Once you have retrieved entities you can modify them, delete them,

or even add new ones and then save all of these changes back to the database. In this

chapter, we’ll take a high-level look at the way in which the Entity Framework is able

to track these changes and get the necessary data back to the database.

Keeping Track of Entities

Entity Framework takes a snapshot of an entity’s values when the ObjectContext first

becomes aware of the entity. This will happen by default when query results are being

materialized into objects. The context stores two sets of these values. The first set represents

the original values and remains static. The second set represents the entity’s

current values, and these will change in response to edits being performed to the entity

properties.

Managing an Entity’s State

By default, as each entity is materialized from the query results, the ObjectContext

creates an extra object behind the scenes, called an ObjectStateEntry. This is where the

snapshot—that is, the two copies of the object’s values—is stored.

The ObjectContext can track only entities. It cannot keep track of anonymous types or

nonentity data that is returned in a DbDataRecord.

ObjectStateEntry also has a State property whose value reflects the state of the entity

(Unchanged, Modified, Added, or Deleted). As the user modifies the objects, the Object

Context updates the current values of the related ObjectStateEntry as well as its

State. As you learn more about the Entity Framework, you’ll discover how to locate

and inspect the details of an ObjectStateEntry.

ObjectStateEntry

keeps track of only the scalar values (including those inside the complex properties) of

its related entity.

As the scalar properties are changed—for example, Contact.LastName—the new value

of LastName is stored in the ObjectStateEntry’s set of current values for that contact,

and if the ObjectStateEntry.State value was Unchanged at the time of the modification,

its value will be set to Modified.

Saving Changes Back to the Database

Example 6-1 shows a simple ObjectQuery to retrieve the first contact from the Contacts

EntitySet. Remember from Chapter 3 that context.Contacts is a method that will return

an ObjectSet of Contact types. The example then uses the LINQ extension method

First to pull back only the first result.

The FirstName and ModifiedDate properties are given new values, and then

SaveChanges is called.

*Example 6-1. Querying for a contact, editing, and then saving back to the database*

using (PEF context = new PEF())

{

var contact = context.Contacts.First();

contact.FirstName = "Julia";

contact.ModifiedDate = DateTime.Now;

context.SaveChanges();

}

exec sp\_executesql N'update [dbo].[Contact]

set [FirstName] = @0, [ModifiedDate] = @1

where ([ContactID] = @2)

',N'@0 nvarchar(50),@1 datetime2(7),@2 int',@0=N'Julia',

@1='2009-11-30 09:27:20.3335098',@2=1

In our sample, the FirstName and

ModifiedDate properties had changed, and therefore those are the only values that it

sends into the command. It uses the value of the property that is marked as the

EntityKey, ContactID, to identify which row to update.

Example 6-2 queries for all contacts named Robert, along with their addresses, then

returns a List of the entity graphs: Contacts with Addresses. The example then randomly

selects one of these contacts and changes its FirstName to Bobby. Another contact

is selected and the Street property of the first Address is edited. Finally, SaveChanges is

called.

*Example 6-2. Editing various entities and calling SaveChanges*

var contacts = context.Contacts.Include("Addresses")

.Where(c =>c.FirstName=="Robert").ToList();

var contact = contacts[3];

contact.FirstName = "Bobby";

contact = contacts[5];

var address = contact.Addresses.ToList()[0];

address.Street1 = "One Main Street";

context.SaveChanges();

Inserting New Objects

In Example 6-3, a new address is created in memory. Rather than use

Address.CreateAddress, this code instantiates a new Address directly, because even

if I had used the factory method, I still would have to set all of the string scalars. Then,

after attaching the address to a contact that was queried from the database,

SaveChanges is called.

There are many different ways to link entities to one another based on

particular scenarios. You will learn about this in Chapter 19.

*Example 6-3. Creating a new address in memory*

var contact = context.Contacts.Where(c => c.FirstName == "Robert").First();

var address = new Address();

address.Street1 = "One Main Street";

address.City = "Burlington";

address.StateProvince = "VT";

address.AddressType = "Business";

address.ModifiedDate = DateTime.Now;

//join the new address to the contact

address.Contact = contact;

context.SaveChanges();

Inserting New Parents and Children

The preceding example inserted a new address to an existing contact. What if you

wanted to create a new contact with a new address? In typical data access scenarios,

you would have to first insert the new contact, retrieve its ContactID, and then use that

to insert the new address. SaveChanges does all of this for you when it sees that both

are new and that they are related. It also uses the model’s mappings to figure out which

is the dependent entity (in this case, Address) and needs the foreign key (ContactID).

With this information, it executes the database inserts in the correct order.

*Example 6-4. Inserting a new contact with a new address*

var contact = Contact.CreateContact

(0, "Camey", "Combs", DateTime.Now, DateTime.Now);

var address = new Address();

address.Street1 = "One Main Street";

address.City = "Olympia";

address.StateProvince = "WA";

address.AddressType = "Business";

address.ModifiedDate = DateTime.Now;

//join the new address to the contact

address.Contact = contact;

//add the new graph to the context

context.Contacts.AddObject(contact);

context.SaveChanges();

Deleting Entities

ObjectContext has a DeleteObject method that takes

an EntityObject as a parameter—for example, an instance of a Contact. When

DeleteObject is called, the context sets the State of that object’s ObjectStateEntry to

Deleted. To be explicit, it does not delete the entity, but marks it as “to be deleted from

the database. ” When SaveChanges is called, the context notes the Deleted State and

constructs a Delete command to send to the database.

If the entity has already been retrieved from the database, this will not pose a problem.

But sometimes you might want to delete data from the database that has not been

queried.

*Example 6-5. Retrieving and deleting a contact entity*

System.Data.EntityKey contactKey =

new System.Data.EntityKey("PEF.Contacts", "ContactID", 438);

var contact = context.GetObjectByKey(contactKey);

context.DeleteObject(contact);

context.SaveChanges();

The sample database has a constraint defined for the Address table’s ContactID column,

called a *cascading delete*. This tells the database that when the contact with the matching

ContactID is deleted from the Contacts table, it should delete any Addresses that have

the same ContactID value.

Using Stored Procedures with the EDM

Many databases use stored procedures to perform predefined logic on database tables,

and many organizations have policies in place that require the use of these stored procedures.

Although one of the key features of the Entity Framework is its ability to

automatically build native commands based on your LINQ to Entities or Entity SQL

queries, as well as build the commands for inserting, updating, or deleting data, you

may want to override these steps and use your own predefined stored procedures. Although

the dynamically built commands are secure, efficient, and generally as good as

or better than those you may write yourself, there are many cases where stored procedures

already exist and your company practices may restrict direct use of the tables.

Alternatively, you may just want to have explicit control over what is executed on the

store and prefer to create stored procedures.

The sample database includes six stored procedures that we skipped in our discussion

of model creation in Chapter 2. In this chapter, you will update the model, pulling in

those six stored procedures, implementing them in the model, and interacting with

them in some code.

**Pourquoi utiliser les procédures stockées plutôt qu’une commande T-SQL ?**

Publié le [**3 août 2009**](http://blog.developpez.com/elsuket/p7312/moteur-de-base-de-donnees-sql-server/pourquoi_utiliser_les_procedures_stockee)

Lorsqu’une application doit exécuter une requête dans une base de données, il est préférable que celle-ci appelle une procédure stockée plutôt que d’envoyer une commande T-SQL construite dans le code de l’application.  
Outre l’avantage de sécurité contre les attaques par injection et la maintenabilité du code, voici les autres avantages que cela procure :

- Dès lors qu’on appelle une procédure stockée, on a simplement besoin de spécifier son nom et de lui adjoindre la valeur de ses paramètres.  
C’est donc moins coûteux en termes de quantité de données qu’une commande T-SQL complète à envoyer à l’instance SQL Server, si courte soit elle.  
Dès lors cela réduit le trafic réseau entre les applications et le serveur.

- Lorsqu’on crée une procédure stockée, celle-ci est compilée en un plan qui demeure dans le cache de procédures, ce qui réduit considérablement le coût de calcul du plan d’une requête, gourmand en ressources processeur.

- L’exécution de requêtes par l’instance elle-même peut s’avérer moins coûteuse pour l’application mais aussi pour le serveur de bases de données.  
En effet, si nous devons par exemple insérer une valeur binaire dans une table à travers une requête soumise directement par l’application, il faut dans un premier temps convertir la valeur binaire en une chaîne de caractères, ce qui double donc sa taille.  
Lorsque l’instance SQL Server reçoit la requête, elle doit reconvertir la valeur chaîne de caractère vers la valeur binaire, ce qui représente pour les deux partis un surcoût.  
L’appel d’une procédure stockée élimine ce problème, puisque les valeurs des paramètres sont conservées sous forme binaire depuis l’application jusque dans les pages de la table de la base de données.

- L’utilisation de procédure stockées permet la réutilisation de code. S’il est clair que cela n’augmente pas les performances, cela augmente la productivité des développeurs qui ont moins de code à produire, et qui passent donc moins de temps à le débugger.

- Le second plus gros avantage à mon sens après le cache de procédures est que le code qu’elles encapsulent, la logique de celles-ci, peut être maintenu sans modifier une seule ligne de code de l’application cliente, s’il n’est pas besoin de modifier les paramètres et la structure de l’ensemble de données produit.

En conclusion, il faut garder à l’esprit que ce n’est pas parce que l’on utilise exclusivement des procédures stockées que l’application sera performante.  
Il reste encore parfois à réécrire et souvent à optimiser le code qui est encapsulé par une procédure stockée, pour que celui-ci soit performant et qu’il reste le plus longtemps possible dans le cache de procédures.

Updating the Model from a Database

The EDM tools provide a feature called Update Model from Database, which is available

from the Designer context menu. You can use it to add previously skipped database

objects or those that have been added to the database since the time you originally

created the model. Update Model from Database can also recognize new fields added

to tables that have already been mapped in the database.



Click Finish to add the stored procedures to the model. When the update is complete,

the model will not look any different when viewed in the Designer. Stored procedures

are not automatically added to the conceptual layer of the model. Instead, they have

been represented in the SSDL as function elements. It will be your job to define how

these functions should be implemented in the conceptual model using mapping.

Working with Functions

Each of these six functions represents a different stored procedure in the database. The

first three return query results. The last three—the Insert, Update, and Delete

procedures—perform the changes you would expect to the database.

Voir le fichier SSDL

Mapping Functions to Entities

You can override this behavior for specific entities by using the SSDL functions (based

on the database stored procedures) instead. You can map these functions to specific

entities. Then, when SaveChanges is called, the Entity Framework will use the designated

stored procedures rather than generate commands. For entities that have no function

mappings, the Entity Framework will perform the default behavior of generating the

commands dynamically.

The single mapping rule that remains in place is that every input parameter of a function

must match a property in the entity. You can’t substitute your own data to use as an

input parameter. You only can use one of the entity’s properties.

Attention :

There is a known behavior with respect to the mapping function feature.

If you map an update function but no delete function, you will get an

error when attempting to delete these entities. Therefore, even though

the schema does not require that you map both, if you want users to be

able to delete a particular entity type and you are mapping its update

function, you should also map its delete function.

Protecting Tables by Using Views and Stored Procedures

There’s another great benefit to mapping the Insert, Update, and Delete functions, and

that is security.

If you are reluctant to expose your database tables for querying, you don’t have to.

Earlier in this book, I discussed database views in the model. Views come into the model

as entities, but because views are read-only, Entity Framework is not able to construct

commands to persist data back to the database when you call SaveChanges. That makes

sense because you don’t persist back to the views; you need to send the data to tables.

However, these entities still participate in change tracking just like any other entities

(with a caveat about EntityKeys that I’ll discuss momentarily). You can then map stored

procedures to these view-based entities in order to persist their data when

SaveChanges is called. This gives you a complete round trip to query and update data

without exposing your database tables.

Mapping Insert, Update, and Delete Functions to an Entity

Attention , fonction et procédure stockés sont souvent utilisées comme synonymes

If you look more closely at the Mapping Details window, you will notice two icons in

the upper-left corner. Select the Contact entity in the Designer to display its mappings.

The icons will become active. Clicking the top icon causes the Mapping Details window

to display the table mappings. The lower icon is for displaying function, a.k.a. stored

procedure, mappings. You can also display function mappings by right-clicking an

entity and choosing Stored Procedure Mapping.

In the Mapping Details window, you will see three placeholders for selecting an

Insert function, an Update function, and a Delete function, as shown in Figure 7-2.





You may recall from Chapter 6 that when the Entity Framework constructs its own

Insert command, it selects the new identity value and automatically pushes it into the

entity object that was inserted. You can achieve the same effect by mapping the returned

NewContactID value directly to the entity’s ContactID property. That will mean it will

not be necessary to requery the database to acquire the ContactID for an inserted

contact.

To map the returned value, type **NewContactID** over the text “<Add ResultBinding>”.

The ContactID will be automatically chosen as the property to map to because it is the

EntityKey for Contact, and therefore it is a very good first guess for the Designer to make

for you. Output parameters are supported, but not for EntityKey properties. See the

note about this in the following section, “Concurrency checking with Use Original

Value and Rows Affected Parameter options”.



Concurrency checking with Use Original Value and Rows Affected Parameter options