Contents

Anytime we save an entity in a multi-user distributed environment, there is the possibility that another user on another machine has saved their own version of the same entity since we last queried for our own copy of the entity. This page describes the mechanisms for **managing concurrency**.

The problem

A multi-user application must decide how to resolve the conflict when two users try to update the same entity. Consider the following:

1. I fetch the Employee with Id = 42
2. You fetch the Employee with Id = 42
3. You change and save your copy of the Employee with Id = 42
4. I try to save my copy of the Employee with Id = 42

Is this really going to happen?

There is always a risk that another client or component will change the data source entity while we are holding our cached copy of it. The risk grows the longer we wait between the time we fetch the entity and the time we save or refresh it. In offline scenarios, the time between fetch and update can be hours or days. There could be a great many concurrency conflicts waiting to happen.

If I save my copy now, should it overwrite the one you saved?

If so, we've chosen "last-in-wins" concurrency checking. My copy replaces your copy; your changes are lost.

This is the default in DevForce but we strongly recommend that you adopt another type of concurrency control. Permitting one user to blithely overwrite changes that another user made can be dangerous or even fatal.

There is an enormous body of literature on this subject available on the web by searching for "concurrency checking".

Basic mechanics of concurrency detection

DevForce defers to the ADO.NET Entity Framework's mechanism for detecting concurrency conflicts at the time an update is submitted to the back-end data source.

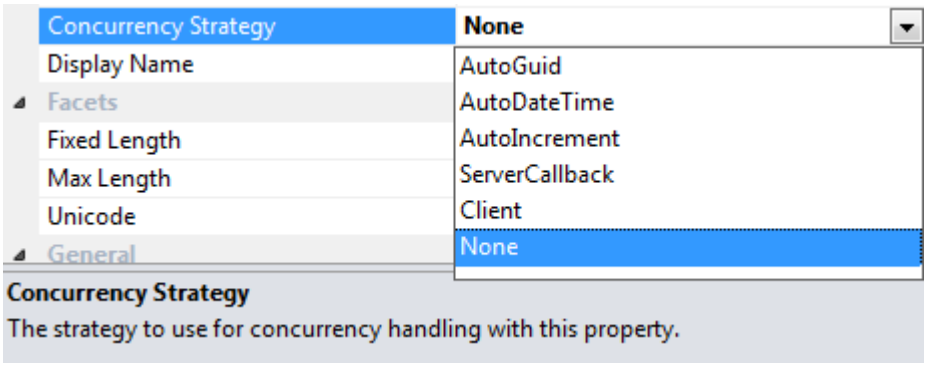
The Entity Framework (EF) permits the developer to designate, in the Entity Model, one or more entity properties as concurrency properties by setting its "Concurrency Mode" to "Fixed". Note that EF does not allow you to define a concurrency property within a Complex Type.

When a client application submits an update order against such a model to the EF, the EF prepares a SQL Update statement. To that statement it adds a WHERE clause that ensures that all columns designated as a concurrency columns have the same value they did when the record was last retrieved by the submitting application. (In other words, they have not been changed in the meantime by another user.) If that proves not to be the case, the exception thrown by the back-end data source will be propagated back down the application's calling chain.

This approach works only if the concurrency properties are updated with each successful save. Entity Framework handles concurrency violation *detection* but not concurrency property *update*. It is the developer's responsibility to ensure that the update happens.

You can perform the update yourself, tell DevForce to do it, or rely on the database to update the corresponding concurrency column itself (as it might do with an update trigger).

You tell DevForce which approach to take by setting each concurrency property's "Concurrency Strategy" in the EDM Designer.



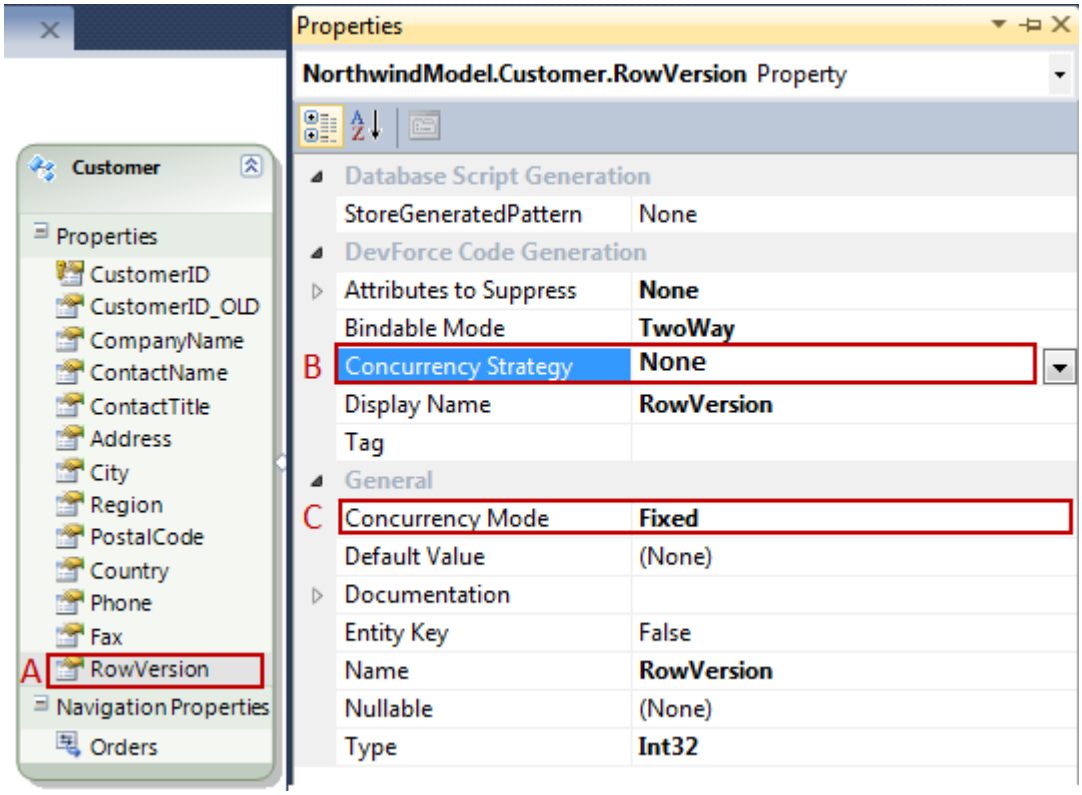
You can pick one of six concurrency property update strategies:

Strategy Name	Strategy for updating the concurrency property
AutoGuid	DevForce replaces the existing property value with a new GUID.
AutoDateTime	DevForce replaces the existing property value with the current Date/Time.
AutoIncrement	DevForce increments the existing property value by 1.
Server Callback	DevForce finds your custom class that implements the *IConcurrencyStrategy* interface and calls its *SetNewConcurrencyValue()* method, identifying the concurrency property and the entity-to-update in the parameters. Your implementation of *SetNewConcurrencyValue()* updates the property.
Client	Your custom code in the client application updates the property before sending modified entities to the server to be saved.
None	Neither you nor DevForce update the property. Appropriate when the data store automatically updates the concurrency column as it might do with a trigger.

Note that some of the strategies only apply to properties of specific types: clearly we cannot force a GUID value into an integer property, or a DateTime value into a boolean property, and so forth. It remains the developer's responsibility to handle any concurrency exception thrown by the back end.

Note that when using *AutoDateTime* DevForce will set the property value to DateTime.Now, which may have more precision than the data type in your database. If stored values are truncated you will need to set the EntityTypesRequiringPostSaveRefresh to avoid later concurrency exceptions.

Here's an example:



A. The *RowVersion* concurrency property of the *Customer* conceptual entity
B. The "Concurrency Strategy" DevForce uses to update the concurrency property
C. The "Concurrency Mode" (*Fixed*) tells Entity Framework that this is a concurrency property

The "Concurency Strategy" in this example is *None* meaning that we expect the database to update the *RowVersion* automatically, probably with an update trigger. Alternatively, DevForce would update it if we picked *AutoIncrement*.

Remember:
1. Set the Entity Framework "Concurrency Mode" to *Fixed* or nothing happens.

2. When "Concurrency Strategy" is *Auto…*, DevForce updates the property for you; otherwise, you handle update whether on server, client, or data tier.

One concurrency column, or many?

Since the Entity Framework permits you to designate any number of columns as concurrency columns, it may be tempting simply to designate them all. That's one way of making sure that, if anything in the record has been changed by another user since you got your copy, a concurrency conflict will be diagnosed.

This may be your only alternative if you have no design access to the database, but be aware that there will be a performance impact. Every update will be accompanied by a flurry of activity comparing values. As with other performance issues, you should do some upfront testing to determine whether the performance impact is unacceptable, or even significant.

If you do have design access to the database, or you're fortunate enough to inherit a database already designed the way you want it, it's generally a better alternative to provide a single column that is guaranteed to be updated whenever anything else is, and to use that as your sole determinant of a concurrency conflict. A simple integer column that is incremented each time the record is updated will do quite nicely; you can also use a GUID, timestamp, or any other type and methodology that guarantees that the value will change in a non-cyclical way. As you have seen, DevForce makes it easy for you to make a column auto-updating.

Pessimistic versus optimistic concurrency checking

There are two popular approaches to concurrency checking: pessimistic and optimistic.

In pessimistic concurrency, we ask the data source to lock the data source entity while we examine it. If we change it, we write it back to the data source. When we are done looking at it or updating it, we free the lock. While we have it locked, no one else can see or use it.

This approach holds up other users trying to reach the object we hold. It gets worse if we need many objects at once. There are potential deadlocks (I grab A, you grab B, I want B too, but can't get it, so I wait. You want A also, but can't get it, so you wait. We both wait forever).

There are more complicated, less draconian implementations to this approach but they amount to the same punishing performance.

Under optimistic concurrency, we don't lock the table row. We bet that no one will change the source data while we're working with it and confirm our bet when (and if) we try to update the data. The mechanism works as follows.

We fetch a copy of the table row and turn it into a business object. We work with this copy of the data source entity. We may decide to update the entity or mark it for deletion. When we save an altered entity, the business object server converts our intention into a data source management command. That command, in the process of updating or deleting the supporting table row, confirms that the row still exists and has not changed since we fetched it. If the row is missing or has changed, the command fails and it's up to the application to figure out what to do about it.

Changes are comparatively rare so we have reason to be optimistic that the row will be exactly as we last found it.

Only optimistic concurrency checking is performed in both DevForce and the Entity Framework.

Resolving concurrency collisions

Our optimism is usually rewarded. Occasional disappointment is inevitable. Eventually, we will encounter a conflict between our cached entity, with its pending changes, and the newly-updated data source entity.

We will want to resolve that conflict one way or the other. The possible resolutions include:

- Preserve the pending changes and ask the user what to do.
- Abandon the pending changes and re-fetch the entity.
- Arrange for the cached entity to become the current entity while preserving the pending changes
- Compare the cached entity with the current data source entity and merge the difference per some business rules or as guided by the user.

1st choice - Stop and ask

The first choice is the easiest place to start. We do nothing with the entity and report the problem to the user. The cached entity cannot be saved. We leave it up to the user to decide either to abandon the changes (option #2) or push them forward (options #2 and #3).

The remaining options involve re-fetching the entity from the data source. They differ in what they do with the entity retrieved – a difference determined by the **MergeStrategy** and how we use it.

```
aManager.RefetchEntity(anEntity, aMergeStrategy);
aManager.RefetchEntity(anEntity, aMergeStrategy)
```

2nd choice - Discard local changes and refresh from the datasource

The second choice uses the *OverwriteChanges* strategy to simply discard the user's changes and update the entity to reflect the one current in the datasource.

```
aManager.RefetchEntity(anEntity, MergeStrategy.OverwriteChanges);
aManager.RefetchEntity(anEntity, MergeStrategy.OverwriteChanges)
```

If you choose this option, you should explain this to the user before erasing her efforts. Note that in many scenarios, concurrency collisions occur so infrequently that this can be a very reasonable choice.

3rd choice - Save changes despite the concurrency violation

The third choice makes the cached entity current by re-fetching with the *PreserveChangesUpdateOriginal* strategy. This strategy causes the cached entity to override the current datasource entity.

```
aManager.RefetchEntity(anEntity, MergeStrategy.PreserveChangesUpdateOriginal);
aManager.RefetchEntity(anEntity, MergeStrategy.PreserveChangesUpdateOriginal)
```

The refetch replaces the cached entity's *original version* (see EntityState discussion) with the values from the current data source entity but it preserves the cached entity's *current version* values, thus retaining its pending changes.

You can access the original version of the values by going through the static PropertyMetadata methods. (You could also create a temporary *EntityManager* and query the database version of the entity into its cache.)

```
// the current value of the property in the cached entity
Employee.PropertyMetadata.FirstName.GetValue(anEmployee, EntityVersion.Current);
// the value from the datasource when most recently retrieved
Employee.PropertyMetadata.FirstName.GetValue(anEmployee, EntityVersion.Original);
```

```
' the current value of the property in the cached entity
Employee.PropertyMetadata.FirstName.GetValue(anEmployee, EntityVersion.Current)
' the value from the datasource when most recently retrieved
Employee.PropertyMetadata.FirstName.GetValue(anEmployee, EntityVersion.Original)
```

The effect is as if we had just read the entity from the datasource and applied the user's changes to it.

If we ask the *EntityManager* to save it now, the datasource will "think" that we modified the most recently saved copy of the entity and welcome the changed record.

This option is much like "last one wins" concurrency with a crucial difference: it was no accident. We detected the concurrency collision and forced the issue in accordance with approved business rules.

PreserveChangesUpdateOriginal strategy works only if the entity is governed by optimistic concurrency. If the entity lacks a concurrency column, the refetch uses the *OverwriteChanges* strategy instead.

Of course we wouldn't be talking about concurrency resolution if there were no concurrency columns.

4th choice - Merge local changes with refetched data

The fourth possibility begins, like the third, with a re-fetch governed by the *PreserveChangesUpdateOriginal* strategy. This time we don't forcibly save the cached entity.

We execute business logic instead which compares the current and original versions, column by column, deciding whether to keep the locally changed value (the "current" value) or the datasource value (now tucked inside the "original" value).

Such logic can determine if and when the cached entity's values should prevail. The logic may be entirely automatic. Alternative, the program could present both versions to the user and let her decide each difference.

Concurrency and the object graph

One interesting concurrency issue revolves around the scope of concurrency checking. Have I changed an *Order* if I add, change or delete one of its *OrderDetail* items? If I change the name of a customer, have I changed its orders?

These considerations have to do with concurrency control of the business object *graph*. DevForce does not support graph concurrency directly. DevForce supports single-table, "business object proper" concurrency control.

The developer can achieve the desired degree of graph concurrency control by employing single-table checking within a properly conceived, transactional concurrency plan.

It doesn't have to be wildly difficult. In brief, the developer adds custom business model code such that

- Added, changed, or deleted children entities always modify their parents.
- An application save attempt always includes the root entity of the dependency graph.
- During a save attempt, the root entity ensures that its children are included in the entity-save list.
- These children include their children.

Concurrency and mutually dependent entities

What if a bunch of entities are mutually dependent?

Suppose we have an order and its details. User 'A' adds two more details and changes the quantity on a third. She deletes the fourth detail and then saves.

In many applications, an order is never less than the sum of its parts. The order and every one of its details must be treated as a unit at least for transactional save purposes. We will describe this network of dependency as a "Dependency Graph".

Detection

Continuing our story, we see that User 'B' changed the fifth order detail and saved before User 'A' tried to save her changes.

User 'A' didn't touch the fifth order detail. She won't know about the change because there will be no concurrency conflict to detect; she can't detect a concurrency conflict unless she save the fifth order detail and she has no reason to do so.

If this worries you (it worries me), you may want to establish business rules that detect concurrency violations for any of entity in a dependency graph. A good approach is to

- Identify the root entity of the graph (Order) and
- Ensure that a change to any node in the graph (OrderDetail) causes the root entity to change.

User 'B's change to the fifth detail would have meant a change to the order. User 'A's changes also modified the order. User 'A's save attempt will provoke a concurrency violation on the root entity, the order.

Resolution

Now that User 'A' has learned about the violation, what can she do? There is no obvious problem. Neither 'A' nor 'B' changed the order entity itself so there are not differences to reconcile. There is only the tell-tale fact that their concurrency column values are different.

It doesn't seem proper to proceed blithely, ignoring the violation and proceeding as if nothing happened. User 'A' should suspect something is amiss in the details. The application should re-read all details, even those the user didn't change. It should look for differences at any point in the graph and only after applying the application-specific resolution rules should it permit the entire order to be saved again.

What are those resolution rules? We suggest taking the easiest way out if possible: the application should tell the User 'A' about the problem and then throw away her changes.

There must be something fundamentally wrong if two people are changing an order at the same time. In any case, the complexity of sorting out the conflict and the risk of making a total mess during "reconciliation" argue for a re-start.

If you can't take the easy way out – if you have to reconcile – here are a few pointers.

It is probably easiest to use a temporary second *EntityManager* for the analysis. A single *EntityManager* can only hold one instance of an entity at a time and we need to compare two instances of the same entity. This is manageable if there is only one entity to deal with – we've seen how to use the current and original versions within each entity to carry the difference information.

This trick falls apart when we are reconciling a dependency graph. Instead we'll put User 'A's cached order and its details in one manager and their dopplegangers from User 'B' in another.

The author thinks it is best to import User 'A's order and details into the *second* manager and put User 'B's version into the *main manager* by getting them with the *OverwriteChanges* strategy. This seems counter-intuitive but there are a couple of good reasons.

- We can *ImportEntities* into the second manager without logging it in. We'd have to log in the second manager before we could use it to get *GetEntities*. This is not hard, but avoiding it is even easier!
- The application probably should favor User 'B's order; if so that order will be in the main manager where it belongs.

Show some restraint

The order's entire object graph is not its dependency graph. The order may consist of details but that may be as far as it goes.

For example, every detail is associated with a product. If User 'B' changed the fifth detail's product name or its color, should this provoke a concurrency conflict? It User 'C' updated the order's customer name, does that mean all orders sold to that customer must be scrutinized for concurrency collisions.

Most businesses will say "no".