Contents

Add a custom "domain-specific" EntityManager to your model to make querying easier.

DevForce entity model code generation creates a custom EntityManager class that derives from *IdeaBlade.EntityModel.EntityManager*. This class has several constructors and a query property for every entity type in the model.

The generated custom *EntityManager* is a great convenience to the application developer. But it's not strictly necessary. You can get by using just the base DevForce *EntityManager*. When you need a query for *Product*s, you can write:

```
EntityQuery<Product> productsQuery = manager.GetQuery<Product>();
```

But most of us prefer to write ...

```
var productsQuery = manager.Products;
```

... which you can do when the *manager* variable is an instance of your custom *EntityManager* class.

You may not want to have a custom *EntityManager* if you always encapsulate queries in a “Repository” or “DataSource” class. Your repository is going to expose the convenience API anyway; internally you can use *EntityManager.GetQuery(Of T)* expressions to get the job done.

To be clear, as long as you've defined a custom *DbContext* you don't have to define an *EntityManager* even if we think it's a good idea.

You must write a custom *EntityManager* if you don't write a custom *DbContext*. You can write one or the other or both. But you must have a custom *EntityManager* or a custom *DbContext*.

Write a custom *EntityManager*

You write all model code yourself when you choose the "Code First" approach. You decide whether you will have one custom *EntityManager*, multiple custom *EntityManagers*, or no custom *EntityManagers*. For each *EntityManager* that you write, you decide which entity types should have query properties.

Here's a custom *EntityManager* for a model with the *Category* and *Product* entities.

```
using IdeaBlade.EntityModel;
public class ProductEntities : EntityManager
 {
   public EntityQuery<Category> Categories { get; set; }
   public EntityQuery<Product> Products { get; set; }
 }
```

In this example, there is an auto-property returning an *EntityQuery(Of TEntity)* for each entity type in the model. DevForce initializes these properties at runtime when constructing an instance of the *ProductEntities* manager.

You don't have to create a query property for every type in the model. You may prefer to omit query properties that return subordinate types belonging to an Aggregate Root. Suppose that, in your design, *Order* entities have *OrderLineItems*. You've decided that *OrderLineItems* shouldn't be queried directly; they should always be retrieved by navigation from an *Order* instance. You want to discourage queries for *OrderLineItems* by omitting the query property for *OrderLineItems* from your *EntityManager*'s API. Of course you could always query for them if you had to by using an *EntityManager.GetQuery<OrderItem>()* expression.

It's your model; it's your *EntityManager*

Add constructors

You can get by with no constructors at all. The DevForce base *EntityManager* class has a constructor with all optional parameters; the example *ProductEntities* class is delegating to that constructor implicitly.

Most developers add constructors eventually. Certainly the most popular is a constructor that allows you to create an instance of the *EntityManager* that does not immediately connect to the database.

```
public ProductEntities(bool shouldConnect) : base(shouldConnect) { }
//... create an instance somewhere in the application ...
var manager = new ProductEntities(shouldConnect: false); // offline
```

```
//... connect when it's time to connect ...
manager.Connect();
```

A disconnected *EntityManager* is especially useful in automated test scenarios.

Someday you may want to have the same constructors that DevForce creates for a generated *EntityManager*. It's boiler-plate code that looks like this for our *ProductEntities* example.

```
#region Constructors
public ProductEntities(
    bool shouldConnect = true,
    string dataSourceExtension = null,
     EntityServiceOption entityServiceOption = EntityServiceOption.UseDefaultService,
    string compositionContextName = null)
    : base(shouldConnect, dataSourceExtension, entityServiceOption, compositionContextName) { }
public ProductEntities(EntityManagerContext entityManagerContext)
    : base(entityManagerContext) { }
public ProductEntities(
     EntityManager entityManager,
    bool shouldConnect,
    string dataSourceExtension = null,
     EntityServiceOption entityServiceOption = EntityServiceOption.UseDefaultService,
    string compositionContextName = null)
    : base(entityManager, shouldConnect, dataSourceExtension, entityServiceOption, compositionContextName) { }
public ProductEntities(EntityManager entityManager, EntityManagerContext entityManagerContext = null)
    : base(entityManager, entityManagerContext) { }
#endregion Constructors
```

You can copy and paste this code fragment, replacing the word "ProductEntities" with the name of your custom *EntityManager* class, or download and use the code snippet that writes this for you.

Model Discovery

DevForce and Entity Framework must learn which types belong in your project's entity model. They learn by inspecting your project and reflecting on its classes.

If you do **not** write a custom *DbContext*, DevForce creates one dynamically, configuring that *DbContext* to discover entity types based on the custom *EntityManager(s)* you write or the types defined in your project.

1. If you write only one *EntityManager*, DevForce enrolls every Code First type that it can find in the project.
2. If you write more than one *EntityManager*, DevForce finds all types mentioned in their *EntityQuery<T>* properties and passes them to the dynamically created *DbContext* as "root" entities for analysis. Entity Framework will walk the navigation paths that extend from these root entities to enroll related entity types.

If you write a custom *DbContext*, DevForce asks the Entity Framework to use your *DbContext* to discover which types belong in the Code First model. The types in your *EntityManager* (if you have one) and the types in your project are not a factor in determining the types in your entity model. Make sure that your application only requires types that Entity Framework will discover with your *DbContext*.