

Contents

- [Overview](#)
- [Create the entity](#)
- [Set the key](#)
- [Initialize properties](#)
 - [Standard default values](#)
 - [EDM default values](#)
 - [Custom default values](#)
 - [Set a default value function](#)
 - [Set the DataEntityType.DefaultValue](#)
 - [Initialization precedence](#)
- [Add to the EntityManager](#)
- [Consider writing a constructor](#)
- [Consider writing an entity factory](#)

Create and manipulate a new entity that you intend to insert into the database.

Overview

Most applications can add new entities to the database. The essential steps are

1. Instantiate the entity object.
2. Set its unique [EntityKey](#).
3. Initialize some properties as appropriate.
4. Add the new entity to an [EntityManager](#).

Steps #1 and #4 are always required. Step #1 must come first; steps #2, #3, and #4 may occur in any order.

You can write the code to create an entity in many ways. Some ways are more maintainable than others. We'll recommend some practices as we describe the technical details.

Create the entity

Most developers create an entity by calling one of its constructors, perhaps the default constructor which every entity must have.

```
cust = new Customer(); // the implicit default constructor
cust = new Customer() ' the implicit default constructor
```

Set the key

Every entity must have a unique [EntityKey](#) before it can be added to an [EntityManager](#) and saved to the database. Key setting is a [topic unto itself](#); you may have to [generate keys with custom code](#).

Initialize properties

Before you write code to initialize properties in a [custom constructor](#), it helps to know how DevForce initializes them first. You may not *need* to initialize properties in code.

We're talking about DevForce entity data properties. Your custom properties and [POCO](#) entity initializations are up to you.

Standard default values

The data properties of a new entity return default values until you initialize them:

- .NET primitive properties return their default values
 - numeric properties return zero
 - string properties return the empty string
 - nullable struct properties (e.g., *int?* and *Nullable(Of Integer)*) return their null instances.
- [ComplexType](#) properties return *ComplexType* objects whose values are initialized as described here.
- Reference navigation properties return the *null* value until attached to an [EntityManager](#); then they behave like active [navigation properties](#).
- Collection navigation properties return an empty list until attached to an [EntityManager](#); then they behave like active [navigation properties](#).

EDM default values

The [Entity Data Model \(EDM\)](#) may hold default values for some of the properties, values that were derived from the database or entered manually through the [EDM Designer](#). DevForce uses these model default values instead of the usual defaults.

Custom default values

We can't set a default value for some data types, such as `DateTime`, in the [Entity Data Model](#), but we can still set a default at run time.

Set a default value function

You can modify or replace the DevForce-defined defaults by setting a *DefaultValueFunction*.

This is set once on the [EntityMetadata](#) class, and allows you to customize the default values for data types throughout your model. This is particularly useful with `DateTime` and `DateTimeOffset` data properties, since the DevForce defaults for these assume local time.

You can set only one *DefaultValueFunction* within your application.

Here's a sample which will return `DateTime.Today` with `DateTimeKind.Unspecified` as the default value for any `DateTime` data property within your model, and allow DevForce defaults to be used for all other data types.

```
EntityMetadata.DefaultValueFunction = (t) => {
    if (t == typeof(DateTime)) { return DateTime.SpecifyKind(DateTime.Today, DateTimeKind.Unspecified); }
    return EntityMetadata.DevForceDefaultValueFunction(t);
};
```

Set the `DataEntityProperty.DefaultValue`

You can also manually set the *DefaultValue* to be used on specific data properties. You need to do this once only for the properties in question, and DevForce will then use these default values when new entities are created.

For example, the following sets default values for several `Employee` data properties.

```
Employee.PropertyMetadata.LastName.DefaultValue = "<Unknown>";
Employee.PropertyMetadata.BirthDate.DefaultValue = DateTime.SpecifyKind(DateTime.Today, DateTimeKind.Unspecified);
```

Initialization precedence

In summary, the data property of a new DevForce entity is initialized as follows:

1. with the EDM default value if defined, or ...
2. with the standard or custom default value unless ...
3. the developer sets the initial value in code.

DevForce decides the default value in a "lazy" manner so you can take your time initializing the property. DevForce settle upon the actual initial value when the property is first read.

Note that the initialization of default values for your properties is independent of how the entity is created. You can use the *CreateEntity* method of the *EntityManager* or a simple entity constructor.

Add to the *EntityManager*

You should add the entity to an *EntityManager* before making it available to the caller. Navigation properties won't navigate until you do. Properties won't validate automatically. You can't save an entity until its been added to an *EntityManager*.

Add the entity to the manager in one line:

```
manager.AddEntity(cust);
```

```
manager.AddEntity(cust)
```

You must have have set the [EntityKey](#) - or be using an auto-generated key - before you add the entity to the manager as [mentioned above](#).

Consider writing a constructor

DevForce does not generate an entity constructor. In many cases you can "new-up" the entity using its default constructor, add it to the *EntityManager*, as we showed above.

This approach suffices in only the simplest cases. Many entities must satisfy specific invariant constraints from the moment they're born. Some child entities should only be created by their parents. Some entities have complex creation rules that you want to hide. Some entities [should never be created at all](#).

More often than not you will [write a constructor](#).

Consider writing an entity factory

It may take only two lines to create an entity but it's rarely a good idea to repeat the same two lines throughout your code base. Two lines have a way of becoming three, four and more. The creation rules become fuzzy and you start to see Customer being created several different ways for no obvious reason.

We urge you to [wrap creation logic in factory methods](#) - even if it is only two lines at the moment. Then decide if the factory methods should be static members of the entity class or instance methods of a separate entity factory class.

[Different guidance](#) applies when creating non-root members of an [Aggregate](#). The mechanical details of entity creation don't change but the way you arrange the code does change.