Contents

- What is Code First?
- Entity Framework Code First
- DevForce Code First
- Your entity classes are DevForce entities
- DevForce AOP entities
- Basic Development Workflow
- Model Projects and Project References
- DevForce AOP classes vs. DevForce POCO classes
- Additional resources

DevForce developers can write their Entity Framework-based models in **Code First** style. In Code First you write your CLR entity classes by hand and later map them to the corresponding database. DevForce makes class authoring easier by weaving DevForce infrastructure into your classes using SharpCrafters' PostSharp Aspect Oriented Programming (AOP) technology.

Code First is not supported in Windows 10 Universal apps at this time.

What is Code First?

Code First is a style of Entity Framework model development in which you build the entity model entirely in code. Code First is one of three development styles supported by Entity Framework.

Database First
Derive entity definitions and entity-to-database mappings from database schema, store the definitions and mappings in an XML file (the EDMX), and then generate .NET entity classes from the entity definitions in that EDMX.

Model First
Define entities in a "neutral" entity modeling language and store them in an EDMX file. You typically generate .NET entity classes from the definitions in the EDMX as in Database first approach. Later, when ready to interact with a database, you map the entity definitions to the database; the mappings are stored in the EDMX which becomes indistinguishable from an EDMX build Database First.

Code First
Write and maintain .NET entity classes by hand. Do not use an EDMX to map the entity classes to a database. Use some combination of convention and programmatic configuration instead.

"Code First" earned its name because it prioritizes development of entity classes. In Code First, you just write code. You don't run the Entity Framework visual design tool, neither to design entities (Model First) nor to map entities to an existing database (Database First). You don't have to create or update a database schema; Entity Framework Code First can generate a temporary database for you that which may be adequate to your development needs for some time. There is no XML file – no EDMX file – to represent your model and storage schemas.

Perhaps it should have been called **Code Only**. Its signature distinguishing characteristic is that you do everything in code; **there is no Entity Data Model (EDM)**.

Entity Framework Code First

"DevForce Code First" is technology that builds upon "Entity Framework Code First" (EF CF). DF2012 Code First support requires Entity Framework version 5 or above, which you can obtain from NuGet here. The *DevForce 2012 Code First* NuGet package automatically installs the EF 5 package if not already present. If the EF 6 package is already present, DevForce will use that.

Microsoft's Data Developer Center is a good place to learn about Entity Framework Code First. But **everyone should begin with Julie Lerman's** short book about Entity Framework Code First. Also check out Julie's 6 minute video introduction.

You will be using your EF Code First skills when you build DevForce entity models in Code First style. We won't try to explain the details of EF Code First development in our documentation. In fact, we typically assume that you have already added EF Code First to your application project(s) and know how EF's Code First works.

Usually you can rely on EF Code First naming conventions to map most of your entity classes to the tables and columns in your database. For example, you don't have to do anything when there's a "Name" column in the "Customer" table to match the Name property of your Customer entity.

When your entity model classes and database schema are less perfectly aligned, you'll clarify their correspondences explicitly with attribute annotations in your entity class code or by configuration with the Entity Framework Fluent API.

However, the workflow for DevForce Code First development is a bit different so you'll want to read more about that workflow here in the DevForce Resource Center.

DevForce Code First

In Code First, you write.NET entity classes by hand.

You can use a tool to generate the initial version of those classes from database schema as described in the Code Second topic. Subsequently, you maintain the class by hand.

Because you are writing the class by hand, you'll want to write as little code as possible. Such a class could be as simple as this Category entity:

```
[ProvideEntityAspect]
public class Category
{
  public int CategoryId { get; set; }
  public string CategoryName { get; set; }
}
```

Notice that there's no base class. *Category* has two properties – a primary key and a name – both auto-properties with no explicit implementations.

There is one bit of noise – the *ProvideEntityAspect* attribute - that identifies *Category* as an entity class participating in the DevForce entity management system.

Your entity classes are DevForce entities

Category is a full-fledged DevForce entity. Like all DevForce entities you can

- Use LINQ to query for *Category* entities and serialize them over a network.
- Modify one.
- Listen for changes when you set a property.
- Decorate its properties with stock or custom validation attribute and expect the DevForce validation system to apply the requested validation checks on client and server.
- Add property interceptors as you would for DevForce generated entities.
- Ask a Category instance for its current EntityState
- Save a set of changed *Category* entities back to the database.

You can do anything with a *Category* entity that you can do with an entity DevForce generated from an EDMX.

"How can this be?" you ask. It looks nothing like the *Category* entity we might have generated from the Northwind database using Database First. That generated class is 140 lines long and looks a bit like this:

```
/// <summary>The auto-generated Category class. </summary>
[DataContract(IsReference=true)]
 [DataSourceKeyName(@"NorthwindIBEntities")]
 [DefaultEntitySetName(@"NorthwindIBEntities.Categories")]
public partial class Category : Entity {
  // … snip …
   [Key]
   [Bindable(true, BindingDirection.TwoWay)]
   [Editable(true)]
   [Display(Name="CategoryID", AutoGenerateField=true)]
   [RequiredValueVerifier( ErrorMessageResourceName="Category_CategoryID")]
   [DataMember]
  public int CategoryID {
    get { return PropertyMetadata.CategoryID.GetValue(this); }
    set { PropertyMetadata.CategoryID.SetValue(this, value); }
   }
  // … snip …
   [Bindable(true, BindingDirection.TwoWay)]
   [Editable(true)]
   [Display(Name="CategoryName", AutoGenerateField=true)]
   [StringLengthVerifier(MaxValue=15, IsRequired=true,
      ErrorMessageResourceName="Category_CategoryName")]
   [DataMember]
  public string CategoryName {
    get { return PropertyMetadata.CategoryName.GetValue(this); }
```

```
    set { PropertyMetadata.CategoryName.SetValue(this, value); }
  }
  // … snip …
}
```

The comparison isn't entirely fair. Much of the material you see here in the generated entity is optional. Within the DevForce EDM Designer property window you can disable generation of some or all of the attributes. You have long been able to take control of DevForce T4 code generation by customizing the template.

On the other hand, we generate these attributes by default for good reason: most DevForce developers write client applications that work directly with DevForce entities. These attributes make it faster and easier to write the UI of a business application.

The *Editable*, *Bindable* and *Display* attributes are hints to the UI about how to bind the properties to visual controls and present messages. The *Verifier* attributes specify validation rules that ensure input data integrity on both client and server. You'd be pleased to have the *DataMember* attributes if you had to write your own utility to serialize *Category* entities.

You may want to add some of these attributes to your own Code First entity classes. The following is a reasonable update to our Code First *Category*:

```
[ProvideEntityAspect]
public class Category
{
    [Bindable(true, BindingDirection.TwoWay)]
    [Editable(false)]
    [Display(Name="ID", AutoGenerateField=true)]
    [RequiredValueVerifier]
  public int CategoryId { get; set; }
    [Bindable(true, BindingDirection.TwoWay)]
    [Display(Name="Category Name", AutoGenerateField=true)]
    [StringLengthVerifier(MaxValue=15, IsRequired=true)]
  public string CategoryName { get; set; }
}
```

Even with these additions, the handwritten class is much smaller than the generated class. Conspicuous among the remaining differences:

Generated *Category*	**Handwritten *Category***
Inherits from DevForce Entity	No base class
partial class definition	Not a *partial* class
Properties are implemented with *PropertyMetadata* helper objects	Auto-properties
	ProvideEntityAspect attribute adorns the class

DevForce no longer requires that Entity Framework-backed entities inherit from the DevForce *Entity* base class. In fact, your handwritten Code First classes **must not** inherit from DevForce *Entity*.

DevForce entity class generation emits a *partial* class definition so you can keep your custom entity code in a separate file, safely apart from the generated code file. The handwritten Code First class is all yours; you wouldn't divide your *Category* business logic among separate files without a compelling reason.

The generated *PropertyMetadata* helper objects are gateways to the DevForce entity infrastructure – persistence, change notification, validation, property interception. You will still be using that infrastructure in your handwritten entity class … but that infrastructure is made available in a different way.

DevForce AOP entities

The *Category* class at runtime is not exactly the class you wrote. DevForce took your compiled class and injected it with DevForce entity infrastructure.

The build process was altered when you installed DevForce. There are new steps in the build pipeline that gather entity metadata and apply that metadata to rewrite your entity classes using SharpCrafters' PostSharp Aspect Oriented Programming (AOP) technology.

Normal classes are untouched. But when DevForce sees a class decorated with the *ProvideEntityAspect* attribute – or sees a class derived from a class decorated with that attribute – the class rewrite process kicks in.

The runtime result is a class that behaves like a generated entity class. Your source code remains the same; the DevForce infrastructure is invisible. But under the hood, the rewritten class is implemented substantially the same way as the generated entity with its *PropertyMetadata* helpers.

You can see the class's runtime implementation by examining its parent assembly with a decompilation tool such as Reflector.

We're not being sneaky. Our objective is to help you focus on the business logic of your entity class by hiding the infrastructural concerns.

There is a lot of infrastructure to hide. DevForce application developers expect to program with rich entity classes on a variety of .NET clients: Windows Forms, WPF, Silverlight, Windows Phone, and Windows 8 Metro.

DevForce entities are designed for distributed application UI scenarios with built-in support for querying, serialization and transmission over a network, data binding, validation, reversible edits, caching, and property interception. These are capabilities that cut across all entities. They aren't particular to any one business purpose and are usually irrelevant to the developer who is trying to understand and write the domain logic for an application. For example, you shouldn't have to write property change notification statements inside your property setters. You shouldn't have to see how navigation properties (e.g., *customer.**Orders***) find their related entities in cache. Aspect Oriented Programming (AOP) is a great way to hide the details of such "cross cutting concerns".

You write entity business logic like custom verifiers and property interceptors the same way you wrote them for generated entity classes. AOP doesn't change that. The only AOP attributes you have to know are the attributes that tell DevForce to rewrite your class (*ProvideEntityAspect* and *ProvideComplexAspect*). You don't create your own AOP attributes.

In fact you can't write new PostSharp attributes under the license we provide with DevForce. You can write custom attributes and do your own AOP programming under your own PostSharp license.

You retain all the functionality of a traditional DevForce entity even if though you can't see the infrastructure in the source code. You can still create the entity with "new". You can still access its infrastructural characteristics through its *EntityAspect* property. You can still debug it and test it in isolation or in combination with other entities cached in an *EntityManager*.

Basic Development Workflow

One can easily become lost in details and miss the overall simplicity of the Code First approach. An overview of the developer's workflow can offer perspective.

The path you take depends first on whether you are starting from scratch ("green field") or working with an existing database ("brown field").

Green Field	Brown Field
Write AOP entity model classes	Generate initial entity model from the database
Add a custom *EntityManager*	
Add a custom *DbContext* (optional)	Add a custom *DbContext* (optional)
Enable metadata generation	Enable metadata generation
Set a database connection string(optional)	Set a database connection string (required)
Run, review,and iterate entity model development	Run, review,and iterate entity model development

The two paths are almost the same, differing mostly in how you get started.

Other topics in this Code First series cover the details of building DevForce AOP entities during Code First model development.

Model Projects and Project References

Your Code First entity model can co-reside with other classes in your application or web project. That's often the practice in demos and tutorials, but most developers prefer to keep the model in its own model project.

Whatever your design, the project hosting your entity model must have references to certain Code First and AOP libraries in addition to the usual DevForce libraries.

Library	Purpose
EntityFramework	Entity Framework Code First support
System.Data.Entity	EF 5 only - Entity Framework library you'll likely need if you define a custom *DbContext*
PostSharp	Performs AOP assembly re-write and injects DevForce infrastructure

IdeaBlade.Aop	DevForce AOP attributes and related components.

The DevForce 2012 Code First NuGet package installs these dependent assemblies for you.

This project requires the .NET 4.5 framework and above because it depends upon the Entity Framework. If you're building a Silverlight or mobile application, you know that you will be creating a separate client project (preferably a corresponding client model project) that references the environment-specific libraries.

Library	Purpose
PostSharp	Portable version of the library for Silverlight, WinRT, Windows Phone and mobile environments. Performs AOP assembly re-write and injects DevForce infrastructure
IdeaBlade.Aop.SL	DevForce AOP attributes and related components for Silverlight applications.
IdeaBlade.Aop.WinRT	DevForce AOP attributes and related components for WinRT applications.
IdeaBlade.Aop.WP8	DevForce AOP attributes and related components for Windows Phone 8 applications.

As before, the DevForce 2012 Code First NuGet package installs these dependencies for you.

Code First support is provided in Windows Store and Phone applications as of DevForce version 7.2.0.

DevForce AOP classes vs. DevForce POCO classes

We close this topic by noting the difference between DevForce AOP entities and DevForce POCO classes.

They appear rather similar. You are both entity classes that you write by hand. You aspire to keep them as free of infrastructure concerns as you can, striving to write them as "Plain Old CLR Objects", POCOs.

The critical difference is that AOP entity classes are Entity Framework classes. They are designed to be queried and saved using the Entity Framework as both a mapping and data access layer.

DevForce POCO classes are not tied to the Entity Framework. You can populate POCO objects from any kind of data source: in-memory objects, a web service, a Web API service, an OData service, a queue, a NoSQL database, pretty much anything you can dream of.

Additional resources

- Code sample: Code First Walk (WPF)
- Code sample: Code First Tour (SL)
- Code sample: DevForce Windows Store with Code First
- Code sample: DevForce Windows Phone with Code First