Contents

Much of the work involved in **configuring a verifier** is performed via the *VerifierOptions* class.

This class collects a number of settings that affect a verifier's behavior, such as when it it is run and how it handles exceptions. An instance of the type is available as a property on the *VerifierEngine* class via the *DefaultVerifierOptions*; property and also on the *Verifier*, *VerifierArgs*, and *VerifierResults* classes as the *VerifierOptions* property.

VerifierOptions are "inherited" or "defaulted" from the VerifierEngine down to all of a VerifierEngine's Verifiers and VerifierArgs and finally to the VerifierResults of each Verifier. This will be described in more detail later.

Properties of the VerifierOptions class

Property	Description	Type	System Default Value
ExecutionModes	Gets or sets the conditions under which which a verifier is executed. See the material immediately below the table for more information about these.	Enum	*VerifierExecutionModes.InstanceAndOnBeforeSetTriggers*
ErrorNotificationMode	Used to determine whether the Entity should throw errors during a property verification or raise the errors thru the INotifyDataErrorInfo interface instead.	Enum	*VerifierErrorNotificationMode.Notify*
ShouldExitOnBeforeSetError	Whether or not to perform the 'set' of a proposed value when an error occurs. Note that this setting is only applicable with 'BeforeSet' ExecutionModes.	Boolean	*false*
ErrorContinuationMode	Gets or sets whether a failure in this verifier should stop the execution of the remainder of the batch in which this verifier is executing	Enum	*VerifierErrorContinuationMode.Continue*
TreatWarningsAsErrors	Whether to treat warnings as errors	Boolean	*false*
ShouldTreatEmptyStringAsNull	For required value validation, this determines whether empty strings are treated as if they were null.	Boolean	*true*
RawVerifierOptions	A version of this same VerifierOptions instance without "inheritence" interpretation. May be used to determine which properties are actually inherited	VerifierOptions	*N/A*

The system default values shown in the table above are the default settings for the properties of *DefaultVerifierOptions*.

All *VerifierOptions* properties can be **inherited** from a parent class and by default do so. A *VerifierEngine* is parent to any *Verifiers* contained within it. A *Verifier*, in turn, is parent to any *VerifierResults* resulting from its execution.

By default, every verifier is created with a *VerifierOptions* instance is created with every property conceptually set to inherit its value from its parent as described below:

All of enum-valued property types on *VerifierOptions* (VerifierExecutionModes, VerifierErrorNotificationMode and VerifierErrorContinuationMode) each have a special *Inherit* enumeration value that may be used to indicate inheritance from a parent. Properties that are of type bool? (nullable booleans) may be set to *null* to accomplish the same result.

This means that if any property on a VerifierEngine's *DefaultVerifierOptions* is changed, then by default, every *Verifier* and *VerifierResult* will inherit this change. And since *Verifier* and *VerifierResult* have their own *VerifierOptions* property, any aspect of *VerifierOptions* can be overwritten (changed from 'Inherit' to another value) for any given *Verifier* or *VerifierResult*.

It's important to remember that your application contains at least two *VerifierEngines*: one on the client and one on the server. If you modify any default options you must be sure to do so on both client and server when appropriate.

ExecutionModes

The determination of when and how a verifier is executed is controlled via the *IdeaBlade.Validation.VerifierOptions.ExecutionModes* and *IdeaBlade.Validation.Verifier.TriggerLinks* properties on each verifier. A verifier can be executed either

1. in response to a single change (e.g., of a property value); or
2. in the context of validating an entire object.

The first type of validation listed above is known as a *property value validation*. The second is known as an *instance validation*.

We would apply a *property value validation* when (for example) a user was changing, or had just changed, the HireDate on an employee. At that time we would only want to run those validation tests whose outcome we feel may have been affected by this specific change. Our goal would be to provide the end user with instant feedback about their change. In most cases, it will never be easier for them to correct a mistake than *immediately* after making it!

Property value validations can be subdivided into *before* and *after* categories. A *BeforeSet* validation is applied *before* a proposed new property value is actually pushed into the business object. This can be used to prevent invalid data from ever getting into the business object. An *AfterSet* validation is applied *after* the new value is pushed into the business object. The *ShouldExitOnBeforeSetError* and *ErrorNotificationMode* properties described later can be used to control further details involving a property value validation.

Instance validation describes an operation that completely validates an entire entity, applying all relevant validation rules, whether those rules apply to individual properties of that entity, to a combination of its properties (the validity of whose values must be assessed in relation to each other), or in some other way to the entity as a whole.

We might perform an instance validation, for example, on a particular instance of an Employee before permitting it to be saved to the database. This would likely require performing validation tests on a number of individual properties of the employee object; it might also require testing the state of related objects (such as the Orders written by that Employee). Only if the *entire suite* of tests required to fully validate the Employee were passed would we give the okay to save it to the database.

Note that instance validation would be unnecessary if we could really be sure that every change affecting our Employee would be addressed by an appropriate set of property value validations. In practice this can be very difficult, or even impossible, to ensure. What if, for example, our Employee is changed by some other application which doesn't apply the same rules that our does? What if it is changed directly in the database by someone with access to that? Even when we can guarantee that neither of those things happen, the mechanisms by which a given entity can be changed inside a single application can become quite complex over time. For all of those reasons, developers commonly perform instance validation at such key junctures as when an entity is submitted for saving. It's an important last line of defense against invalid data.

It is common for a given verifier to be applicable during instance validation as well as during property value validation. Every verifier comes with information about the situations in which it should be executed, via its *VerifierOptions.ExecutionModes* property. That property takes values from a *VerifierExecutionModes* enumeration whose choices permit you to make your verifier run in any and all of the circumstances you deem appropriate. The values in the *VerifierExecutionModes* enumeration, and their impacts, are as follows:

Enum value	Description
Instance	Run during instance validation.
OnBeforeSetTriggers	Run in response to designated triggers, *before* the proposed value has been pushed into the business object.
OnAfterSetTriggers	Run in response to designated triggers, *after* the proposed value has been pushed into the business object.
InstanceAndOnBeforeSetTriggers	Run during instance validation and in response to designated triggers (usually property value changes), *before* the proposed value is pushed into the business object.
InstanceAndOnAfterSetTriggers	Run during instance validation and in response to designated triggers (usually property value changes), *after* the proposed value has been pushed into the business object.

All	Run during instance verification and in response to designated triggers (both *before* the proposed value is pushed into the business object, and also *after* it has been pushed into the business object).
Disabled	Do not run.
Inherit	Inherit the ExecutionModes setting from the type's parent (*Verifier* if a *VerifierResult*, and *VerifierEngine* if a *Verifier*).

Settings of *InstanceAndOnBeforeSetTriggers* and *InstanceAndOnAfterSetTriggers*are by far this most common (after *Inherit*). *All* is a particularly **uncommon** setting because it orders triggered execution both before and immediately after a new value is pushed into a business object. However, a verifier, as it so happens, has access to information about the triggering context in which it is being run (i.e., before or after), so it is possible to include within a single verifier logic that will only be applied in one context or another. From this arises the possibility that you, the developer, might want such a verifier to run in both triggering contexts.

BeforeSet Versus AfterSet Execution

This topic deserves special discussion, as the choice between these two execution modes can be difficult.

All other things being equal, it is desirable never to allow invalid values into a business object in the first place. If all things *were* equal, one would use *OnBeforeSetTriggers* or *InstanceAndOnBeforeSetTriggers* as the *ExecutionModes* for all property-level verifiers. However, in practice, these settings can cause problems, especially before an application has been fully fleshed out. For example, in the user interface, a *BeforeSet* verifier can demand that the end user fix a bad value entered for a property right then and there, before moving on to any other work -- even if she hasn't the faintest idea what to change the bad value *to*. [Note that this will NOT occur where the **ErrorNotificationMode** is set to *Notify* (the default).] In such as case, in order to handle such issues in a manner that's friendly to the end user then you may want to use *VerifierExecutionModes.InstanceAndOnAfterSetTriggers*.

ErrorNotificationMode

Regardless of whether validation is performed before or after a property is set, the next issue deals with how to handle any validation errors. The *VerifierErrorNotificationMode* enumeration offers the following choices:

Enum value	Description
Notify	Causes notification through the *INotifyDataErrorInfo* and *IDataErrorInfo* interfaces.
ThrowException	Throws an exception.
NotifyAndThrowException	Notify using *INotifyDataErrorInfo* and *IDataErrorInfo* interfaces and then throw an exception.
Inherit	Inherit the ErrorNotificatioMode from the parent.

Basically, there are two choices, either make use of the **INotifyDataErrorInfo/IDataErrorInfo** interfaces or throw an exception. A third choice, that of doing both, is available but is rarely used. We tend to recommend going with the use of notification interfaces, although there are use cases where throwing an exception makes a good deal of sense. You can even mix and match with some verifiers set up one way and others another way.

ShouldExitOnBeforeSetError

In the event that one of the "BeforeSet" ErrorNotificationModes is selected *AND* you have an ErrorNotificatioMode of Notify, you have one further choice:

- Do you want the property setter to exit before actually setting the value in the case of a validation error?

Note that exiting simply means that the property never gets set, but no error is thrown. This can be very handy, if you want to insure that "bad" values never make it into your entity but at the same time you really want to use notification semantics and don't want any errors thrown within your property setters.

Setting this value to null (Nothing in VB) is the equivalent of setting any of the enumerated values to "Inherit".

ErrorContinuationMode

Within an instance validation, many individual verifiers may need to be executed in sequence. The ErrorContinuationMode allows any individual validation to stop the process. This is often desirable if we know that the failure (or success) of one validation means that we can short-circuit the need for further validations of the instance. The ErrorContinuationMode enumeration consists of the following values:

Enum value	Description
Stop	Stop any further verifier execution within this batch if an error is encountered.
Continue	Continue executing other verifiers in this batch even if an error is encountered (the default).
Inherit	Inherit the ErrorContinuationMode from the parent.

TreatWarningsAsErrors

The execution of any verifier results in a VerifierResult instance. Each VerifierResult has a *VerifierResultCode* that describes the type of the result. Setting the *TreatWarningsAsErrors* to true, will cause any *VerifierResults* with a *VerifierResultCode* of *OkWarning* to be treated as validation errors.

ShouldTreatEmptyStringAsNull

The ShouldTreatEmptyStringAsNull property is applicable in the context of string length and/or required value validations.

The idea is to provide the developer with the ability to consider an "Empty string" as a null for the purposes of performing required value validations. This is usually a good practice, because it is very difficult for a user of an application to "see" the difference in a text box between a *null* value and an *empty string*, and having an empty string pass validation whereas a null value fails it can be very confusing.

The default setting is *True* - this means a property will fail validation if it is required and the value is an empty string. When changing the default option be sure to make the change on both client and server; this ensures that both client-side property validation and server-side instance validation performed during a save use the same settings.

RawVerifierOptions

An "uninterpreted" version of any *VerifierOptions* instance is available via the *RawVerifierOptions* property. The version returned by this property will not perform any interpretation of the property values, so you may use this version to determine whether any given property is actually inherited or not.