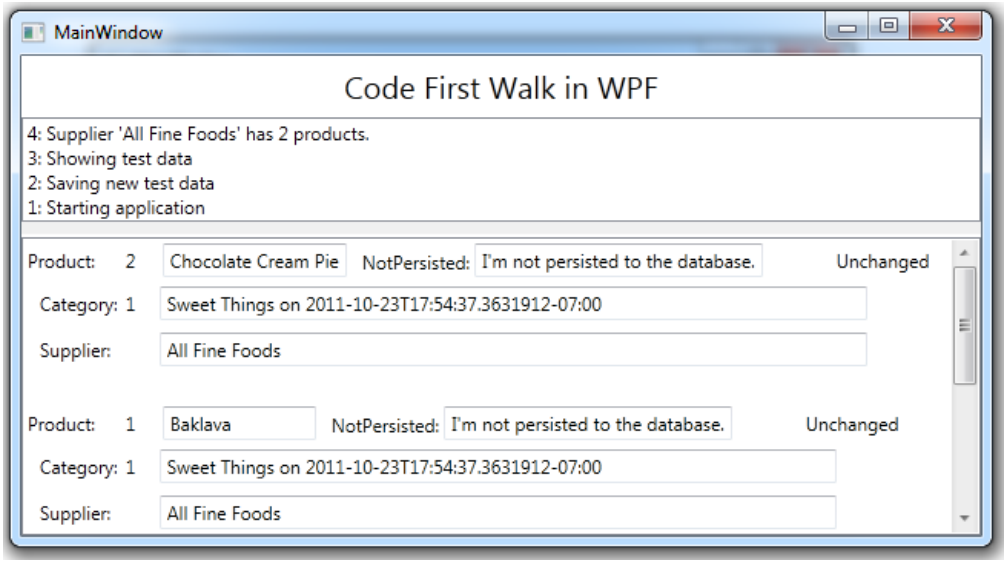
Contents

When the Entity Framework naming conventions don't convey the database schema you need, use Entity Framework mapping configuration. We add another new type (*Supplier*) and map declaratively with attributes. Then we create our first EF *DbContext* class and map imperatively with the Fluent API.



- **Platform:** WPF
- **Language:** C#
- **Download:** Code First Walkthrough

Add a Supplier Entity

1. **Add reference** to *System.ComponentModel.DataAnnotations* if it's not in your project already.

2. **Paste** the following at the bottom of the Model class file:

```
[ProvideEntityAspect]
public class Supplier
{
  public Guid SupplierKey { get; set; }
  public string CompanyName { get; set; }
}
```

3. **Add the Suppliers EntityQuery** to the *ProductEntities* entity manager.

```
public EntityQuery<Supplier> Suppliers { get; set; }
```

We've done this before for our other entity classes to make it easier to query for them.

Map and initialize the *SupplierKey*

1. **Return** to the *Supplier* class.

The *SupplierKey* property is the entity key; it has two unusual aspects:

1. its unconventional name
2. its *Guid* return type

The name, "SupplierKey", does not conform to Entity Framework Code First naming conventions. We must configure this critical key mapping explicitly.

We could use the Code First Fluent API to configure the mapping imperatively (as we will later). Instead, for now, we'll take a declarative approach and apply mapping attributes. We'll mark the *SupplierKey* property with the [Key] attribute:

```
[Key] // Key for "Supplier" does not conform to conventions
public Guid SupplierKey { get; set; }
```

The *[Key]* attribute requires "*using System.ComponentModel.DataAnnotations;*".

The key's *Guid* type presents a different challenge. Entity Framework maps integer keys to "Identity" columns by default; that means EF expects the database to generate the key value when new entities are inserted.

Entity Framework treats *Guid* key properties differently; it assumes that the application will assign their values. Many developers assign the key in the entity constructor; we will follow their lead by adding one here.

```
public Supplier()
{
    SupplierKey = Guid.NewGuid();
}
```

Entity Framework can generate *Guid* keys if you wish. Learn more in the topic on Guid keys.

Set the Table Mapping

If you happened to study the database that Entity Framework generated, you perhaps noticed that the tables corresponding to the entities have plural names: "Categories" and "Products". That is EF's Code First convention.

We might prefer that the table names be singular like their corresponding entity class names. We can tell Code First to expect a singular table name with the *System.ComponentModel.DataAnnotations.**Table*** mapping attribute, decorating the class as follows:

```
[Table("Supplier")]
public class Supplier { ... }
```

Associate Suppliers and Products

Suppliers will have Products and each Product will have a parent Supplier.

Products have multiple Suppliers in the real world. We could model that with a ProductSupplier association table and perhaps a many-to-many relationship between the two classes. That's an exercise for the future. At the moment, we'll design for a one-to-many relationship between Supplier and its Products.

1. **Add SupplierKey and Supplier to the Product class.**

2. **Go** to the Product class and add the following *SupplierKey* and *Supplier* properties which are just like *CategoryId* and *Category*.

```
public Guid SupplierKey { get; set; } // Foreign key for "Supplier"
public Supplier Supplier { get; set; }
```

DevForce and Entity Framework can tell that *SupplierKey* holds the foreign key value uniting *Product* and *Supplier* because it has the same name as the Supplier's key property. If you wanted to be explicit about the role of the *SupplierKey* property (as you'd have to be if you gave it a different name), you could decorate it with the *ForeignKey* attribute:

```
[ForeignKey("Supplier")] // Specifies the corresponding navigation property
public Guid SupplierKey { get; set; } // Foreign key for "Supplier"
public Supplier Supplier { get; set; }
```

3. **Add *Products* property to the *Supplier* class**
Go back to Supplier and add the following Products property.

```
public RelatedEntityList<Product> Products { get { return null; } }
```

The return type must be *RelatedEntityList<T>*, a derivative of *ICollection<T>* as required by EF Code First. We've implemented the getter but not the setter because we don't want anyone to set the *Products* collection. Add and remove products, yes, but not change the collection itself. That's a DevForce job.

The *Supplier* class at this point is as follows:

```
[ProvideEntityAspect]
[Table("Supplier")]
public class Supplier
{
  public Supplier()
   {
     SupplierKey = Guid.NewGuid();
   }
   [Key] // Key for "Supplier" does not conform to conventions
  public Guid SupplierKey { get; set; }
  public string CompanyName { get; set; }
  public RelatedEntityList<Product> Products { get { return null; } }
}
```

Add an unmapped property

Your business logic may require entity properties that do retrieve or store data in the database. Calculation fields are typical but you might have a settable property as well. Let's add an example to the *Product* class.

```
// Example of non-persisted property
[NotMapped]
public string NotPersisted
{
  get { return _notPersisted; }
  set { _notPersisted = value; }
}
private string _notPersisted = "I'm not persisted to the database.";
```

Add a custom *DbContext* Class

Those of you who are familiar with Entity Framework Code First may be surprised that we have not even mentioned EF's *DbContext* class. EF Code First requires a *DbContext*.

DevForce has been using a *DbContext* all along. It's been using its own *DbContext*. DevForce will use your *DbContext* instead … if you write one.

Most developers write a *DbContext* in order to configure entity-database mapping imperatively using the Code First Fluent API. We've done quite well with just the mapping attributes. But some people don't like attributes and some mappings (e.g., certain forms of inheritance) can only be defined through the Fluent API.

Another reason to write a *DbContext* is to control how and when the Entity Framework creates and initializes the database … or to stop it from ever creating a database.

This is why we will write a *DbContext*. We're tired of running our application, watching it crash because the model changed, deleting the database, and running again. In these early stages of development, we are happy to let Entity Framework detect the mismatch and re-create the database for us automatically.

1. **Add reference to System.Data.Entity** if not referenced already. *DbContext* depends upon this assembly.
2. **Add | New Item** (Ctrl-Shift-A) **| Class**
3. Call it "*ProductDbContext*"
4. **Inherit from *DbContext***. Leave the class *internal*; no one will call it except DevForce.

```
class ProductDbContext : DbContext
```

5. **Add** "*using System.Data.Entity;*"
6. **Add a constructor with a string parameter** as follows:

```
public ProductDbContext(string connection) : base(connection)
```

```
{
  // Do not use in production; for early development only
  Database.SetInitializer(
     new DropCreateDatabaseIfModelChanges<ProductDbContext>());
}
```

Notice that the constructor takes a string. When DevForce creates an instance of your *DbContext*, it supplies either a database connection string or the name of a connection string in the assembly's configuration file.

Always add a constructor that takes a string parameter.

Notice the *DropCreateDatabaseIfModelChanges<T>* object passed in the static *Database.SetInitializer* method call. That's an initialization strategy. We're asking Entity Framework to create the database if it doesn't exist or (b) drop and recreate it according to the revised model if the model has changed. With a little more work we could write a strategy that seeds the database with test data as well.

Pass null into the SetInitializer to prevent EF from creating the database … ever. That's the correct call for a production release or when accessing a database with irreplaceable data or schema.

Configure to ignore *EntityAspect*

All DevForce AOP entities have an *EntityAspect* property through which you gain access to your entity's internal entity capabilities. We haven't written such a property in any of our entity source code, but it's there after DevForce rewrites each entity class with its DevForce infrastructure.

Entity Framework assumes that the *EntityAspect* object returned by the property is an entity type … which it is **not**. EF model validation will fail unless someone tells EF to ignore the *EntityAspect* type and all properties that return that type.

The DevForce default *DbContext* does that for us. But when we write our own *DbContext* - as we are doing now -, **we** have to tell EF to ignore it, using the Entity Framework Code First Fluent API to which we gain access by overriding *DbContext*'s *OnModelCreating* method.

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
   modelBuilder.Ignore<EntityAspect>();
}
```

Add "*using IdeaBlade.EntityModel;*"

Add *DbSet* Properties for roots

We have to tell Entity Framework which entity types are in our model. We do that by specifying a *DbSet<T>* property where "*T*" is an entity type. We only need one such property for this model:

```
public DbSet<Supplier> Suppliers { get; set; } // root entity for model discovery
```

Supplier is a "root entity", meaning that one can walk an "object graph" from this entity to other entities by following navigation properties recursively. ***Supplier***.*Products* returns ***Product*** entities. *Product.Category* returns a ***Category*** entity. Those are the three entities in our model and *Supplier* is a root to all of them.

Developers with previous Entity Framework Code First experience tend at first to write a *DbSet* property for all entity types just as we add *EntityQuery<T>* properties to the *ProductDbEntities* entity manager. They do so to make it easier to access model types from the *DbContext*. But a DevForce developer rarely (if ever) uses this *DbContext* so that kind of effort is usually a waste of time.

Add *[DataSourceKeyName]* attribute

DevForce no longer uses the name of your *EntityManager*, "*ProductEntities*", as the name for finding the database connection string or for naming the database it creates. Instead DevForce now uses the name of your *DbContext*, "*ProductDbContext*".

If you could run the application now (which you can't quite do), you'd see Entity Framework create a database named "ProductDbContext".

"ProductDbContext" is not a good name for the database, even in a demo. "ProductEntities" wasn't a great name either. Neither of them is a wonderful name for the connection string in a configuration file.

Rather than be at the mercy of potentially changing class names, it is best to specify the connection string name explicitly. In DevForce, this name is called the *DataSourceKeyName* and you specify it by decorating the *DbContext* class with the *DataSourceKeyNameAttribute*. For our example, we'll name it "CodeFirstWalk".

```
[DataSourceKeyName("CodeFirstWalk")]
class ProductDbContext : DbContext {...}
```

We could have added this attribute to the entity manager class, *ProductEntities*, and had the same effect. We saved the attribute for our *DbContext* because the name (or attribute) of the *DbContext* always trumps the name (or attribute) of the *EntityManager*.

Re-configure *Supplier* mapping with fluent API (optional)

The author of this sample does not want to see database specifics in the entity model classes. He dislikes the [*Table*] and [*Column*] attributes in particular. If you are not similarly troubled by it, you can skip this section or read-and-ignore it.

The Entity Framework naming conventions translate a singular entity class name to a plural table name. We prefer singular table names. Recall that we applied [*Table*] to the *Supplier* class so that our model would conform to our preferred table naming style:

```
[Table("Supplier")]
public class Supplier
```

We can achieve that goal through the fluent interface.

1. **Delete that *Table* attribute** from the *Supplier* class definition in the Model file.

2. **Add imperative Table configuration** to *ProductDbContext*'s *OnModelCreating* method.

```
modelBuilder.Entity<Supplier>().ToTable("Supplier");
```

It seems that other people don't like the Entity Framework plural table name convention either. This is the one convention you can change in the current EF Code First release. Let's remove that convention so that all of our table names are singular.

3. **Replace the "*ToTable()*" line we just added with**

```
modelBuilder.Conventions.Remove<PluralizingTableNameConvention>();
```

The *PluralizingTableNameConvention* type requires "*using System.Data.Entity.ModelConfiguration.Conventions;*".

The revised *ProductDbContext* follows:

```
[DataSourceKeyName("CodeFirstWalk")]
class ProductDbContext : DbContext
{
  public ProductDbContext(string connection) : base(connection)
   {
     // Do not use in production; for early development only
     Database.SetInitializer(
        new DropCreateDatabaseIfModelChanges<ProductDbContext>());
   }
  protected override void OnModelCreating(DbModelBuilder modelBuilder)
   {
     modelBuilder.Conventions.Remove<PluralizingTableNameConvention>();
     modelBuilder.Ignore<EntityAspect>();
   }
  public DbSet<Supplier> Suppliers { get; set; } // root entity for model discovery
}
```

Clean and build

Build the model (Ctrl-Shift-B). We recommend that you **always build the model after making a change to the model because it is usually easier to catch and repair a modeling error immediately rather than discover and figure it out later.**

You may notice in the project inventory that the old *ProductEntities.ibmmx* metadata file associated with the *ProductEntities EntityManager* has disappeared and been replaced by a new metadata *CodeFirstWalk.ibmmx*, named to match the *DataSourceKeyName*.

Did you get a build error that mentions Microsoft SQL Server Express? See the Build Errors section below.

Update the UI for *Supplier*

We'll only use one Supplier in our demo system so our *MainWindowViewModel* revisions will be a little different from the way we've done them before.

1. **Add** to the top of the *AddTestData()* method …

```
private void AddTestData()
{
    GetOrAddSupplier();
   // ...
}
```

2. **Add** the ***GetOrAddSupplier*** **method** and ***CurrentSupplier*** **property**:

```
/// <summary> Get first supplier, if exists, or make one</summary>
private void GetOrAddSupplier()
{
    CurrentSupplier = Manager.Suppliers.FirstOrDefault();
   if (null != CurrentSupplier) return;
    CurrentSupplier = new Supplier { CompanyName = "All Fine Foods" };
    Manager.AddEntity(CurrentSupplier);
}
private Supplier CurrentSupplier { get; set; }
```

3. If there's a *Supplier* in the database, that's our current supplier; if not, we make a new one and add it to the *Manager* for subsequent save with the new *Category* and new *Products* that we're already making in *AddTestData*.

4. **Assign the new *Product's Supplier*** in the **AddNewProduct** method by adding another initialization. The revised method is as follows:

```
private Product AddNewProduct(string productName = "A new product")
{
    var newProduct = new Product
    {
        ProductName = productName,
        Category = CurrentCategory,
        Supplier = CurrentSupplier,
    };
   //Manager.AddEntity(newProduct); // harmless but unnecessary
   return newProduct;
}
```

We don't have to add the new *Product* to the Manager because both the *CurrentCategory* and the *CurrentSupplier* are in the *Manager*'s cache already and will pull the new *Product* into the cache upon assignment.

Prove that *Supplier.Products* works

The *MainWindow.xaml* is already data bound to the *Product*'s supplier (we got ahead of ourselves). That binding shows how to navigate from *Product* to *Supplier*. We should demonstrate the *Supplier.Products* navigation property works in the opposite direction by querying the count of a *Supplier* entity's products and logging the count in the view.

1. **Go** to the *ShowTestData* method.

2. **Add** to the bottom of the *ShowTestData* method:

```
LogCurrentSupplierProductsCount();
```

3. Implement that method:

```
private void LogCurrentSupplierProductsCount()
{
    Log(
        String.Format("Supplier '{0}' has {1} products.",
        CurrentSupplier.CompanyName, CurrentSupplier.Products.Count));
}
```

The *CurrentSupplier.Products* navigation property returns a *RelatedEntityList<Product>* whose *Count* method causes DevForce to query for related products.

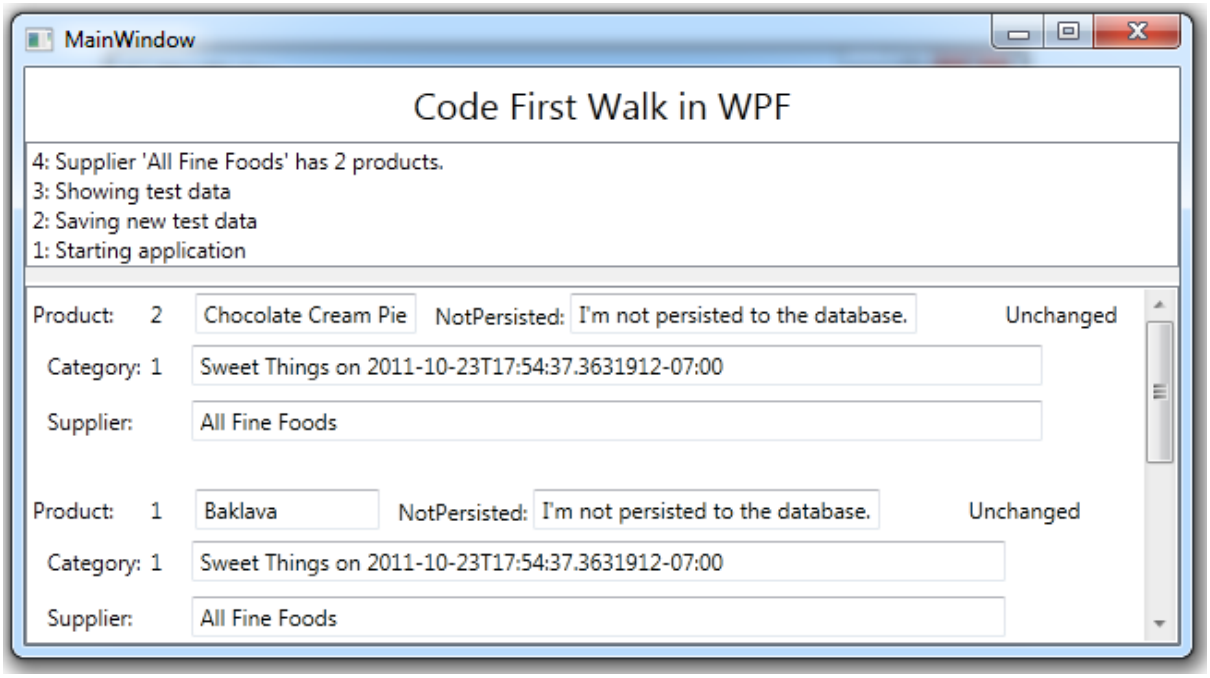
You may recall that we prevented further querying of the database when we set the *Manager*'s *DefaultQueryStrategy* to *CacheOnly*.

```
// DEMO ONLY – DO NOT DO THIS IN YOUR CODE
Manager.DefaultQueryStrategy = QueryStrategy.CacheOnly; // let's see only what's in cache
```

Fortunately, *ShowTestData* already fetches every product in the database; by the time we look for them, the *CurrentSupplier*'s products will be in the local cache.

Run in Debug [F5]

It should display a window such as this one:



Notice:

- The log presents the count of products from the *CurrentSupplier.Product* navigation property.
- The *NotPersisted* value appears on screen but is not a column the database.
- The *Product.Supplier* property returns the product's supplier

Summary

In this part of the "Code First Walk" we

- added the *Supplier* entity whose properties defied EF's naming conventions.
- consequently, we mapped *Supplier* explicitly with attributes and the fluent API.
- added a collection navigation property (*Supplier.Products*) that returns a *RelatedEntityList<T>*
- added a custom *DbContext*, having relied on the DevForce default *DbContext* until now.
- applied the *DataSourceKeyName* attribute to determine the connection string name and the name of the generated database
- now re-create the database whenever the model changes – a technique suitable only during early development.

Appendix: Common Build Errors

You got a build error such as the following:

> An error occurred **during metadata generation** and a metadata file could not be created. Error: A connectionString was not found. **Since SQLExpress is not installed DevForce metadata discovery cannot continue**. Add a .config file with a connectionString named 'CodeFirstWalk' to this project. Providing a connection string at build time ensures that the correct model metadata is generated.

Entity Framework, by default, expects you to have installed Microsoft **SQL Server Express**. You can get around that by specifying a connection string in a configuration file or change the default database server to **SQL Server** by setting the *DefaultConnectionFactory*. These options are discussed in the Advanced Database Connection Options topic of the DevForce Resource Center.

If you don't have SQL Server Express but do have the full SQL Server installed, try adding the following static constructor to the *ProductDbContext*. Remember to look closely at the parts of the *baseConnectionString* to ensure they match your SQL Server name.

```
[DataSourceKeyName("CodeFirstWalk")]
class ProductDbContext : DbContext
{
  static ProductDbContext()
   {
     // Set base connection string
     const string baseConnectionString =
     "Data Source=.; " +  // your SQL Server name
     "Integrated Security=True; " +
     "MultipleActiveResultSets=True; " +
     "Application Name=CodeFirstWalk"; // change to suit your app
      Database.DefaultConnectionFactory = new SqlConnectionFactory(baseConnectionString);
   }
  // elided.
}
```

If this sounds familiar, you may have done something just like it to your *ProductEntities* custom *EntityManager*. You **must now go remove that static constructor** so that it doesn't interfere with this one.