

## Contents

- [MergeStrategy](#)
- [MergeStrategy behavior](#)
  - [Is the entity current or obsolete relative to the data source?](#)
  - [How has it changed?](#)
  - [Is the entity represented in the data source?](#)
  - [Merging when the entity is in the data source](#)
  - [Merging when the cached entity is not in the data source](#)
  - [DataSourceOnly subtleties](#)

DevForce uses a [MergeStrategy](#) to determine how to reconcile potential conflicts between entities that are being merged into the cache with the entities that are already present in the cache.

For example, you may have an existing version of an entity in the cache, and are trying to merge with a "new" version of the same entity (the two entities have the same primary key). This "new" entity may be the result of a query, a call to [RefreshEntities](#), or may be an entity that is being imported from another workflow using a different *EntityManager*. If one or both entities have been changed, the *MergeStrategy* is used to determine which version of the entity is stored in cache.

## MergeStrategy

DevForce supports five different [MergeStrategies](#): *PreserveChanges*, *OverwriteChanges*, *PreserveChangesUnlessOriginalObsolete*, *PreserveChangesUpdateOriginal*, and *NotApplicable*. Their meanings are shown in the table below.

When reviewing the table, remember that, for every cached DevForce entity, two states are maintained: [Original and Current](#). The *Original* state comprises the set of values for all properties as they existed at the time of the last retrieval from, or save to, the datasource. The *Current* state comprises the set of values for the object's properties as the end user sees them. That is, the *Current* state values reflect any local changes that have been made since the entity was retrieved, or last saved. When an entity is persisted, it is the values in its *Current* state that are saved.

### MergeStrategies:

Strategy	Action when cached entity has pending changes
<b>PreserveChanges</b>	Preserves the state of the cached entity.
<b>OverwriteChanges</b>	Overwrites the cached entity with data from the data source. Sets the <i>EntityState</i> of the cached entity to <i>Unchanged</i> .
<b>PreserveChangesUnlessOriginalObsolete</b>	Preserves the values in the <i>Current</i> state of the cached entity, <i>if</i> its <i>Original</i> state matches the state retrieved from the datasource.  If the state as retrieved from the datasource differs from that found locally in the <i>Original</i> set of property values, this indicates that the entity has been changed externally by another user or process. In this case (with this <i>MergeStrategy</i> ), DevForce overwrites the local entity, setting the values in both its <i>Current</i> and <i>Original</i> states to match that found in the datasource. DevForce also then sets the <i>EntityState</i> of the cached instance to <i>Unchanged</i> .
<b>PreserveChangesUpdateOriginal</b>	Preserves the values in the <i>Current</i> version for the cached entity; and also updates the values in its <i>Original</i> version to match the values in the instance retrieved from the datasource. This has the effect of rendering the local entity savable (upon the next attempt), when it might otherwise trigger a concurrency exception.
<b>NotApplicable</b>	This merge strategy must be used – and may only be used – with the <i>CacheOnly</i> fetch strategy. No merge action applies because no data is retrieved from any source outside the cache.

## MergeStrategy behavior

What happens during the merge of a data source entity and a cached entity depends upon the answers to three crucial questions:

1. Is the entity current or obsolete?
2. How has it changed?
3. Is the entity represented in the data source?

### Is the entity current or obsolete relative to the data source?

We compare the cached entity's concurrency column property value to that of its data source entity. If the two are the same, the cached entity is current; if they differ, the cached entity is obsolete.

As it happens, the cached entity has **two** concurrency column property values, a *current* one and an *original* one. The value of the concurrency column in the current version is meaningless. It's the value of the concurrency column in the original version that counts.

Every DevForce entity has an original version and a current version of its persistent state. We can get to one or the other by means of a static *GetValue()* method defined on the *EntityProperty* class. For example, the following code gets the original value (as retrieved from the database) for the *RequiredDate* property of a particular Order instance:

```
DomainModelEntityManager mgr = DomainModelEntityManager.DefaultManager;
anOrder = mgr.Orders.Where(o => o.OrderID == 10248).First();
Datetime reqdDate = Order.PropertyMetadata.RequiredDate.GetValue(
    anOrder, EntityVersion.Current);
```

```
DomainModelEntityManager.DefaultManager
anOrder = mgr.Orders.Where(Function(o) o.OrderID = 10248).First()
Dim reqdDate As Datetime = Order.PropertyMetadata.RequiredDate.GetValue(
    anOrder, EntityVersion.Current)
```

Both of the following statements get the *current* value for the same property:

```
reqdDate = Order.PropertyMetadata.RequiredDate.GetValue(
    anOrder, EntityVersion.Current);
reqdDate = anOrder.RequiredDate; // same as above (but simpler!)
```

```
reqdDate = Order.PropertyMetadata.RequiredDate.GetValue(anOrder, EntityVersion.Current)
reqdDate = anOrder.RequiredDate ' same as above (but simpler!)
```

Again, DevForce and the Entity Framework determine if our cached entity is current or obsolete based on the *original version* of the property value.

## How has it changed?

The merge action depends upon whether the entity was added, deleted, or changed since we set its original version. The entity's *EntityState* property tells us if and how it has changed.

## Is the entity represented in the data source?

If there is a data source entity that corresponds to the cached entity, we may use the data from data source entity to change the cached entity in some way.

If we don't find a matching data source entity, we have to decide what to do with the cached entity. Maybe someone deleted the data source entity in which case we might want to discard the cached entity. If we, on the other hand, we want to save the cached entity, we'll have to insert it into the data source rather than update the data source.

## Merging when the entity is in the data source

We'll look at each strategy and describe the outcome based on (a) whether or not the cached entity is current and (b) the entity's *EntityState*.

If the entity is *Unchanged*, we always replace both its original and current versions with data from the data source entity.

Our remaining choices are evident in the following table.

### Merge strategy consequences for a changed cached entity that exists in the data source.

Merge Strategy	Current	Added	Deleted	Detached	Modified	Post Current
PreserveChanges	Y	NC	NC	NC	NC	Y
	N	NC	NC	NC	NC	N
OverwriteChanges	Y or N	OW	OW	OW	OW	Y
PreserveChangesUnless		----	NC	NC	NC	Y
OriginalObsolete						
	N	OW	OW	OW	OW	Y
PreserveChangesUpdateOriginal		NC	NC	NC	NC	Y

- **NC** = No change; preserve the current version values of the cached entity
- **OW** = Overwrite the cached entity's current version values with data from the data source entity
- **Post Current** = 'Y' means the cached entity is "current" relative to the data source after the merge.

There are important artifacts not immediately observable from this table.

The entity's *EntityState* may change after the merge. It will be marked *Unmodified* after merge with *OverwriteChanges*. It will be marked *Unmodified* after merge with *PreserveChangesUnlessOriginalObsolete* if the entity is obsolete.

Note that deleted and detached entities are resurrected in both cases.

An added cached entity must be deemed “obsolete” if it already exists in the data source. The entity exists in the data source if the query returns an object with a matching primary key. If we think we created Employee with Id=3 and we fetch one with Id=3, someone beat us to it and used up that Id value. Our entity is obsolete. We will not be able to insert that entity into the data source; we’ll have to update the data source instead.

The *PreserveChangesUpdateOriginal* strategy enables us to force our changes into the data source even if the entity is obsolete. An added entity merged with *PreserveChangesUpdateOriginal* will be marked *Modified* so that DevForce knows to update the data source when saving it.

These effects are summarized in the following table:

#### EntityState after merge.

Merge Strategy	Current	Added	Deleted	Detached	Modified
<b>PreserveChanges</b>	Y or N	A	D	Dt	M
<b>OverwriteChanges</b>	Y or N	U	U	U	U
<b>PreserveChangesUnlessOriginalObsolete</b>		---	D	Dt	M
	N	U	U	U	U
<b>PreserveChangesUpdateOriginal</b>		M	D	Dt	M

- **A** = Added
- **D** = Deleted
- **Dt** = Detached
- **M** = Modified
- **U** = Unchanged

The merge may change the original version of a changed cached entity to match the data source values.

- *PreserveChanges* never touches the original version.
- The original version is always changed with the *OverwriteChanges* strategy.
- It is reset with the *PreserveChangesUnlessOriginalObsolete* strategy if (and only if) the entity is obsolete..
- *PreserveChangesUpdateOriginal* updates the original version (but not the current version!) if the entity is obsolete. This step ensures that the cached entity appears current while preserving the pending changes.

These effects are summarized in the following table:

#### Merge strategy effect on the original version of the cached entity.

Merge Strategy	Current	Added	Deleted	Detached	Modified
<b>PreserveChanges</b>	Y or N	NC	NC	NC	NC
<b>OverwriteChanges</b>	Y or N	OW	OW	OW	OW
<b>PreserveChangesUnlessOriginalObsolete</b>		---	NC	NC	NC
	N	OW	OW	OW	OW
<b>PreserveChangesUpdateOriginal</b>		OW	OW	OW	OW

#### Merging when the cached entity is not in the data source

We begin by considering cached entities that are unchanged. If the query applied to the cache returns an unchanged entity, ‘X’, and the query applied to the data source did not return its mate, we can safely assume that ‘X’ was deleted after we fetched it. We can remove ‘X’ from the cache.

We turn next to changed cached entities where we must distinguish between a query that tests only for the primary key and one that tests for something other than the primary key.

If the query tests for anything other than the primary key, we can draw no conclusions from the fact that a cached entity was not found in the database. For what does it mean if we have an employee named "Sue" in cache and we don't find her in the data source? Perhaps someone deleted her from the data source. Maybe someone merely renamed her. Maybe we renamed her. The combinations are too many to ponder.

On the other hand, if we query for Employee with Id = 3 and we don't find that employee in the data source, we can be confident of a simple interpretation. DevForce confirms that the primary key has not changed. While it is good practice to use immutable keys, it is not always so. If the primary key has been changed, DevForce leaves the cached entity alone. A business

object must have unique identity so if it isn't there, either it was never there or it has been deleted. What happens next depends upon the *EntityState* of the cached entity and the merge strategy.

- DevForce recovers gracefully when it attempts to save an entity marked for deletion and it can't find the data source entity to delete so the merge can leave this cached entity alone. It can also skip over the detached entities.
- *PreserveChanges* forbids merge effects on changed entities. The entity stays put in the cache.
- *OverwriteChanges* takes the data source as gospel. If the cached entity's *EntityState* is *Modified*, there should be an existing data source entity. There is not, so DevForce assumes the data source entity has been deleted and the cache should catch up with this reality. It removes the entity from the cache. Removal from the cache is just that. The entity disappears from cache and will not factor in a save. It does not mean "delete" which requires DevForce to try to delete the entity from the data source. It is an action neutral to the data source.  
On the other hand, if the cached entity is new (*Added*), we don't expect it to be in the data source. The entity remains "as is" in the cache, a candidate for insertion into the data source.
- *PreserveChangesUnlessOriginalObsolete* behaves just like *OverwriteChanges*.
- *PreserveChangesUpdateOriginal* strives to position the entity for a successful save. It must intervene to enable data source insertion of a modified entity by changing its *EntityState* to *Added*. An update would fail because there is no data source entity to update.

In summary:

**Merge strategy consequences for a changed cached entity that does not exist in the data source.**

Merge Strategy	Added	Modified
<b>PreserveChanges</b>	A	M
<b>OverwriteChanges</b>	A	R
<b>PreserveChangesUnlessOriginalObsolete</b>	A	R
<b>PreserveChangesUpdateOriginal</b>	A	A

- **A** = Added
- **M** = Modified
- **R** = Removed

### DataSourceOnly subtleties

We may get a nasty surprise if we use a *DataSourceOnly* or *DataSourceThenCache* query with other than the *OverwriteChanges* merge strategy. Consider the following queries using the *PreserveChanges* merge strategy.

Suppose we hold the "Nancy" employee in cache. We change her name to "Sue" and then search the database for all Employees with first names beginning with 'S'. We will not get "Sue" because she is still "Nancy" in the database.

Suppose we search again but this time we search for first names beginning with 'N'. This time we get "Sue". That will confuse the end user but it is technically correct because the "Sue" in cache is still "Nancy" in the database. *DataSourceThenCache* will produce the same anomaly for the same reason: the database query picks up the object in the database as "Nancy" but preserves the modification in cache which shows her as "Sue".