Contents

We provide a brief **overview** of validation within DevForce here. Subsequent topics will discuss each area in more depth.

The VerifierEngine

All validation in DevForce is performed by a *VerifierEngine*. Every *EntityManager* has its own instance of a *VerifierEngine*, accessible via its *VerifierEngine* property. *VerifierEngines* can also be created and used independently. The *VerifierEngine* is not thread-safe, but with some care may be shared by multiple *EntityManagers*.

A *VerifierEngine* contains a collection of *verifiers*. A *verifier* is an instance of some subclass of the *Verifier* class.

The *VerifierEngine* also provides methods that allow *verifiers* to be evaluated sequentially against an instance of a .NET class. This is the process of "performing a validation." The object to be validated can be a DevForce entity but it doesn't have to be; the object can be of any concrete type.

Each verifier execution produces a *VerifierResult*. *The engine accumulates these results in a* VerifierResultCollection as it proceeds and returns the entire collection as its own result.

Define Verifiers

Most *verifiers* are responsible for the validation a single property on a single target type. Validations of this form are usually specified by marking up the target type's properties with a variety of *validation attributes*. These *validation attributes* can either have been automatically generated on the entity by DevForce during code generation, or may have been added directly to the class by the developer using custom attributes.

DevForce also provides a number of pre-defined verifiers and support for custom verifiers.

Add Verifiers to a VerifierEngine

Verifiers can be added to a *VerifierEngine* in two ways:

- The engine can discover them automatically by inspecting the .NET types for verifier attributes.
- The developer can add them programmatically.

The application can combine these methods.

Whenever a *VerifierEngine* is asked to perform validation on a type, its first step is to discover all of the verifiers that are applicable to that type. Some of these verifiers are defined using property level attributes that may be applied to the type being validated; but verifiers may also be defined in .NET code, and even in XML. The *VerifierEngine* discovers all of these verifiers, and creates instances of each in an internal collection. These instances will be used to perform the actual validations.

Configure Validation

Each *verifier instance* has its own properties which tell the *VerifierEngine* the conditions under which it is applicable. For example, you can define a verifier so that it runs *before* a proposed new property value is pushed into the business object; or *after*; or even both (though that is unusual). You also want most verifiers to run whenever an entire *instance* of a type is being validated. To specify these things, you specify the *ExecutionModes* on an instance of the *VerifierOptions* type.

Other configuration options allow you to control error handling and how empty strings are handled.

Perform a validation

The *VerifierEngine* has several overloads available to actually cause a validation to occur. The methods can be called directly, but they will also be called automatically by DevForce at the following points:

- Whenever any entity within an *EntityManager* has any of its property values changed. In this case the EntityManager sees the change to the entity and internally calls its *VerifierEngine* to perform the validation.

- Whenever an entity is saved, the *EntityServer* validates every entity before it is saved. If any do not validate, the save is canceled.

See the perform a validation topic for additional information.

Property Validation

In the case of a change to a property value, we really only want to perform the minimum number of validations that are relevant to the property being changed. This is called a property validation. Using its *VerifierEngine*, the *EntityManager* will perform automatic "before" and "after" property validation as property setters are invoked.

Instance Validation

During a save DevForce will validate each entity. This will mean validating every property along with any validations that cross properties or involve related objects. This is referred to as an instance validation.

This instance validation is automatically performed by DevForce on the *EntityServer*. Instance validation may also be performed on demand by calling *VerifierEngine.Execute*.

The server-side validation performed during a save has some important considerations, most importantly that any application, whether n-tier or 2-tier, will always have at least two *VerifierEngines*:

- One within the local EntityManager;
- and one within the EntitServer's EM.

It's important to have this in mind when adding verifiers or modifying configuration so that your settings apply to both "client" and "server".

For instance, if setting:

```
myEM.VerifierEngine.DefaultVerifierOptions.ShouldTreatEmptyStringAsNull = false;
```

in the client application, validation will not fail when setting a non-nullable property to "" (empty string).

However, when *myEM.SaveChanges()* is called, validation will be performed once more in the *EntityServer* and if *ShouldTreatEmptyStringAsNull* is not explicitly set to false there, the save will fail.

In the *EntityServer*, we can set validation in the *SaveInterceptor* ValidateSave method:

```
protected override bool ValidateSave() {
  EntityManager.VerifierEngine.DefaultVerifierOptions.ShouldTreatEmptyStringAsNull = false;
 return base.ValidateSave();
 }
```

See the validate on the server topic for more information on server-side validation.

Result of a validation

Regardless of which form of validation is performed the result of a validation is always a *VerifierResultsCollection* which as the name suggests, is a collection of *VerifierResult*s. Each *VerifierResult* contains a reference to the object being validated, the validation instance used to perform the validation and most importantly the "result" of the validation. *VerifierResults* that represent "errors" are automatically added to each entities *EntityAspect.ValidationErrors* collection.