("CompareTo",
      new Type[] { typeof(TKey) });
    var zero = Expression.Constant(0, typeof(Int32));
    var upper = Expression.LessThanOrEqual(Expression.Call(body, compareMethod,
      Expression.Constant(high)), zero);
    var lower = Expression.GreaterThanOrEqual(Expression.Call(body, compareMethod,
      Expression.Constant(low)), zero);
    var andExpression = Expression.AndAlso(upper, lower);
    var whereCallExpression = Expression.Call
    (
      typeof(Queryable),
      "Where",
      new Type[] { source.ElementType },
      source.Expression,
      Expression.Lambda<Func<TSource, Boolean>>(andExpression,
        new ParameterExpression[] { parameter })
    );

    return (source.Provider.CreateQuery<TSource>(whereCallExpression));
  }
}
```

For a good understanding on how this is implemented, it is crucial to understand LINQ expressions. There are some good links on the Internet. This technology, although complex to master, has great potential and has drawn a lot of attention.

This is an extension method on [IQueryable<T>](#), and it can be used like this.

```
//get projects starting between two dates
var projectsBetweenTodayAndTheDayBefore = ctx.Projects
.Between(x => x.Start, DateTime.Today.AddDays(-1), DateTime.Today).ToList();

//projects with 10 to 20 resources
var projectsWithTwoOrThreeResources = ctx.Projects.Select(x =>
new { x.Name, ResourceCount = x.ProjectResources.Count() })
.Between(x => x.ResourceCount, 10, 20).ToList();
```

The LINQ provider will happily chew the new expression and translate it to the appropriate SQL.

Chapter 9 Exposing Data to the World

Overview

We live in a connected world, and distributed applications are common these days. The Microsoft .NET stack offers some interesting technologies for exposing entities to the outside world by leveraging on Entity Framework. In the next pages, we will be talking about two web service technologies and one for dynamic CRUD interface.

WCF Data Services

WCF Data Services is Microsoft's implementation of the OData standard. Basically, this is a standard for the publication, querying, and exchange of data coming from entities, such as a REST web service, and it is implemented on top of WCF. You can issue queries through the URL and it is even possible to generate strongly typed proxies that use LINQ for querying the published model.

Visual Studio has an item type called WCF Data Service in Web projects.
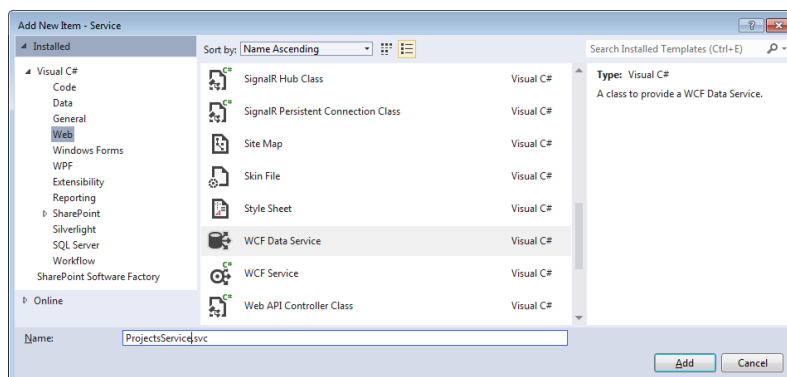

Figure 44: WCF Data Service item type

When adding an item of this type, Visual Studio creates a DataService<T>-derived class that will be the endpoint for the OData service. We need to replace the T parameter with a context, but for now we must use ObjectContext instead of our DbContext class, because unfortunately it still does not recognize it as a valid context. Complementary to that, we need to return an instance of ObjectContext, which we can obtain from our own context in an override of CreateDataSource.

```
public class ProjectsService : DataService<ObjectContext>
{
  public static void InitializeService(DataServiceConfiguration config)
  {
    config.SetEntitySetAccessRule("*", EntitySetRights.AllRead);
    config.SetServiceOperationAccessRule("*", ServiceOperationRights.All);
    config.DataServiceBehavior.MaxProtocolVersion = DataServiceProtocolVersion.V3;
```

```
    }

  protected override ObjectContext CreateDataSource()
  {
    ProjectsContext ctx = new ProjectsContext();
    ObjectContext octx = (ctx as IObjectContextAdapter).ObjectContext;

    return (octx);
  }
}
```

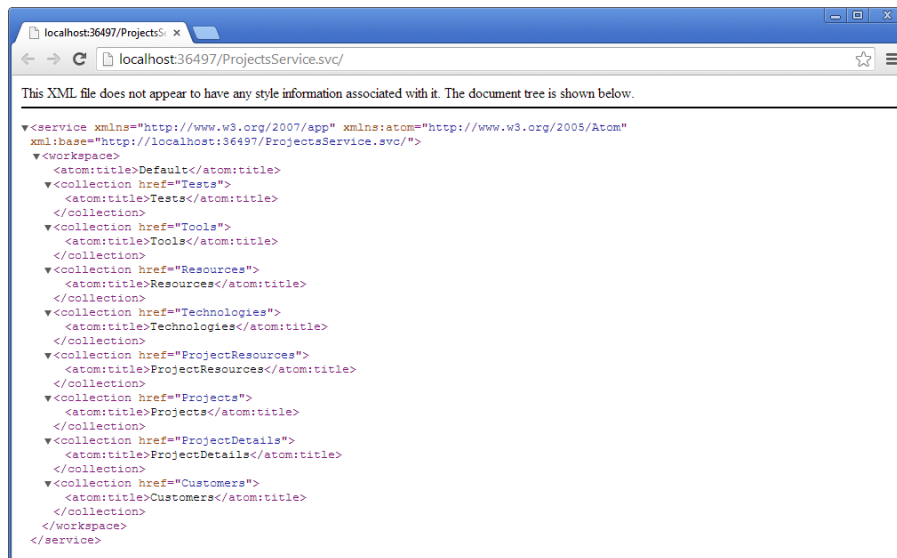After that, we are ready to query our data service.

Figure 45: A WCF Data Service

💡 ***Tip: In the current version of WCF Data Services, properties of enumerated or spatial types are not supported.***

As you can see, when you access the service URL, you will get all of the collections exposed by your context in XML format. We can then navigate to each of these collections, and see all of its records.
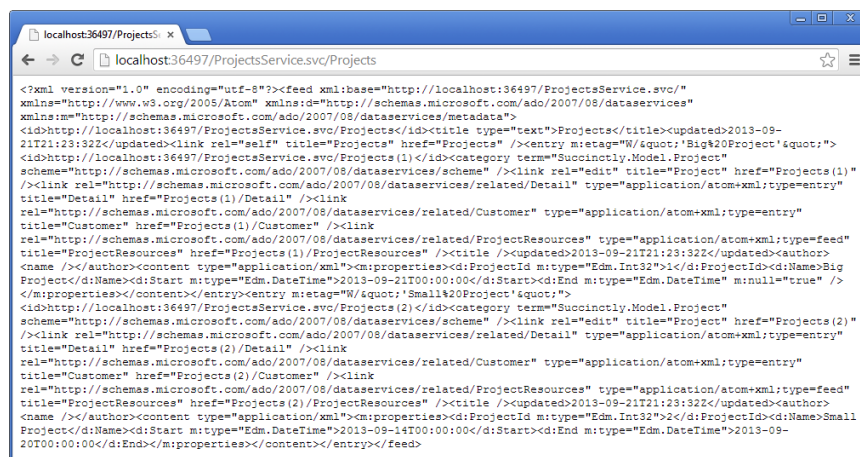
Entity Framework Code First Succinctly

Figure 46: A WCF Data Service returning all data for an entity

It is possible to query the model through the URL, but I won't cover that here. Please see here for the full specification: http://www.odata.org/documentation/odata-v3-documentation/url-conventions/.

ASP.NET Web API

Another technology that is gaining a lot of attention is ASP.NET Web API. It offers another REST API for exposing an entity model, but this one was introduced as part of MVC 4, and is built on it, which means it is based on controllers and actions. It is sometimes regarded as being to web services what Entity Framework Code First is to data access.

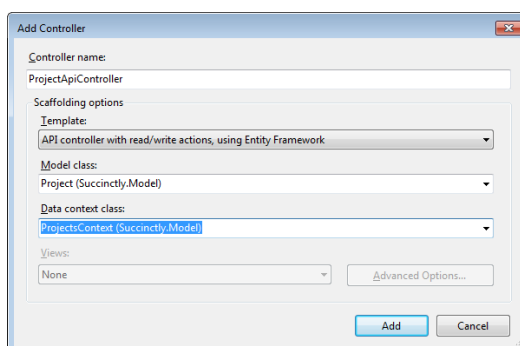We create a Web API controller on the **Controllers** folder in an MVC project.

Figure 47: A Web API controller based on an Entity Framework context

Visual Studio already sets up some configuration for us, but we typically need some tweaks to have everything working well. One of these is having results coming in JSON format, and the other is enabling proper serialization of circular references in model entities (for example, in our model, a Customer has a collection of Projects and each project references the Customer).

Let's access the **WebApiConfig** class in the **App_Start** method and add the following lines.

```
//remove the XML formatter – skip this if you prefer XML over JSON
```

```
GlobalConfiguration.Configuration.Formatters.XmlFormatter.SupportedMediaTypes
.Clear();

//add support for circular references to the JSON serializer – do the same for XML
GlobalConfiguration.Configuration.Formatters.JsonFormatter.SerializerSettings
.ReferenceLoopHandling = ReferenceLoopHandling.Serialize;
GlobalConfiguration.Configuration.Formatters.JsonFormatter.SerializerSettings
.PreserveReferencesHandling = PreserveReferencesHandling.Objects;
```

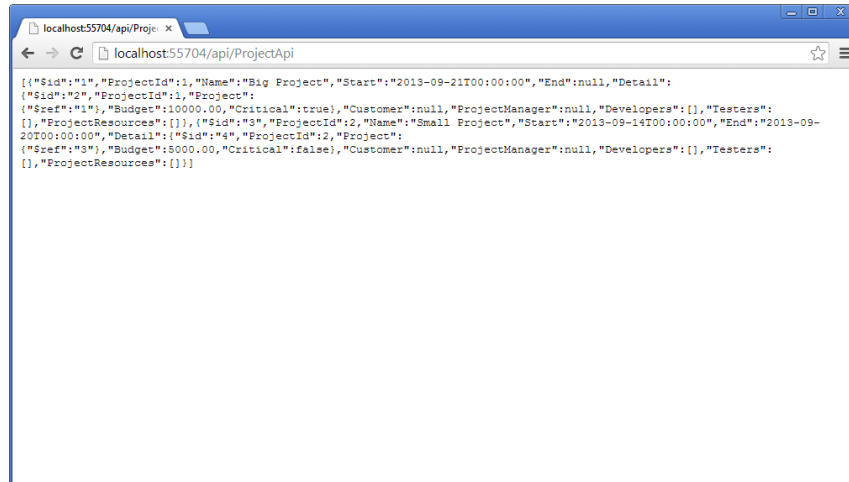We can now navigate to the route configured for Web API controllers.

Figure 48: A Web API endpoint

There's a lot more to Web API than this, and I advise you to navigate to its home site, at http://www.asp.net/web-api, and check it out for yourself. It is certainly worth it.

ASP.NET Dynamic Data

ASP.NET Dynamic Data has been around for some years now; for those not familiar with it, it is a technology that generates Create Read Update Delete (CRUD) web interfaces automatically for an entity model, such as those created for Entity Framework Code First. This can be a great help when we quickly need to implement a site for editing records in a simple way.

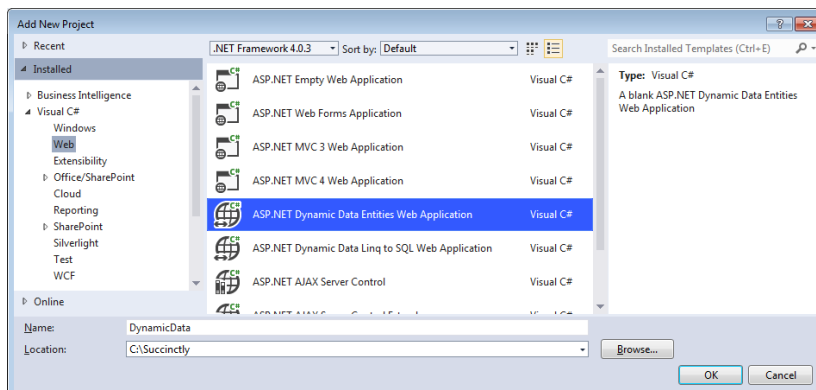We create an ASP.NET Dynamic Data project on its own.

Figure 49: ASP.NET Dynamic Data project

After we create the new project, we need to:

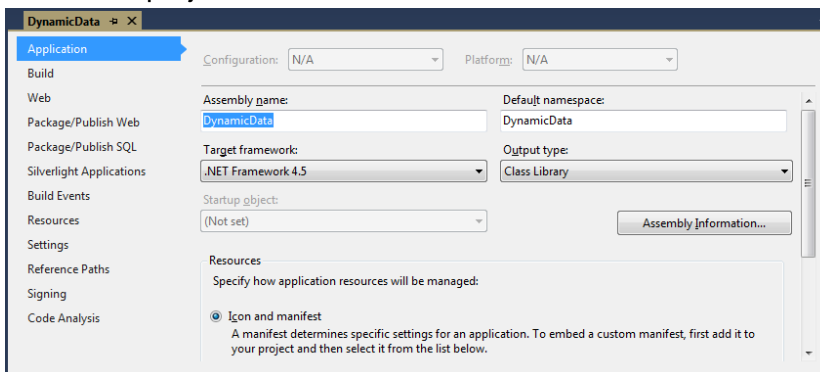1. Turn it into a .NET 4.5 project.

Figure 50: ASP.NET Dynamic Data project properties

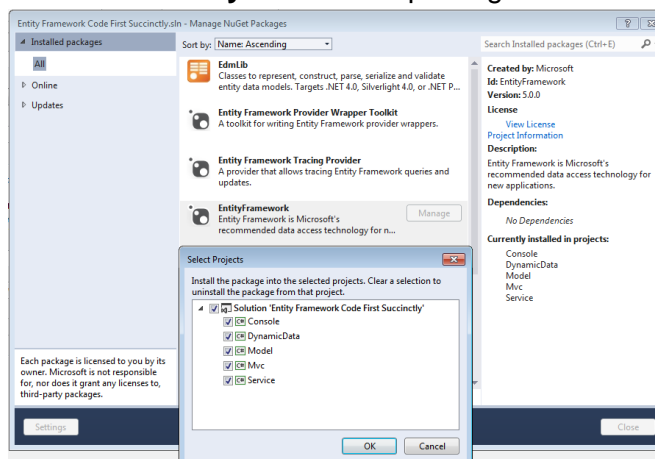2. Add a NuGet reference to the **EntityFramework** package.

Figure 51: Adding a NuGet reference

3. Add a reference to the project containing the Entity Framework Code First context (in our case, it is the **Model** project).
4. Copy your connection string into the new project's **Web.config** file.

Now we have to tell Dynamic Data to use our context. For that, open the **Global** class and go to its **RegisterRoutes** method. Add the following line.

```
DefaultModel.RegisterContext(() => (new ProjectsContext() as IObjectContextAdapter)
.ObjectContext, new ContextConfiguration() { ScaffoldAllTables = true });
```

And we're ready to go! Start the project and you will see the entry page.
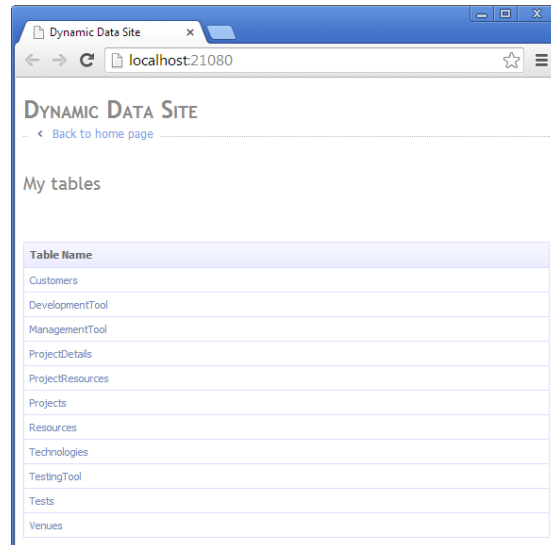

Figure 52: ASP.NET Dynamic Data entry page
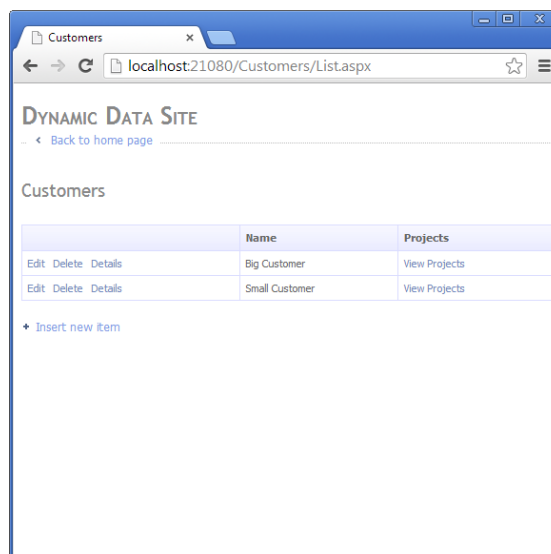
Then navigate to some entity.

Figure 53: ASP.NET Dynamic Data entity items

You will be glad to know that enumerated and spatial types are fully supported. Let's leave Dynamic Data for now, there's a lot more to be said, and I hope I got your attention! For further information, go to http://msdn.microsoft.com/en-us/library/cc488545.aspx.

Chapter 10 Tracing and Profiling

Getting the SQL for a Query

LINQ and Entity SQL queries are translated by Entity Framework to SQL. Both of them offer a way to see what the generated SQL looks like; this is achieved by the ToTraceString method, which is a public method of the ObjectQuery<T> class. The LINQ implementation of Code First also has a way to get to this ObjectQuery<T>, but it requires reflection. Here's a possible implementation of a general-purpose tracing method.

```csharp
public static String ToSqlString<TEntity>(this IQueryable<TEntity> queryable)
where TEntity : class
{
  ObjectQuery<TEntity> objectQuery = null;

  if (queryable is ObjectQuery<TEntity>)
  {
    objectQuery = queryable as ObjectQuery<TEntity>;
  }
  else if (queryable is DbQuery<TEntity>)
  {
    var dbQuery = queryable as DbQuery<TEntity>;
    var iqProp = dbQuery.GetType().GetProperty("InternalQuery",
BindingFlags.Instance | BindingFlags.NonPublic | BindingFlags.Public);
    var iq = iqProp.GetValue(dbQuery);
    var oqProp = iq.GetType().GetProperty("ObjectQuery", BindingFlags.Instance |
BindingFlags.NonPublic | BindingFlags.Public);

    objectQuery = oqProp.GetValue(iq) as ObjectQuery<TEntity>;
  }
  else
  {
    throw (new ArgumentException("queryable"));
  }

  var sqlString = objectQuery.ToTraceString();

  foreach (var objectParam in objectQuery.Parameters)
  {
    if ((objectParam.ParameterType == typeof(String))
    || (objectParam.ParameterType == typeof(DateTime))
    || (objectParam.ParameterType == typeof(DateTime?)))
    {
      sqlString = sqlString.Replace(String.Format("@{0}", objectParam.Name),
      String.Format("'{0}'", objectParam.Value.ToString()));
    }
    else if ((objectParam.ParameterType == typeof(Boolean))
    || (objectParam.ParameterType == typeof(Boolean?)))
    {
      sqlString = sqlString.Replace(String.Format("@{0}", objectParam.Name),
```

```
String.Format("{0}", Boolean.Parse(objectParam.Value.ToString()) ? 1 : 0));
    }
    else
    {
      sqlString = sqlString.Replace(String.Format("@{0}", objectParam.Name),
String.Format("{0}", objectParam.Value.ToString()));
    }
  }

  return (sqlString);
}
```

This method will pick up the SQL string returned by ToTraceString and will replace the parameter placeholders by the actual parameters' values. You can use it on any IQueryable<T> implementation. If the argument is already an ObjectQuery<T>, then it is straightforward, and if it is a DbQuery<T> (a LINQ query from Code First), it first extracts from it the underlying ObjectQuery<T>.

```
//get the SQL for an Entity SQL query
var finishedProjectsSQL = octx.CreateQuery<Project>(
"SELECT VALUE p FROM Projects AS P WHERE p.[End] IS NOT NULL").ToTraceString();

//get the SQL for an Entity SQL query
var projectsStartingAWeekAgoSQL = (from p in ctx.Projects where p.Start ==
DateTime.Today.AddDays(-1) select p).ToSqlString();

//get the SQL for all the Projects using the ToSqlString extension method
var allProjectsSQL = ctx.Projects.ToSqlString();
```

MiniProfiler

MiniProfiler is an open source project that offers a code profiler for ASP.NET MVC and Entity Framework. I am going to demonstrate how to use it in an MVC project, but except for the MVC console, it might as well be used in a Windows Forms, Console, or even Web Forms application.

In order to use it, you need to first obtain its core package from NuGet.

```
PM> Install-Package MiniProfiler
```

Then the MVC and EF packages.

```
PM> Install-Package MiniProfiler.MVC3
```

```
PM> Install-Package MiniProfiler.EF
```

In **Global** class of you web application, you will need to add the following lines to the **Application_Start** method.

```
MiniProfilerEF.Initialize();
MiniProfiler.Settings.SqlFormatter = new SqlServerFormatter();
```

In the layout view (or master page, depending on your view engine of choice), you need to add a specific method call for adding the required JavaScript.

For Razor, it will be the **~Views\Shared_Layout.cshtml** file.

```
@StackExchange.Profiling.MiniProfiler.RenderIncludes()
```

Whereas for ASPX, it will be the master page **~Views\Shared\Site.Master**.

```
<%: StackExchange.Profiling.MiniProfiler.RenderIncludes() %>
```

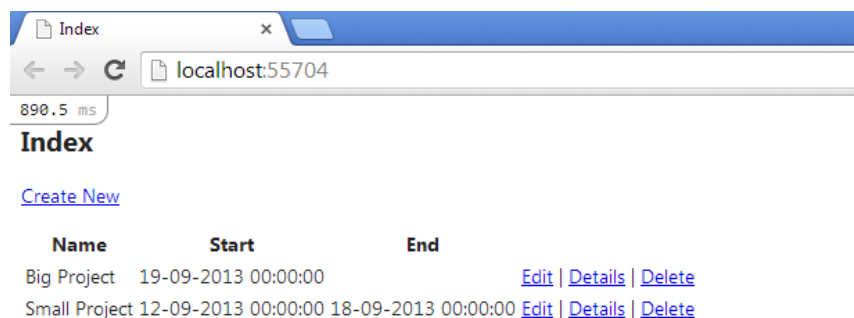After you do that, you will start to see the MiniProfiler console on the top left corner of every page.

Figure 54: MiniProfiler indicator

By clicking on it, it will expand and show some statistics for the current request.
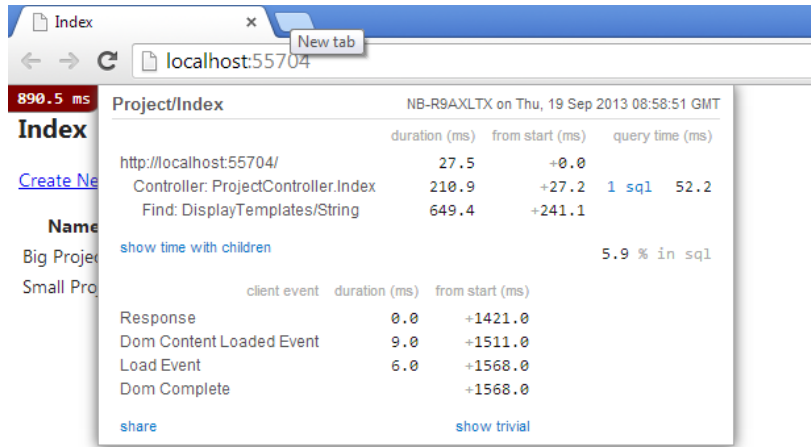
Figure 55: MiniProfiler MVC console

Up until now, this has nothing to do with Entity Framework, but if you click the **sql** link, then all the fun starts.
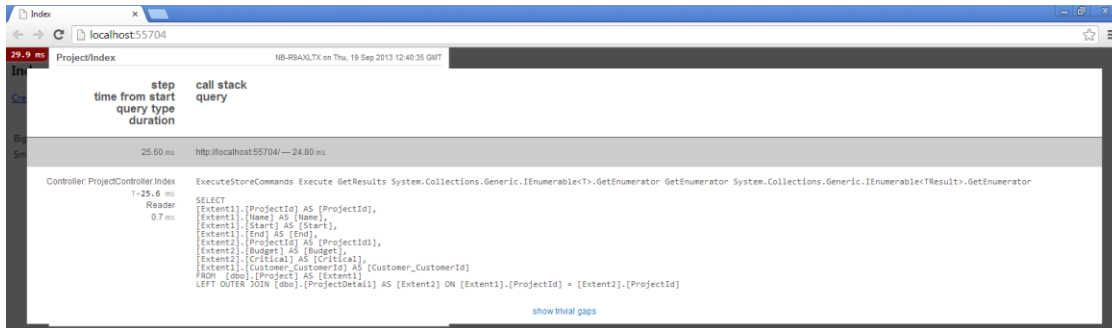
Figure 56: MiniProfiler SQL console

MiniProfiler will show you all the queries executed as part of the current web request, will detect duplicate queries, and will also show some other problems, like SELECT N+1.

If you do not want to use the MVC console, you can still get relevant information from the current profiler instance.

```
//some statistics
var nonQueriesCount = MiniProfiler.Current.ExecutedNonQueries;
var readersCount = MiniProfiler.Current.ExecutedReaders;
var scalarsCount = MiniProfiler.Current.ExecutedScalars;
var duration = MiniProfiler.Current.DurationMillisecondsInSql;

//all queries executed as part of the current request
var timings = MiniProfiler.Current.GetSqlTimings();
var sql = timings.First().FormattedCommandString;
var isDuplicate = timings.First().IsDuplicate;
var stackTrace = timings.First().StackTraceSnippet;
var parameters = timings.First().Parameters;
```

SQL Server Profiler

SQL Server Profiler is an invaluable tool included with the commercial editions of SQL Server from 2005 onwards. In a nutshell, it allows you to monitor, in real time, the SQL that is sent to the database.

By using SQL Server Profiler, you can get an inside picture of what Entity Framework is doing behind your back. For a simple query such as `ctx.Projects.OrderBy(x => x.Start).ToList()`, the following SQL will be issued.
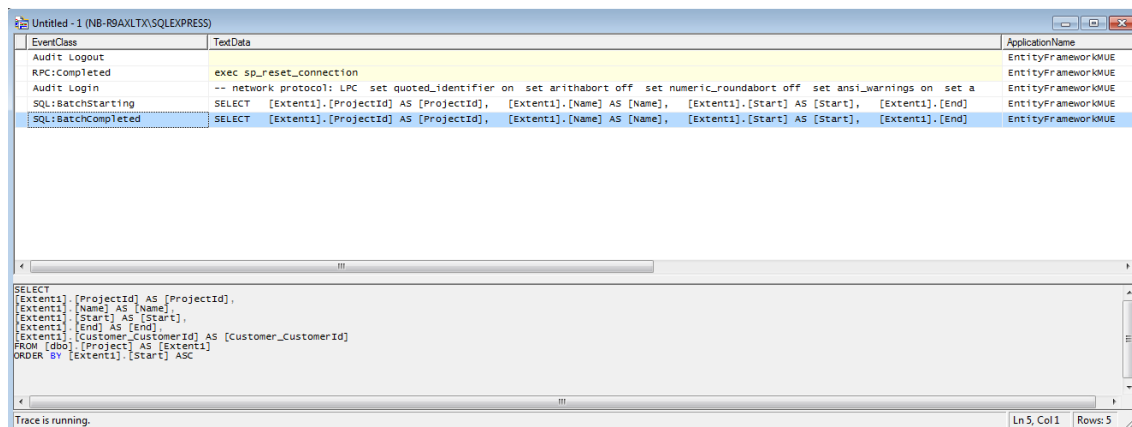
Figure 57: SQL Server Profiler output

Did you notice the ApplicationName column? Entity Framework always sets the application name as "EntityFrameworkMUE", this way you can always tell which SQL queries are sent by it, and you can create a filter on SQL Server Profiler to only show these entries. If you want, you can a different application name, by supplying the Application Name parameter in the connection string.

```
<connectionStrings>
  <add name="ProjectsContext"
  connectionString="Data Source=.\SQLEXPRESS;Integrated Security=SSPI;
  Initial Catalog=Succinctly;MultipleActiveResultSets=true;Application
Name=ProjectsContext"
  providerName="System.Data.SqlClient"/>
</connectionStrings>
```

Chapter 11 Performance Optimizations

Filter Entities in the Database

You are certainly aware that Entity Framework works with two flavors of LINQ:

- LINQ to Objects: operations are performed in memory.
- LINQ to Entities: operations are performed in the database.

LINQ to Entities queries that return collections are executed immediately after calling a "terminal" method – one of ToList, ToArray, or ToDictionary. After that, the results are materialized and are therefore stored in the process' memory space. Being instances of IEnumerable<T>, they can be manipulated with LINQ to Objects standard operators without us even noticing. What this means is, it is totally different to issue these two queries, because one will be executed fully by the database, while the other will be executed by the current process.

```
//LINQ to Objects: all Technologies are retrieved from the database and filtered in
memory
var technologies = ctx.Technologies.ToList().Where(x => x.Resources.Any());

//LINQ to Entities: Technologies are filtered in the database, and only after
retrieved into memory
var technologies = ctx.Technologies.Where(x => x.Resources.Any()).ToList();
```

Beware, you may be bringing a lot more records than what you expected!

Do Not Track Entities Not Meant For Change

Entity Framework has a first level (or local) cache where all the entities known by an EF context, loaded from queries or explicitly added, are stored. This happens so that when time comes to save changes to the database, EF goes through this cache and checks which ones need to be saved, meaning which ones need to be inserted, updated, or deleted. What happens if you load a number of entities from queries when EF has to save? It needs to go through all of them, to see which have changed, and that constitutes the memory increase.

If you don't need to keep track of the entities that result from a query, since they are for displaying only, you should apply the AsNoTracking extension method.

```
//no caching
var technologies = ctx.Technologies.AsNoTracking().ToList();

var technologiesWithResources = ctx.Technologies.Where(x => x.Resources.Any())
.AsNoTracking().ToList();

var localTechnologies = ctx.Technologies.Local.Any();   //false
```

This even causes the query execution to run faster, because EF doesn't have to store each resulting entity in the cache.

Disable Automatic Detection of Changes

Another optimization is related to the way Entity Framework runs the change tracking algorithm. By default, the DetectChanges method is called automatically in a number of occasions, such as when an entity is explicitly added to the context, when a query is run, etc. This results in degrading performance whenever a big number of entities is being tracked.

The workaround is to disable the automatic change tracking by setting the AutoDetectChangesEnabled property appropriately.

```
//disable automatic change tracking
ctx.Configuration.AutoDetectChangesEnabled = false;
```

Do not be alarmed: whenever SaveChanges is called, it will then call DetectChanges and everything will work fine. It is safe to disable this.

Use Lazy, Explicit or Eager Loading Where Appropriate

As you saw on section *Lazy, Explicit and Eager Loading*, you have a number of options when it comes to loading navigation properties. Generally speaking, when you are certain of the need to access a reference property or go through all of the child entities present in a collection, you should eager load them with their containing entity. There's a known problem called SELECT N + 1 which illustrates just that: you issue one base query that returns N elements and then you issue another N queries, one for each reference/collection that you want to access.

Performing eager loading is achieved by applying the Include extension method.

```
//eager load the Technologies for each
Resource
var resourcesIncludingTechnologies = ctx.Resources.Include(x => x.Technologies)
.ToList();

//eager load the Customer for each Project
var projectsIncludingCustomers = ctx.Projects.Include("Customer").ToList();
```

This way you can potentially save a lot of queries, but it can also bring much more data than the one you need.

Use Projections

As you know, normally the LINQ and Entity SQL queries return full entities; that is, for each entities, they bring along all of its mapped properties (except references and collections). Sometimes we don't need the full entity, but just some parts of it, or even something calculated from some parts of the entity. For that purpose, we use projections.

Projections allow us to reduce significantly the data returned by hand picking just what we need. Here are some examples in both LINQ and Entity SQL.

```
//return the resources and project names only with LINQ
var resourcesXprojects = ctx.Projects.SelectMany(x => x.ProjectResources)
.Select(x => new { Resource = x.Resource.Name, Project = x.Project.Name }).ToList()
;

//return the customer names and their project counts with LINQ
var customersAndProjectCount = ctx.Customers
.Select(x => new { x.Name, Count = x.Projects.Count() }).ToList();

//return the project name and its duration with ESQL
var projectNameAndDuration = octx.CreateQuery<Object>("SELECT p.Name,
DIFFDAYS(p.Start, p.[End]) FROM Projects AS p WHERE p.[End] IS NOT NULL").ToList();

//return the customer name and a conditional column with ESQL
var customersAndProjectRange = octx.CreateQuery<Object>(
"SELECT p.Customer.Name, CASE WHEN COUNT(p.Name) > 10 THEN 'Lots' ELSE 'Few' END AS

Amount FROM Projects AS p GROUP BY p.Customer").ToList();
```

Projections with LINQ depend on anonymous types, but you can also select the result into a .NET class for better access to its properties.

```
//return the customer names and their project counts into a dictionary with LINQ
var customersAndProjectCountDictionary = ctx.Customers
.Select(x => new { x.Name, Count = x.Projects.Count() })
.ToDictionary(x => x.Name, x => x.Count);
```

Disabling Validations Upon Saving

We discussed the validation API on section Validation. Validations are triggered for each entity when the context is about to save them, and if you have lots of them, this may take some time.

When you are 100% sure that the entities you are going to save are all valid, you can disable their validation by disabling the ValidateOnSaveEnabled property.

```
//disable automatic validation upon save
ctx.Configuration.ValidateOnSaveEnabled = false;
```

This is a global setting, so beware!

Working with Disconnected Entities

When saving an entity, if you need to store in a property a reference another entity for which you know the primary key, then instead of loading it with Entity Framework, just assign it a blank entity with only the identifier property filled.

```
//save a new project referencing an existing Customer
var newProject = new Project { Name = "Some Project", Customer =
new Customer { CustomerId = 1 } /*ctx.Customers.Find(1)*/ };

ctx.Projects.Add(newProject);
ctx.SaveChanges();
```

This will work fine because Entity Framework only needs the foreign key set.

Also for deletes, no need to load the entity beforehand, its primary key will do.

```
//delete a Customer by id
//ctx.Customers.Remove(ctx.Customers.Find(1));
ctx.Entry(new Customer { CustomerId = 1 }).State = EntityState.Deleted;
ctx.SaveChanges();
```

Do Not Use IDENTITY for Batch Inserts

While the IDENTITY identifier generation strategy may well be the most obvious for those coming from the SQL Server world, it does have some problems when it comes to ORM; it is not appropriate for batching scenarios. Since the primary key is generated in the database and each inserted entity must be hydrated – meaning, its identifier must be set - as soon as it is persisted, SQL Server needs to issue an additional SELECT immediately after the INSERT to fetch the generated key. Let's see an example where we use IDENTITY.

```
ctx.Save(new Parent { Name = "Batching Example" });
ctx.SaveChanges();
```

We can see the generated SQL in SQL Server Profiler (notice the INSERT followed by a SELECT).

Figure 58: Inserting a record and obtaining the generated key

If we change the generation strategy to manual and explicitly set the primary key beforehand then we only have one INSERT.

EventClass	TextData		ApplicationName
RPC:Completed	exec sp_executesql N'insert [dbo].[Parent]([ParentId], [Name]) values (@0, @1) ',N'@0 int,@1 nvarchar(max) ',@0=10,@1=N'Batch		EntityFrameworkMUE

```
exec sp_executesql N'insert [dbo].[Parent]([ParentId], [Name])
values (@0, @1)
',N'@0 int,@1 nvarchar(max) ',@0=10,@1=N'Batching Example'
```

Figure 59: Inserting a record with a manually assigned key

If you are inserting a large number of entities, this can make a tremendous difference. Of course, sometimes it may not be appropriate to use manual insertions due to concurrent accesses, but in that case it probably can be solved by using Guids as primary keys.

> ***Note: Keep in mind that, generally speaking, ORMs are not appropriate for batch insertions.***

Use SQL Where Appropriate

If you have looked at the SQL generated for some queries, and if you know your SQL, you can probably tell that it is far from optimized. That is because EF uses a generic algorithm for constructing SQL that picks up automatically the parameters from the specified query and puts it all blindly together. Of course, understanding what we want and how the database is designed, we may find a better way to achieve the same purpose.

When you are absolutely certain that you can write your SQL in a better way than what EF can, feel free to experiment with SqlQuery and compare the response times. You should repeat the tests a number of times, because other factors may affect the results, such as other accesses to the database, the Visual Studio debugger, number of processes running in the test machines, all of these can cause impact.

One thing that definitely has better performance is batch deletes or updates. Always do them with SQL instead of loading the entities, changing/deleting them one by one, and them saving the changes. Use the ExecuteSqlCommand method for that purpose.

Chapter 12 Common Pitfalls

Overview

Whenever you start using a new technology, chances are there's always the possibility that you will fall into one of its traps. Here I will list some of them.

Changes Are Not Sent to the Database Unless SaveChanges Is Called

This is probably obvious, but still people sometimes forget about this.

LINQ Queries over Unmapped Properties

Visual Studio intellisense makes it very easy to write LINQ queries because it automatically shows all available properties. It may happen that some of these properties are not actually mapped. For example, they are read-only calculated columns that are computed from other properties. Accesses to these properties cannot be translated to SQL queries, so any attempt to use them in an Entity Framework query will result in an exception.

Null Navigation Properties

If in a loaded entity you have a null navigation property, this is probably due to one of these:

- You have disabled lazy loading globally by setting either LazyLoadingEnabled or ProxyCreationEnabled to false.
- The navigation property or its containing class does not qualify for lazy loading (either the class is sealed or the property is not marked as virtual).

For more information, revisit the *Lazy, Explicit and Eager Loading* section of *0*.

Validation Does Not Load References

Suppose you have a validation rule in your entity that checks something on a reference property when the entity is about to be persisted. Reference properties, provided lazy loading is enabled, are automatically loaded when accessed. However, while EFCF is performing validation, this won't happen, and references will be null. Either make sure you load all required properties before you call SaveChanges or explicitly force its loading on the validation method.

Concrete Table Inheritance and Identity Keys

When applying the Concrete Table Inheritance pattern, you cannot use IDENTITY for generating keys. This is described in *Inheritance Strategies*.

Cannot Return Complex Types from SQL Queries

When using SQL for querying your model, you cannot return entities that have complex types. This is a known limitation, and the only workaround is to get all of the entity's values as an object array and then manually create an entity from these values.

Cannot Have Non Nullable Columns in Single Table Inheritance

When using the Single Table Inheritance pattern, you cannot have non-nullable properties in derived classes, because all will be created in the same table, and they won't exist for all derived classes.

Deleting Detached Entities with Required References Doesn't Work

In *0 Chapter 5 Writing Data to* the Database, I presented a simple way to delete entities from the database without loading them first. It just happens that this won't work if your entity has any required references. If that is the case, you need to either load the entity or attach it to the context and load that reference before deleting it.

Attempting Lazy Loading of Navigation Properties in Detached Entities

If you try to load a navigation property in a detached entity, such as one retrieved from the ASP.NET session, the access will throw an exception because the originating context has been disposed. You will need to load all required navigation properties prior to storing them, or attach the entity to a new context. See *Lazy, Explicit and Eager Loading* for more information.

SELECT N + 1

This is by far the most typical problem when it comes to performance. What happens is: you issue a query, and then for each returned record, as you navigate them, you access some navigation property and another query is executed. This can be easily prevented by explicitly including the required navigation properties on the original query.

Appendix A Working with Other Databases

Entity Framework is agnostic regarding databases, which means it can potentially be used with any database for which there is an ADO.NET provider. Having said that, the harsh truth is, it can get really cumbersome to try to access databases other than SQL Server. The major problems are:

- The only supported primary key generation strategies are IDENTITY and manual, and even IDENTITY doesn't work in other database that offer similar functionality, such as MySQL and Oracle 12c; sequences are also not supported.
- Entity Framework's SQL generation engine sometimes produces SQL fragments that are specific to SQL Server. One of the most notorious is CROSS and OUTER APPLY, which won't even work in versions of SQL Server prior to 2005.
- EFCF assumes a database schema of dbo, which doesn't exist in other databases, so we are explicitly forced to specify a schema in each entities' mapping.
- Some .NET types are not supported by some other databases, such as enumerated types, Guid, **DbGeometry**, and **DbGeography**.
- The database data types, even if conceptually identical, have different names. For example, a variable length Unicode string in SQL Server is called NVARCHAR, where in Oracle it is called VARCHAR2. Be careful when you need to specify it.
- Some equivalent types are slightly different. For example, a DateTime property can be translated to a DATETIME column in SQL Server, which has both a date and a time part, but when translated to a DATE column in Oracle, it will only have the date part. Another example, DateTimeOffset even has equivalents in Oracle and others, but not on SQL Server 2005.
- The ROWVERSION type implied by the TimestampAttribute for concurrency checks, or better, the way it works, also only exists in the SQL Server family.
- There may be different accuracies for floating-point or decimal types.
- Last but not least, don't expect database generation to work: it won't, meaning you're on your own.

The list could go on, but having stated some of the problems, it is certainly possible to use Entity Framework to target other databases. Having the same code base for that almost certainly won't work. I'll just leave you with some guidelines for cross-database projects:

- Do not use database initializers, always generate the database yourself.
- Do use only "safe" base types, such as strings, numbers, byte arrays, and date/times.
- Do specify the physical column name for each property.
- Do specify the physical name and schema for each entity.
- Do use manually assigned identifiers.
- Do not specify a database type name for a property.

For an up-to-date list of third-party Entity Framework providers, check out: http://msdn.microsoft.com/en-us/data/dd363565.aspx.

Appendix B Additional References

ADO.NET Data Providers - http://msdn.microsoft.com/en-us/data/dd363565.aspx

ASP.NET Dynamic Data Overview - http://msdn.microsoft.com/en-us/library/ee845452.aspx

Entity Framework Automatic Code First Migrations - http://msdn.microsoft.com/en-us/data/jj554735.aspx

Entity Framework Code First Migrations - http://msdn.microsoft.com/en-us/data/jj591621.aspx

Entity Framework Feature Suggestions - http://data.uservoice.com/forums/72025-ado-net-entity-framework-ef-feature-suggestions

Entity Framework Home - https://entityframework.codeplex.com/

Entity Framework Included Conventions - http://msdn.microsoft.com/en-us/library/system.data.entity.modelconfiguration.conventions

Entity Framework Optimistic Concurrency Patterns - http://msdn.microsoft.com/en-us/data/jj592904.aspx

Entity Framework Power Tools - http://visualstudiogallery.msdn.microsoft.com/72a60b14-1581-4b9b-89f2-846072eff19d

Entity Framework Provider Support for Spatial Types - http://msdn.microsoft.com/en-us/data/dn194325.aspx

Entity SQL Language - http://msdn.microsoft.com/en-us/library/bb399560.aspx

How Entity SQL Differs from Transact-SQL - http://msdn.microsoft.com/en-us/library/bb738573.aspx

MiniProfiler - http://miniprofiler.com/

NuGet Gallery – Entity Framework - http://www.nuget.org/packages/EntityFramework

NuGet Gallery – MiniProfiler - http://www.nuget.org/packages/MiniProfiler

NuGet Gallery – MiniProfiler.EF - http://www.nuget.org/packages/MiniProfiler.EF/

NuGet Gallery – MiniProfiler.MVC3 - http://www.nuget.org/packages/MiniProfiler.MVC3/

SQL Server Profiler - http://technet.microsoft.com/en-us/library/ms181091.aspx

###

<div align="center">

<u>Copy Editor Checklist</u>

</div>

Writing style and mechanics

- Correct styles are applied to body text, headers, notes, figures, etc.
- All spellings adhere to American English.
- Above and below have been revised in references to figures, tables, chapters, sections, or code examples.
- All emoticons have been removed.
- All excessive punctuation has been removed.
- All chapters are numbered.

Figures and tables

- All figures have captions and numbers.
- Figure annotations do not contain spelling errors.
- All tables have captions.
- Tables that break across pages have repeating headers.
- All code tables are formatted correctly.

Notes and tips

- All notes say Note and have the orange notepad image.
- All tips say Tip and have the orange light bulb image.