Contents

- You may not need a custom DbContext
- Write a custom DbContext
- Add the DataSourceKeyName attribute
- Add a constructor with a connection parameter
- Ignore the EntityAspect type
- Specify the DbSets
- Initialize the database (optional)
- Multiple DbContext classes
- No DbContext in Silverlight and mobile projects
- You write it, DevForce calls it
- Learn more

Add a custom Entity Framework DbContext to your model when mapping entities to database objects with EF's fluent interface or configuring other aspects of model behavior that are specific to EF.

Entity Framework uses a DbContext to respond to client queries and save entity data to the database. DevForce uses a *DbContext* at build time to find the metadata for your model and again at runtime in the *Entity Server* when querying and saving.

The typical Entity Framework developer writes a class that inherits from *DbContext*. Inside that custom class may be code to

- map entity classes to database objects using EF's Fluent API
- initialize a generated development database
- define the DbSets that perform create, read, update, and delete operations; EF uses the types discovered in these *DbSets* to identify the entity types within the model managed by this *DbContext*.

You may not need a custom *DbContext*

You may not need to write a custom *DbContext* if you've written a custom *EntityManager* ... as most people do. DevForce uses its own *DbContext* at runtime if you don't write one.

You won't need a custom *DbContext* if you've met all of your mapping needs for the entity classes you've written through a combination of naming conventions and mapping attributes and you don't need any other Entity Framework *DbContext* customizations.

You can have both a custom *EntityManager* and a custom *DbContext*. But you may not need your own *DbContext* just yet and it's easy to add one later.

Write a custom DbContext

You may decide to write a *DbContext* in order to map entities to the database using Entity Framework's Fluent API. With the Fluent API you can specify mappings that you can't define with data annotations alone. Some folks simply prefer the Fluent API. Control over database initialization is a popular second reason to write a custom *DbContext*.

A few rules and suggestions for writing your custom *DbContext*:

- Add a *DataSourceKeyName* attribute to the class
- Write a constructor with a connection parameter
- Always override the *OnModelCreating* method and tell EF to ignore the *EntityAspect* type.
- Add *DbSet* definitions for every root entity in your model.
- Consider adding mappings to the *OnModelCreating* method using the Fluent API.
- Consider adding database initialization code to your constructor.

These points are illustrated in the following example and then discussed in more detail below:

```
using System.Data.Entity;
using IdeaBlade.EntityModel;
namespace CodeFirstWalk
{
   [DataSourceKeyName("CodeFirstWalk")]
  class ProductDbContext : DbContext // leave it internal; no one calls it except DevForce
  {
     public ProductDbContext(string connection = null) : base(connection)
      {
        // Do not use in production; for early development only
        Database.SetInitializer(
           new DropCreateDatabaseIfModelChanges<ProductDbContext>());
      }
```

```
    protected override void OnModelCreating(DbModelBuilder modelBuilder)
     {
        modelBuilder.Ignore<EntityAspect>(); // Always ignore EntityAspect
       modelBuilder.Entity<Category>().ToTable("Category"); // otherwise EF assumes the table is called "Categories"
       modelBuilder.Entity<Product>().ToTable("Product"); // otherwise EF assumes the table is called "Products"
     }
    //public DbSet<Category> Categories { get; set; } // not necessary
    //public DbSet<Product> Products { get; set; } // not necessary
    public DbSet<Supplier> Suppliers { get; set; }
  }
}
```

You may need to add a project reference to *System.Data.Entity*.

Add the *DataSourceKeyName* attribute

Every model has a *DataSourceKeyName*. DevForce uses that name to find the associated database connection string. If DevForce and Entity Framework can't find a connection string with that name, Entity Framework typically creates a database with that name.

You should specify the *DataSourceKeyName* using the attribute.

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

If you don't apply the attribute, DevForce uses the name of your *DbContext* class (e.g., "ProductDbContext") as the *DataSourceKeyName*. The class name is rarely a good database or connection string name and you don't want that name changing every time you change the class name.

The *DataSourceKeyName* must not include spaces or underscores ("_").

Add a constructor with a connection parameter

We strongly recommend that you add a constructor that takes a string connection parameter as we did in the example:

```
[DataSourceKeyName("CodeFirstWalk")]
public class ProductDbContext : DbContext
{
  public ProductDbContext(string connection) : base(connection) { ... }
   ...
}
```

In fact, you **must** add such a constructor if you annotate the class with the *DataSourceKeyName* attribute.

Technically, you don't have to write a constructor at all. A *DbContext* can have an implicit parameterless constructor and you only have to write a constructor if you have special logic to run when an instance is created.

If you omit the constructor or if your constructor has no parameters, DevForce lets Entity Framework figure out how to find and connect to the database. While this works, it is rarely desirable.

When you provide a constructor with a string parameter, DevForce can build a string that contains database connection information and will pass that string into the constructor. We cover how DevForce builds that string elsewhere.

Ignore the *EntityAspect* type

All DevForce AOP entities have an *EntityAspect* property through which you gain access to your entity's internal entity capabilities. You may or may not write that property in your entity source code, but it's there after DevForce rewrites the class with DevForce entity infrastructure.

Entity Framework discovers this *EntityAspect* property as it investigates the types in your entity model. EF sees that the property returns an *EntityAspect* type which it assumes is an *entity* type. It's not an *entity* type and it fails validation as an entity type because it doesn't have a key. You'll get a build error message that says

> EntityType 'EntityAspect' has no key defined. Define the key for this EntityType.

Someone must tell EF to ignore the *EntityAspect* and all properties that return that type. The DevForce default *DbContext* does it for us automatically. But because we are writing own *DbContext*, we must tell EF to ignore it through the Entity Framework Code First Fluent API while overriding *DbContext*'s *OnModelCreating* method.

You are probably already overriding the *OnModelCreating()* method if you are writing a custom *DbContext*. We did that in the example above when we mapped two entity types to unconventional table names.

While you're implementing *OnModelCreating()*, be sure to add the "ignore" instruction as shown here:

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

Specify the *DbSet*s

The Entity Framework *DbContext* knows which entities to include in the model it manages by examining the properties that return *DbSet(Of TEntity)*.

If you were using this *DbContext* directly, you would probably add a *DbSet* definition for almost every type in your model. But you won't be using this *DbContext*. You'll be using an *EntityManager*. So you can get away with specifying the minimum number of *DbSet*s necessary for EF to identify all of the desired model types.

EF automatically includes in the model both the entity type mentioned in the *DbSet* definition **and every entity type returned by that type**. The entity type discovery process is recursive so you only need to mention the "**root**" types.

While there is no harm in mentioning the other entity types, in the example we only had to mention *Supplier* because the *Supplier.Products* navigation property returns *Product* entities and the *Product* entity has a *Category* navigation. All types in our model are reachable from *Supplier* making it the one "root" entity in the model.

Initialize the database (optional)

Notice the *DropCreateDatabaseIfModelChanges<T>* object passed into the static *Database.SetInitializer* method call.

```
public ProductDbContext(string connection = null) : base(connection)
{
  // Do not use in production; for early development only
  Database.SetInitializer(
          new DropCreateDatabaseIfModelChanges<ProductDbContext>());
}
```

That's an initialization strategy object that tells EF to re-create the database if it detects model changes.

Here are the stock initialization strategies that are useful in early development when you don't care about the database schema and data. **All of them are dangerous in production code**:

```
//Default strategy: creates the DB only if it doesn't exist
Database.SetInitializer(new CreateDatabaseOnlyIfNotExists<ProductDbContext>());
//Recreates the DB if the model changes but doesn't insert seed data.
Database.SetInitializer(new RecreateDatabaseIfModelChanges<ProductDbContext>());
//Always recreates the DB every time the app is run.
Database.SetInitializer(new DropCreateDatabaseAlways<ProductDbContext>());
```

You can create your own initialization strategy by inheriting from one of these and overriding the *Seed* method.

Never enable **any** of these database initialization strategies in production code. Never enable them if there is a chance that the Entity Framework could destroy potentially valuable data ... even valuable test data.

Entity Framework (re)creates a database on SQL Server Express by default. You can change that default.

You can stop Entity Framework from creating or re-creating the database - **and should do so in production** - by calling the *Database.SetInitializer* static method with a null argument. One possible place to do that is in the constructor as follows:

```
public ProductDbContext(string connection) : base(connection)
{
   Database.SetInitializer(null); // Never create a database
}
```

Multiple *DbContext* classes

You can write more than one *DbContext*. You might have multiple *DbContext*s if you sourced entity data from multiple databases; each *DbContext* can only connect with a single database.

You might have multiple domain models, each targeting a separate application module and serving a separate business purpose. Although the domain models are different, the may store their entity data in the same database.

A DevForce *EntityManager* can hold entities sourced from multiple databases and multiple *DbContext*s. DevForce will use a distributed transaction if an *EntityManager* is asked to save entities from different databases.

No *DbContext* in Silverlight and mobile projects

You add *DbContext* classes to .NET 4.5 projects, **never to Silverlight and mobile projects**. The *DbContext* is an Entity Framework component. Entity Framework components are not defined in client-only technologies.

You **share the source code** for your entity classes with your client application, typically by linking to entity class files in the full .NET model project(s). Remember to exclude all *DbContext* classes when linking to .NET model project files. You can do this using compiler directives (e.g., #if SILVERLIGHT) if you only need to exlude portions of a file, or by not including these files in the client project.

You write it, DevForce calls it

If you have previous EF Code First, you are used to constructing a *DbContext* instance and using it to query and save entities. You won't be doing that anymore; that's DevForce's job. The DevForce *EntityServer* discovers your *DbContext* class and uses it on your behalf to perform Entity Framework tasks in response to client requests.

Learn more

Authoring a custom *DbContext* is not something we can cover adequately in the DevForce Resource Center.Please review external sources, starting with Microsoft's ADO.NET Entity Framework website, to learn more.