Handling Concurrency Conflicts

Article • 07/05/2023

> **Tip**
>
> You can view this article's **sample** on GitHub.

In most scenarios, databases are used concurrently by multiple application instances, each performing modifications to data independently of each other. When the same data gets modified at the same time, inconsistencies and data corruption can occur, e.g. when two clients modify different columns in the same row which are related in some way. This page discusses mechanisms for ensuring that your data stays consistent in the face of such concurrent changes.

Optimistic concurrency

EF Core implements *optimistic concurrency*, which assumes that concurrency conflicts are relatively rare. In contrast to *pessimistic* approaches - which lock data up-front and only then proceed to modify it - optimistic concurrency takes no locks, but arranges for the data modification to fail on save if the data has changed since it was queried. This concurrency failure is reported to the application, which deals with it accordingly, possibly by retrying the entire operation on the new data.

In EF Core, optimistic concurrency is implemented by configuring a property as a *concurrency token*. The concurrency token is loaded and tracked when an entity is queried - just like any other property. Then, when an update or delete operation is performed during SaveChanges(), the value of the concurrency token on the database is compared against the original value read by EF Core.

To understand how this works, let's assume we're on SQL Server, and define a typical Person entity type with a special `Version` property:

C#

```
public class Person
{
    public int PersonId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }

    [Timestamp]
    public byte[] Version { get; set; }
}
```

In SQL Server, this configures a concurrency token that automatically changes in the database every time the row is changed (more details are available below). With this configuration in place, let's examine what happens with a simple update operation:

C#

```
var person = context.People.Single(b => b.FirstName == "John");
person.FirstName = "Paul";
context.SaveChanges();
```

1. In the first step, a Person is loaded from the database; this includes the concurrency token, which is now tracked as usual by EF along with the rest of the properties.
2. The Person instance is then modified in some way - we change the `FirstName` property.
3. We then instruct EF Core to persist the modification. Since a concurrency token is configured, EF Core sends the following SQL to the database:

SQL

```
UPDATE [People] SET [FirstName] = @p0
WHERE [PersonId] = @p1 AND [Version] = @p2;
```

Note that in addition to the `PersonId` in the WHERE clause, EF Core has added a condition for `Version` as well; this only modifies the row if the `Version` column hasn't changed since the moment we queried it.

In the normal ("optimistic") case, no concurrent update occurs and the UPDATE completes successfully, modifying the row; the database reports to EF Core that one row was affected by the UPDATE, as expected. However, if a concurrent update occurred, the UPDATE fails to find any matching rows and reports that zero were affected. As a result, EF Core's SaveChanges() throws a DbUpdateConcurrencyException, which the application must catch and handle appropriately. Techniques for doing this are detailed below, under Resolving concurrency conflicts.

While the above examples discussed *updates* to existing entities. EF also throws DbUpdateConcurrencyException when attempting to *delete* a row that has been concurrently modified. However, this exception is never thrown when adding entities; while the database may indeed raise a unique constraint violation if rows with the same key are being inserted, this results in a provider-specific exception being thrown, and not DbUpdateConcurrencyException.

Native database-generated concurrency tokens

In the code above, we used the `[Timestamp]` attribute to map a property to a SQL Server rowversion column. Since `rowversion` automatically changes when the row is updated, it's very useful as a minimum-effort concurrency token that protects the entire row. Configuring a SQL Server `rowversion` column as a concurrency token is done as follows:

Data Annotations

c#

```
public class Person
{
    public int PersonId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }

    [Timestamp]
    public byte[] Version { get; set; }
}
```

The `rowversion` type shown above is a SQL Server-specific feature; the details on setting up an automatically-updating concurrency token differ across databases, and some databases don't support these at all (e.g. SQLite). Consult your provider documentation for the precise details.

Application-managed concurrency tokens

Rather than have the database manage the concurrency token automatically, you can manage it in application code. This allows using optimistic concurrency on databases - like SQLite - where no native automatically-updating type exists. But even on SQL Server, an application-managed concurrency token can provide fine-grained control on exactly which column changes cause the token to be regenerated. For example, you may have a property containing some cached or unimportant value, and don't want a change to that property to trigger a concurrency conflict.

The following configures a GUID property to be a concurrency token:

Data Annotations

```c#
public class Person
{
    public int PersonId { get; set; }
    public string FirstName { get; set; }

    [ConcurrencyCheck]
    public Guid Version { get; set; }
}
```

Since this property isn't database-generated, you must assign it in application whenever persisting changes:

```c#
var person = context.People.Single(b => b.FirstName == "John");
person.FirstName = "Paul";
person.Version = Guid.NewGuid();
context.SaveChanges();
```

If you want a new GUID value to always be assigned, you can do this via a SaveChanges interceptor. However, one advantage of manually managing the concurrency token is that you can control precisely when it gets regenerated, to avoid needless concurrency conflicts.

Resolving concurrency conflicts

Regardless of how your concurrency token is set up, to implement optimistic concurrency, your application must properly handle the case where a concurrency conflict occurs and DbUpdateConcurrencyException is thrown; this is called *resolving a concurrency conflict*.

One option is to simply inform the user that the update failed due to conflicting changes; the user can then load the new data and try again. Or if your application is performing an automated update, it can simply loop and retry immediately, after re-querying the data.

A more sophisticated way to resolve concurrency conflicts is to *merge* the pending changes with the new values in the database. The precise details of which values get merged depend on the application, and the process may be directed by a user interface, where both sets of values are displayed.

There are three sets of values available to help resolve a concurrency conflict:

- **Current values** are the values that the application was attempting to write to the database.
- **Original values** are the values that were originally retrieved from the database, before any edits were made.
- **Database values** are the values currently stored in the database.

The general approach to handle a concurrency conflicts is:

1. Catch `DbUpdateConcurrencyException` during `SaveChanges`.
2. Use `DbUpdateConcurrencyException.Entries` to prepare a new set of changes for the affected entities.
3. Refresh the original values of the concurrency token to reflect the current values in the database.
4. Retry the process until no conflicts occur.

In the following example, `Person.FirstName` and `Person.LastName` are set up as concurrency tokens. There is a `// TODO:` comment in the location where you include application specific logic to choose the value to be saved.

```C#
using var context = new PersonContext();
// Fetch a person from database and change phone number
```

```
var person = context.People.Single(p => p.PersonId == 1);
person.PhoneNumber = "555-555-5555";

// Change the person's name in the database to simulate a concurrency conflict
context.Database.ExecuteSqlRaw(
    "UPDATE dbo.People SET FirstName = 'Jane' WHERE PersonId = 1");

var saved = false;
while (!saved)
{
    try
    {
        // Attempt to save changes to the database
        context.SaveChanges();
        saved = true;
    }
    catch (DbUpdateConcurrencyException ex)
    {
        foreach (var entry in ex.Entries)
        {
            if (entry.Entity is Person)
            {
                var proposedValues = entry.CurrentValues;
                var databaseValues = entry.GetDatabaseValues();

                foreach (var property in proposedValues.Properties)
                {
                    var proposedValue = proposedValues[property];
                    var databaseValue = databaseValues[property];

                    // TODO: decide which value should be written to database
                    // proposedValues[property] = <value to be saved>;
                }

                // Refresh original values to bypass next concurrency check
                entry.OriginalValues.SetValues(databaseValues);
            }
            else
            {
                throw new NotSupportedException(
                    "Don't know how to handle concurrency conflicts for "
                    + entry.Metadata.Name);
            }
        }
    }
}
```

Using isolation levels for concurrency control

Optimistic concurrency via concurrency tokens isn't the only way to ensure that data stays consistent in the face of concurrent changes.

One mechanism to ensure consistency is the *repeatable reads* transaction isolation level. In most databases, this level guarantees that a transaction sees data in the database as it was when the transaction started, without being affected by any subsequent concurrent activity. Taking our basic sample from above, when we query for the `Person` in order to update it in some way, the database must make sure no other transactions interfere with that database row until the transaction completes. Depending on your database implementation, this happens in one of two ways:

1. When the row is queried, your transaction takes a shared lock on it. Any external transaction attempting to update the row will block until your transaction completes. This is a form of pessimistic locking, and is implemented by the SQL Server "repeatable read" isolation level.
2. Rather than locking, the database allows the external transaction to update the row, but when your own transaction attempts to do the update it, a "serialization" error will be raised, indicating that a concurrency conflict occurred. This is a form of optimistic locking - not unlike EF's concurrency token feature - and is

implemented by the SQL Server snapshot isolation level, as well as by the PostgreSQL repeatable reads isolation level.

Note that the "serializable" isolation level provides the same guarantees as repeatable read (and adds additional ones), so it functions in the same way with respect to the above.

Using a higher isolation level to manage concurrency conflicts is simpler, doesn't require concurrency tokens, and provides other advantages; for example, repeatable reads guarantee that your transaction always sees the same data across queries inside the transaction, avoiding inconsistencies. However, this approach does have its drawbacks.

First, if your database implementation uses locking to implement the isolation level, then other transactions attempting to modify the same row must block for the entirety of the transaction. This could have an adverse effect on concurrent performance (keep your transaction short!), although note that EF's mechanism throws an exception and forces you to retry instead, which also has an impact. This applies to the SQL Server repeatable read level, but not to the snapshot level, which does not lock queried rows.

More importantly, this approach requires a transaction to span all the operations. If you, say, query `Person` in order to display its details to a user, and then wait for the user to make changes, then the transaction must stay alive for a potentially long time, which should be avoided in most cases. As a result, this mechanism is usually appropriate when all contained operations executed immediately and the transaction doesn't depend on external inputs which may increase its duration.

Additional resources

See Conflict detection in EF Core for an ASP.NET Core sample with conflict detection.