Dist Sys. Optimistic Concurrency Control

We know by now that Entity Framework will handle race conditions and helps our databases to manage concurrent connections. However, this doesn't prevent us from accidentally getting into a lost update, inconsistent retrieval or dirty read scenario (see the lectures).

We have two options available to us: introduce pessimistic concurrency control (through the use of locking in our code) or modify our database to allow us to use optimistic concurrency control. It often makes sense to allow optimistic concurrency control – especially where the likelihood of a conflict is low.

Entity Framework is designed to work well with optimistic concurrency control but we still have to allow for it in our database tables.

We'll be basing this on the Entity Framework code we wrote in a previous lab. We'll be using some principles discussed in that workshop too so make sure you've done that lab already.

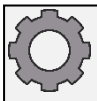
Download the Skeleton Solution from Canvas.

Open the NuGet Package Manager and install Microsoft.EntityFrameworkCore, Microsoft.EntityFrameworkCore.Tools and Microsoft.EntityFrameworkCore.SqlServer if they aren't already installed

Open the Package Manager Console window and type: **Update-Database**

Look at the database and solution. The code/database links Person, Address and BankAccount so that Addresses can have many People and each person can have one BankAccount.

Open EntryPoint.cs and note the similarities from the EF lab, and the addition of a BankAccount.

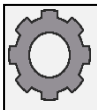


Press F5 to run the program. It lay take some time to load, then run.

Once execution is finished your database should now be populated with the details of 'Jane Doe'. We are going to be using Jane's table entries but we don't want to create any more copies of this data in the database so…

comment out the entire using block in EntryPoint.cs

What we're going to do next is access Jane's BankAccount twice, simultaneously trying to update its balance – but, in order to force the worst case scenario, we're going to manipulate it manually by using two processes.



Add a new using block to the entrypoint:

```
using (var ctx = new Context())
{ }
```

We'll be using Linq so add: `using System.Linq;` to your using statements

Inside your new using block, insert the following code:

```
Person prsn = ctx.People.First();
decimal balance = prsn.BankAccount.Balance;

decimal balancechange = 0;

do
{
    Console.WriteLine("Enter a balance modifier");
}
while (!decimal.TryParse(Console.ReadLine(), out balancechange));

balance += balancechange;

prsn.BankAccount.Balance = balance;
ctx.SaveChanges();
```

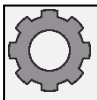
This code takes the first person in your database and loads up their balance. It then asks the user to give a balance modifier, applies this modifier to the balance and finally updates the database.

Press f5 to test your code is working

You should see an exception. Look at the locals for prsn - you should see that Address and BankAccount are both **Null**. If you examine the data in the database you'll see the relationships exist there! What do you think happened?

The problem is that Entity Framework hasn't loaded in the references for the linked objects. You are effectively working with Lazy Loading turned off. We have a few ways to solve this.

1. Explicit loading



After the line: `Person prsn = ctx.People.First();` add the new line:

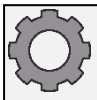
```
ctx.Entry(prsn).Reference("BankAccount").Load();
```

Now place a breakpoint on the line: `decimal balancechange = 0;`
and run the code with f5

When the breakpoint is hit, examine the balance variable – it should be 50. Now look at the prsn object. It should have BankAccount loaded but Addresses not loaded. You have explicitly loaded the BankAccount reference using the line you added.

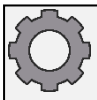
2. Lazy Loading

Let's turn Lazy loading on.



Open the NuGet Package Manager and install **Microsoft.EntityFrameworkCore.Proxies**

Inside Context.cs find the OnConfiguring method and add, at the top of the method, the line: `optionsBuilder.UseLazyLoadingProxies();`



Remove the new line you added previously:
`ctx.Entry(prsn).Reference("BankAccount").Load();`

Open Person.cs and add the `virtual` keyword before the `Address` and `BankAccount` properties. Open Address.cs and do the same before `People`.

This allows Entity Framework to override your properties at runtime, thus giving you access to objects through the Lazy Loading mechanism.

Ensure you still have a breakpoint on the line: `decimal balancechange = 0;`
and run the code with f5

When the breakpoint is hit, examine the balance variable – it should be 50. Now look at the prsn object. It should have BankAccount **and** Addresses loaded. You haven't had to explicitly load the BankAccount reference – Entity Framework has done this for you when you got the entry.

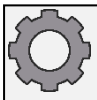
Remove the breakpoint and press f5 again to continue past it.

When prompted, enter -20 into the console window and hit enter. The app should finish execution after a few moments.

This is good… and bad.

It's easy, and just works out of the box… but we have loaded Address and we aren't actually going to use it, so it was a wasteful load.

If we had lots of data associated with a Person that we didn't actually need to load, we have now automatically loaded all of it when we loaded in the Person - which is likely to have taken quite a while and used up machine resources that could have been used for other things.



Now look at your BankAccount database table.

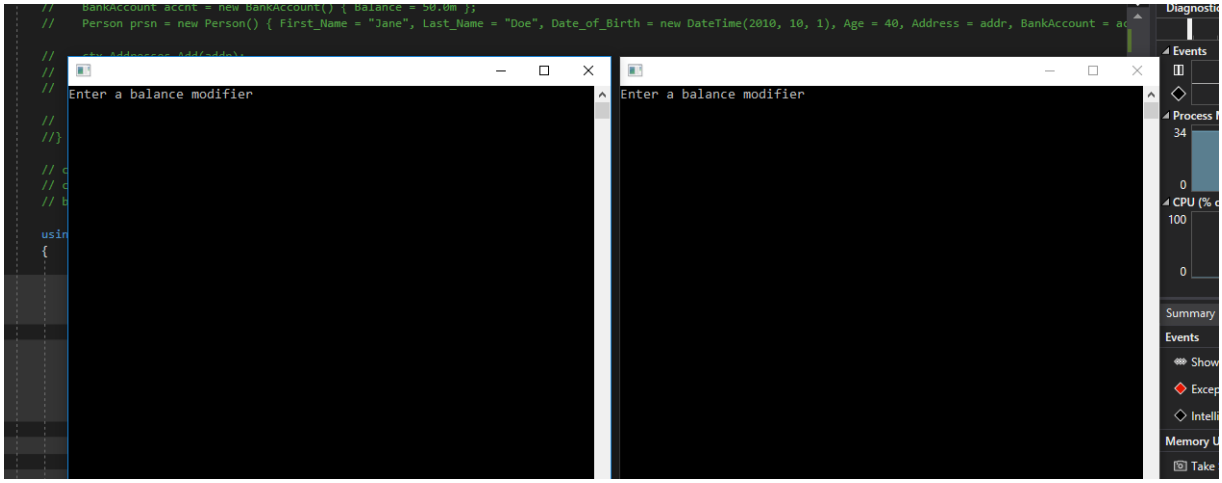
Our code should have subtracted 20 from the balance, resulting in a balance of 30

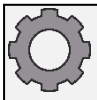
Now we'll simulate two concurrent requests:

Right-Click on the project, and then click Debug->Start New Instance to run the code.

Once an instance is running, do the same thing again to start another instance.

You should now have two consoles open, side by side, looking like this:





Enter 60 into one console window (and press enter)

Then enter -70 into the other console window (and press enter)

As our original balance was 30 (was 50-20), in theory this should perform the calculation:

30 + 60 - 70 = 20

Open up your BankAccount database table (SQL Server Object Explorer) and examine the Balance. It probably reads -40.00

Can you identify why?

The key thing here is that our calls were offset. If our application was multithreaded, this could actually have occurred on different concurrent threads and it may be very difficult to diagnose.

The process was:

1. Variable a = balance from database = 30
2. Variable b = balance from database = 30

3. a + 60 = 90
4. b - 70 = -40

5. Save a to database - Balance = 90
6. Save b to database - Balance = -40

We need a concurrency control mechanism and, as we've established, Entity Framework is set up to work best with Optimistic Concurrency Control.

However, in order to set it up, we have to make some changes to our code.

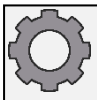
Open Person.cs and add the lines:

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

Then do the same in Address.cs and BankAccount.cs

The Timestamp attribute classifies this parameter (column in the db) as a row version identifier. This allows Entity Framework to automatically begin using this parameter to identify if conflicts have occurred.

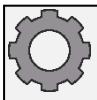
Because we've now changed our data model we'll need to update the database…



Add a new migration.

Check the migration code is adding three new rows, one in each table.

Update the database to the new migration



Now we'll simulate two concurrent requests again:

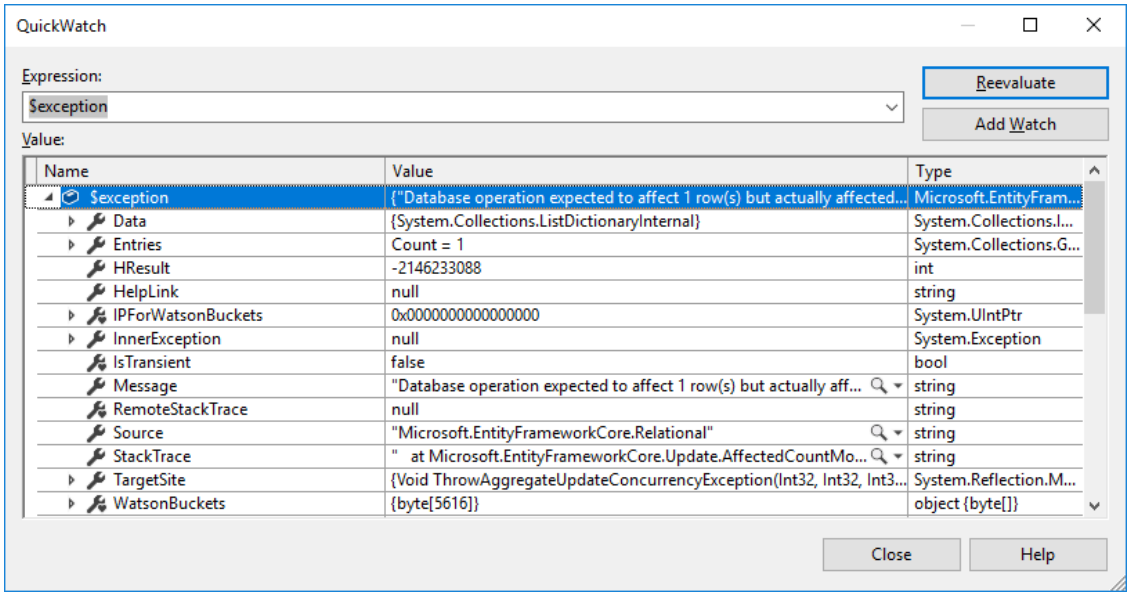
Right-Click on the project, and then click Debug->Start New Instance to run the code.

Once an instance is running, do the same thing again to start another instance.

In one console window type 90 (and press enter).

Then enter -50 into the other console window (and press enter)

You should now be staring at a nifty little exception – and (unusually) you should actually be very happy to see this exception. It should look something like this:



A quick read through of the exception identifies a few things that are useful to us.

1. The exception is a **ConcurrencyException**. It tells us that there has been a problem with the database operation and that the database didn't affect any rows. Of particular importance is the line: "Data may have been modified or deleted since entities were loaded".
2. Specifically, the exception is a DbUpdateConcurrencyException. This allows us to identify that this problem occurred explicitly when an update was attempted.

Under the hood, EntityFramework is using our RowVersion column… whenever an update is made to the row it checks the RowVersion hasn't changed since the read was done and, if it hasn't changed it automatically updates the byte array in the RowVersion column to a new value.

If an update is attempted but the RowVersion has changed since a read was done this exception is thrown to prevent data errors. We can catch the exception to allow recovery. In this case we'll throw it back to the client but what you do in other applications is up to you…

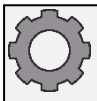
Replace the entire using block in your entrypoint with this code:

```
using (var ctx = new Context())
{
    bool success = false;
    while(!success)
    {
        Person prsn = ctx.People.First();
        decimal balance = prsn.BankAccount.Balance;

        decimal balancechange = 0;

        do
        {
            Console.WriteLine("Enter a balance modifier");
        }
        while (!decimal.TryParse(Console.ReadLine(), out balancechange));

        balance += balancechange;

        prsn.BankAccount.Balance = balance;
        try
        {
            ctx.SaveChanges();
            success = true;
        }
        catch (DbUpdateConcurrencyException ex)
        {
            var entry = ex.Entries[0];
            var d = entry.GetDatabaseValues();
            entry.OriginalValues.SetValues(d);
            entry.CurrentValues.SetValues(d);
            Console.WriteLine("Sorry. That didn't work. Try again.");
        }
    }
}
```

This code will catch an update concurrency exception and reset the OriginalValues (data originally read from database) and CurrentValues (current changes to data) with the new values from the database.

As soon as the Save succeeds it escapes the loop and closes the application.

Let's test if it works:



We'll simulate two concurrent requests again:

Right-Click on the project, and then click Debug->Start New Instance to run the code.

Once an instance is running, do the same thing again to start another instance.

1. In one console window type 20 (and press enter).
2. Then enter 30 into the other console window (and press enter)
3. You should be prompted to try again. **Start another instance.**
4. In this new console window type 20 (and press enter).
5. Re-enter 30 into the remaining console window (and press enter)
6. You should be prompted to try again. Re-enter 30 into the remaining console window (and press enter)

Check your database. Your balance should be 120 (50 + 20 + 20 + 30 = 120).
If so, you are successfully using optimistic concurrency control.

2 – Further Work

1. Now you understand the concept, implement similar functionality without having to make use of the [TimeStamp] keyword – this will test whether you fully understand how OCC works. Consider implementing a solution as below:

Once OCC identifies a possible issue, it would be very useful to assess what was actually changed on the row to determine if a conflict has occurred.

You can do something similar by encoding, in the version field, a unique version number for each field in the row.

For example, just using integers as version numbers for the rows in a table with **Name**, **Age** and **HairColour** columns, the Version column could be: "5|1|4|5". This can be interpreted as:

5: Overall row Version Number
1: Name field Version Number
4: Age field Version Number
5: HairColour field Version Number

If the overall version has not changed, the change can be made in the database, otherwise the system can check the fields that have been modified.

If the field(s) we are concerned with did not change between overall versions, the change can be made safely.

An example of this case would be that:

- Process 1 changed Age and updated Overall row Version from 5 to 6, and Age field Version from 4 to 5.
- Process 2 tried to change Hair Colour but saw that the Overall row Version had been changed from 5 to 6 since it read. Therefore it inspects the HairColour field Version Number and finds that it has not been updated so it can safely update it. It also changes the HairColour field Version from 5 to 6 and changes the Overall row Version from 6 to 7.

In the last (and most unlikely) case that the field we are interested in DID change, we can ask for a retry.

This is **very** useful for databases with tables that have lots of columns. If the rows in these tables are being updated very regularly but the updates don't usually affect many of the columns, this can make sure OCC is still an efficient option.