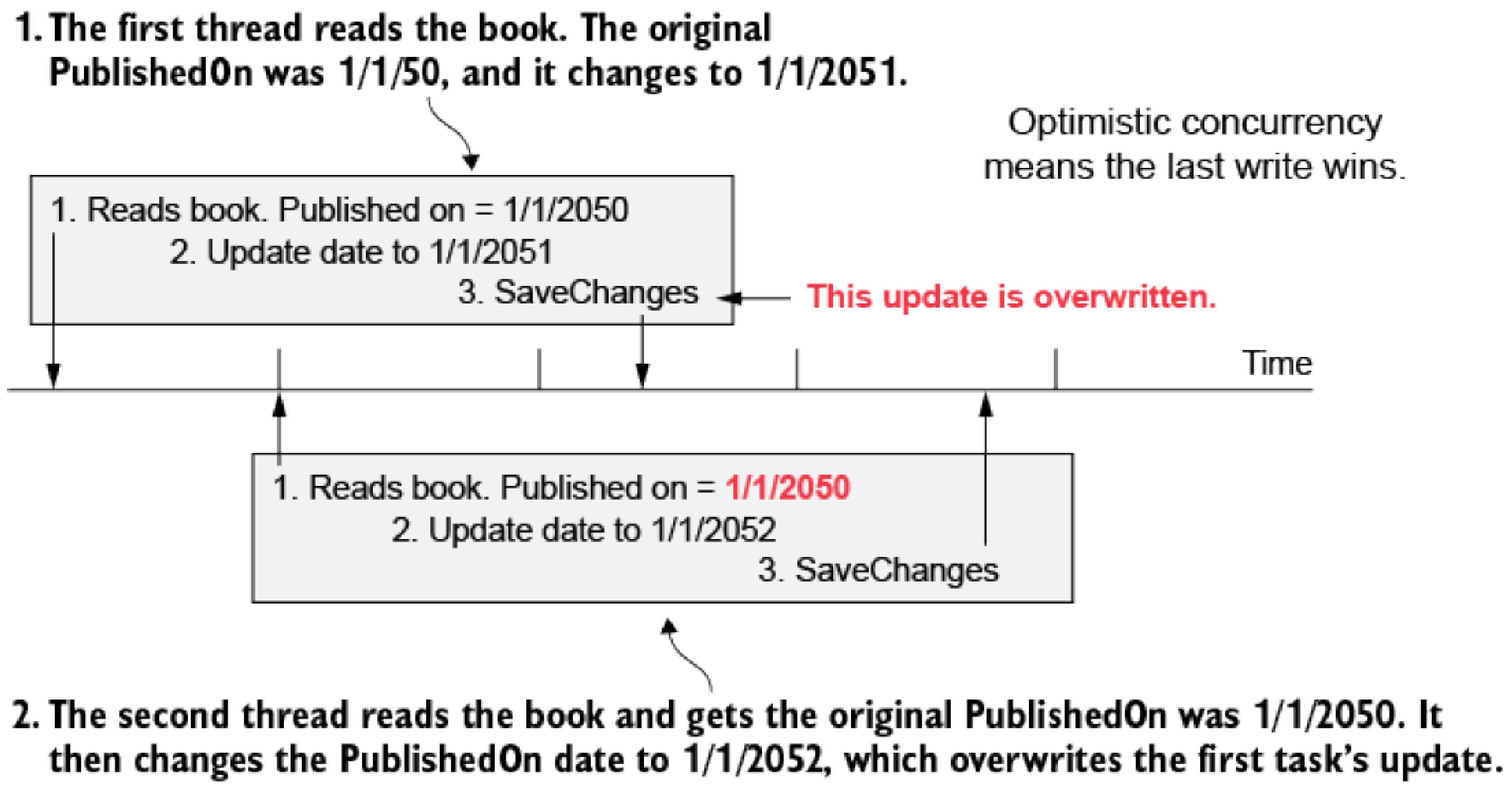
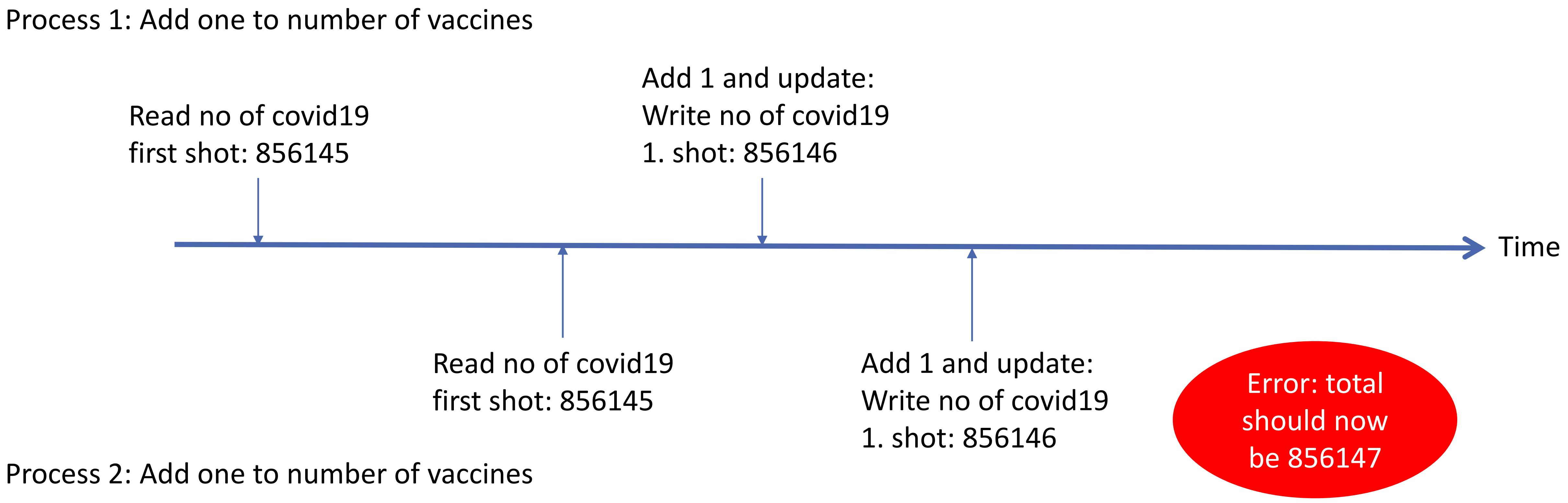
EF Core
Handling simultaneous updates

Concurrency conflicts


1. The first thread reads the book. The original PublishedOn was 1/1/50, and it changes to 1/1/2051.
Optimistic concurrency means the last write wins.
1. Reads book. Published on = 1/1/2050
2. Update date to 1/1/2051
3. SaveChanges
This update is overwritten.
Time
1. Reads book. Published on = 1/1/2050
2. Update date to 1/1/2052
3. SaveChanges
2. The second thread reads the book and gets the original PublishedOn was 1/1/2050. It then changes the PublishedOn date to 1/1/2052, which overwrites the first task's update.

Concurrency conflicts

- The “last write win” rule may not be useful in all cases.


Process 1: Add one to number of vaccines
Read no of covid19 first shot: 856145
Add 1 and update: Write no of covid19 1. shot: 856146
Time
Read no of covid19 first shot: 856145
Add 1 and update: Write no of covid19 1. shot: 856146
Error: total should now be 856147
Process 2: Add one to number of vaccines

Solving concurrency conflicts by design

- In an order-processing system that uses background tasks, which may cause concurrent conflicts.
- Remove the possibility of concurrent updates:
  - Split the customer order information into an immutable order part that never changes.
  - This contains data, such as what was ordered and where should it be sent.
- For the changing parts of the order, such as status create a separate table in which you add each new order status as it occurs, with the date and time
  - **This approach is known as *event sourcing***.
- You can then get the latest order status by sorting them by date/time order and picking the status with the newest date and time.

Pessimistic concurrency

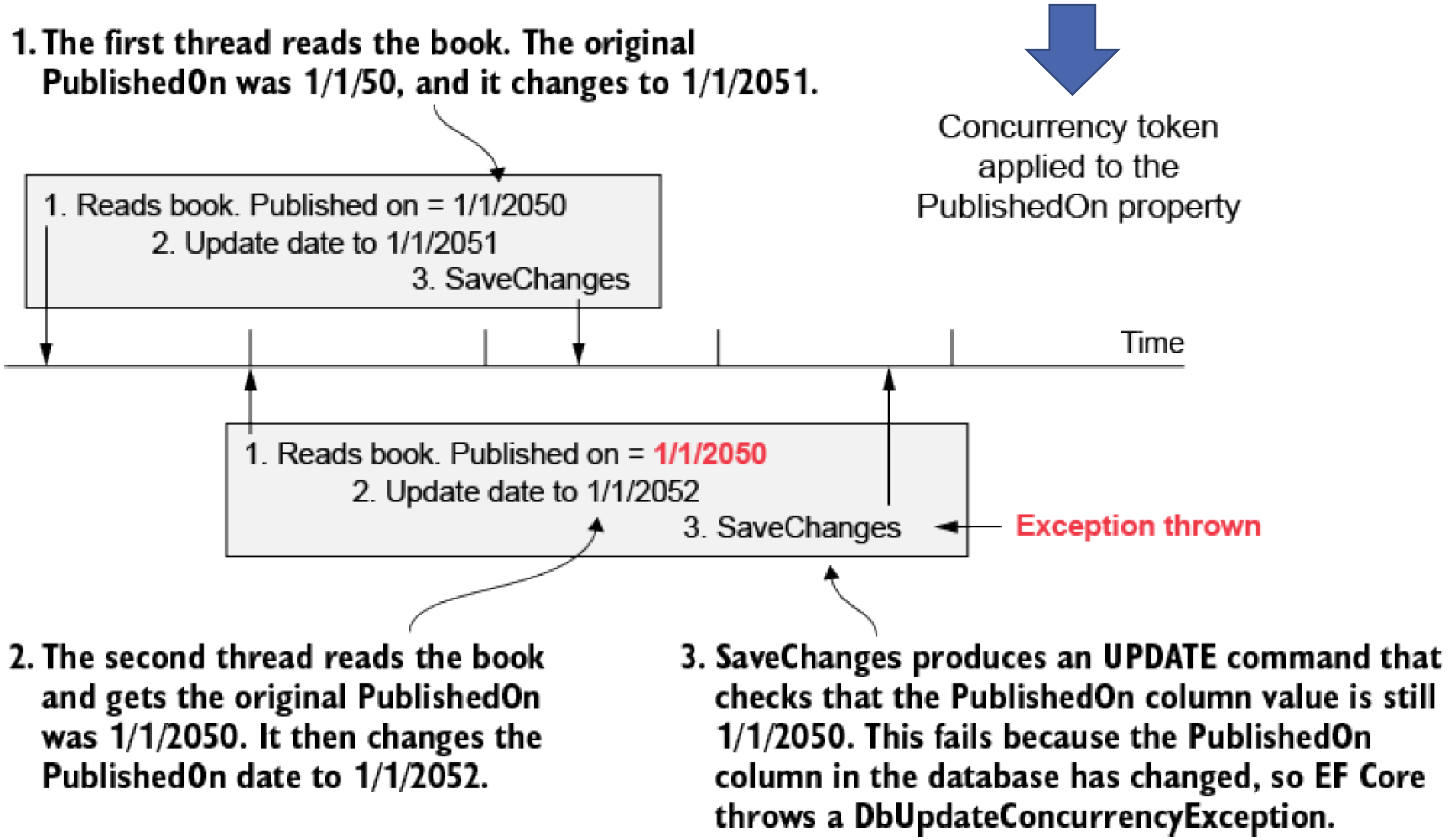
- One way to prevent accidental data loss in concurrency scenarios is to use database locks.
  - This is called pessimistic concurrency.
- Before you read a row from a database, you request a lock for read-only or for update access.
  - If you lock a row for update access, no other users are allowed to lock the row either for read-only or update access.
  - If you lock a row for read-only access, others can also lock it for read-only access but not for update.
- **Entity Framework Core provides no built-in support for pessimistic concurrency.**

EF Core's concurrency conflict–handling features

EF Core provides two ways of detecting a concurrent update and, once detected, a way of getting at all the relevant data so you can implement code to fix the issue.

- **Concurrency token**
  - to mark a specific property/column in your entity class as one to check for a concurrency conflict.
- **Timestamp** (also known as *rowversion*)
  - which marks a whole entity class/row as one to check for a concurrency conflict.
- In both cases, when **SaveChanges** is called, EF Core produces database server code to check updates of any entities that contain concurrency tokens or timestamps.
  - If that code detects that the concurrency tokens or timestamps have changed since it read the entity, it throws a **DbUpdateConcurrencyException** exception.

Detecting a concurrent change via concurrency token


1. The first thread reads the book. The original PublishedOn was 1/1/50, and it changes to 1/1/2051.
Concurrency token applied to the PublishedOn property
1. Reads book. Published on = 1/1/2050
2. Update date to 1/1/2051
3. SaveChanges
Time
1. Reads book. Published on = 1/1/2050
2. Update date to 1/1/2052
3. SaveChanges
Exception thrown
2. The second thread reads the book and gets the original PublishedOn was 1/1/2050. It then changes the PublishedOn date to 1/1/2052.
3. SaveChanges produces an UPDATE command that checks that the PublishedOn column value is still 1/1/2050. This fails because the PublishedOn column in the database has changed, so EF Core throws a DbUpdateConcurrencyException.

How to setup a concurrency token

- To configure a property as a concurrency token, you add the ConcurrencyCheck data annotation to the relevant property in our entity class, e.g.:

```
public class Book
{
  public int BookId { get; set; }
  public string Title { get; set; }
  [ConcurrencyCheck]
  public DateTime PublishedOn { get; set; }
  public Author Author { get; set; }
}
```

Define a concurrency token via the Fluent API

- Setting a property as a concurrency token using the Fluent API

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
  modelBuilder.Entity<Book>()
    .Property(p => p.PublishedOn)
    .IsConcurrencyToken();
  //… other configurations
}
```

Simulate a concurrent update by running an SQL command

The SQL command represents another thread of the web application, or another application that has access to the same database, updating the PublishedOn column.

```
var firstBook = context.Books.First();
context.Database.ExecuteSqlRaw(
    "UPDATE dbo.Books SET PublishedOn = GETDATE()"+
    " WHERE ConcurrencyBookId = @p0",
    firstBook.ConcurrencyBookId);
firstBook.Title = Guid.NewGuid().ToString();
context.SaveChanges();
```

SaveChanges will throw DbUpdateConcurrencyException

How it works

- The SQL that EF Core produces to update the Title – the code from previous slide:

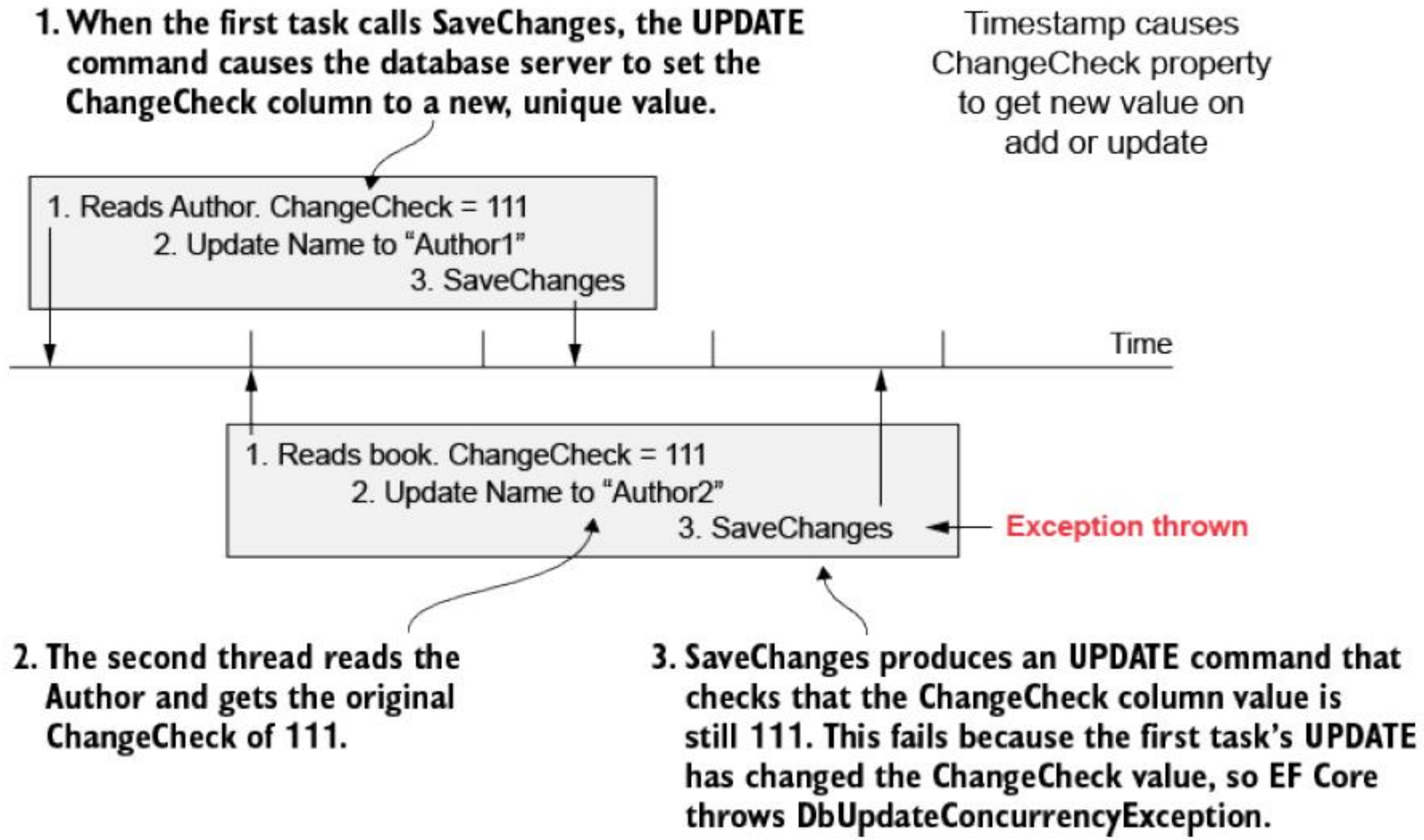
```
SET NOCOUNT ON;
UPDATE [Books] SET [Title] = @p0
WHERE [ConcurrencyBookId] = @p1
AND [PublishedOn] = @p2;
SELECT @@ROWCOUNT;
```

- When EF Core runs this SQL command, the WHERE clause will find a valid row to update only if the PublishedOn column hasn't changed from the value EF Core read in from the database.
- EF Core then checks the number of rows that have been updated by the SQL command.
- If the number of rows updated is zero, EF Core raises DbUpdateConcurrencyException to say that a concurrency conflict exists.

Detecting a concurrent change via timestamp

- Uses a unique value provided by the database server that’s changed whenever a row is inserted or updated.
- The whole entity is protected against concurrency changes, rather than specific properties/columns as with the concurrency token.

Detecting a concurrent change via timestamp


1. When the first task calls SaveChanges, the UPDATE command causes the database server to set the ChangeCheck column to a new, unique value.
Timestamp causes ChangeCheck property to get new value on add or update
1. Reads Author. ChangeCheck = 111
2. Update Name to "Author1"
3. SaveChanges
Time
1. Reads book. ChangeCheck = 111
2. Update Name to "Author2"
3. SaveChanges
Exception thrown
2. The second thread reads the Author and gets the original ChangeCheck of 111.
3. SaveChanges produces an UPDATE command that checks that the ChangeCheck column value is still 111. This fails because the first task's UPDATE has changed the ChangeCheck value, so EF Core throws DbUpdateConcurrencyException.

Timestamp type

- The timestamp database type is database type specific:
  - SQL Server’s concurrency type is ROWVERSION which maps to byte[] in NET
  - PostgreSQL has a column called xmin which is a unsigned 32 bit number
  - Cosmos DB has a json property called _etag which is a string containing a unique value.
- EF Core can use any of these via the appropriate database provider.

How to setup a timestamp

- Add a ChangeCheck property, which will watch for any updates to the whole entity, to the entity class.
  - If you use MS SQL server the type must be `byte[]`
- Give the ChangeCheck property a Timestamp data annotation.
  - This tells EF Core to mark this as a special column that the database will update with a unique value.

```
public class Author
{
    public int AuthorId { get; set; }
    public string Name { get; set; }
    [Timestamp]
    public byte[] ChangeCheck { get; set; }
}
```

Alternatively, you can use the Fluent API to configure a timestamp

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Author>()
        .Property(p => p.ChangeCheck)
        .IsRowVersion();
}
```

The SQL to create the Authors table, with a timestamp column

```
CREATE TABLE [dbo].[Authors] (
    [AuthorId] INT IDENTITY (1, 1),
    [ChangeCheck] TIMESTAMP NULL,
    [Name] NVARCHAR (MAX) NULL
);
```

Simulating a concurrent update

The SQL command represents another thread of the web application, or another application that has access to the same database, updating the same entity.

```
var firstAuthor = context.Authors.First();
context.Database.ExecuteSqlRaw(
    "UPDATE dbo.Authors SET Name = @p0"+
    " WHERE AuthorId = @p1",
    firstAuthor.Name,
    firstAuthor.AuthorId);
firstAuthor.Name = "New Name";
context.SaveChanges();
```

SaveChanges will throw DbUpdateConcurrencyException

AARHUS UNIVERSITY
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

How it works

- The SQL that EF Core produces to update the Name – the code from previous slide:

```
SET NOCOUNT ON;
UPDATE [Authors] SET [Name] = @p0
WHERE [AuthorId] = @p1
    AND [ChangeCheck] = @p2;
SELECT [ChangeCheck]
FROM [Authors]
WHERE @@ROWCOUNT = 1
    AND [AuthorId] = @p1;
```

Because the update will change the ChangeCheck column, EF Core needs to read it back so its in-memory copy is correct.

The check that the ChangeCheck column is the same as the value EF Core read in.

Checks that one row was updated in the last command. If not, it won't return the ChangeCheck value and EF Core will know that a concurrent change has taken place.

Choosing between the two approaches

- The concurrency token approach
  - A specific protection of the property/properties you place it on.
  - Works on any database
    - Because it uses basic commands.
- The timestamp approach
  - Catches any update to that entity.
  - Relies on a database server-side feature
    - Different type for different database types

Handling a DbUpdateConcurrencyException

- The way you write your code to fix a concurrency conflict depends on your business reasons for capturing it.

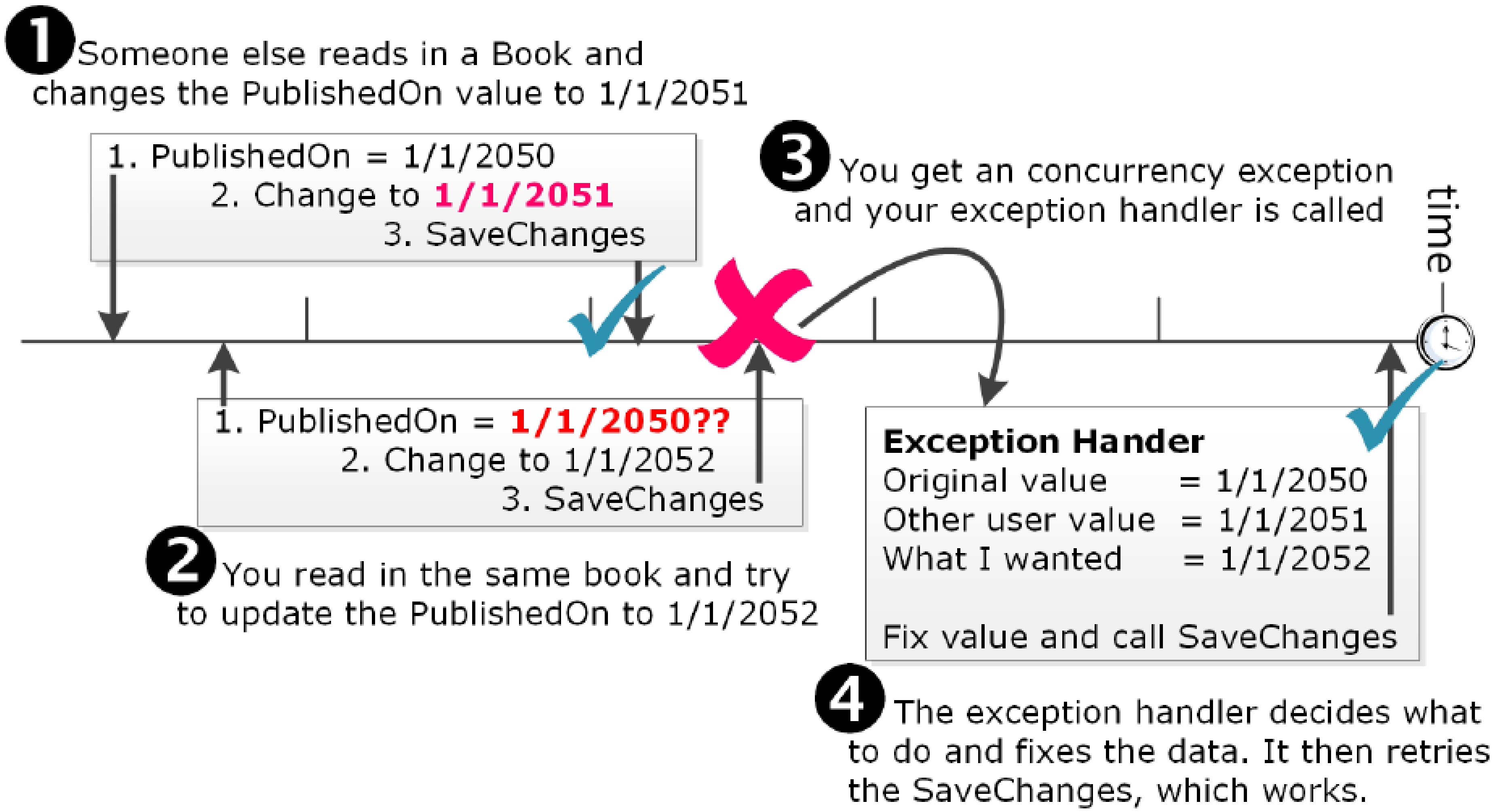
```
public static string BookSaveChangesWithChecks (ConcurrencyDbContext context)
{
    string error = null;
    try
    {
        context.SaveChanges();
    }
    catch (DbUpdateConcurrencyException ex)
    {
        var entry = ex.Entries.Single();
        error = HandleBookConcurrency(context, entry);
        if (error == null)
            context.SaveChanges();
    }
    return error;
}
```

Se next slide

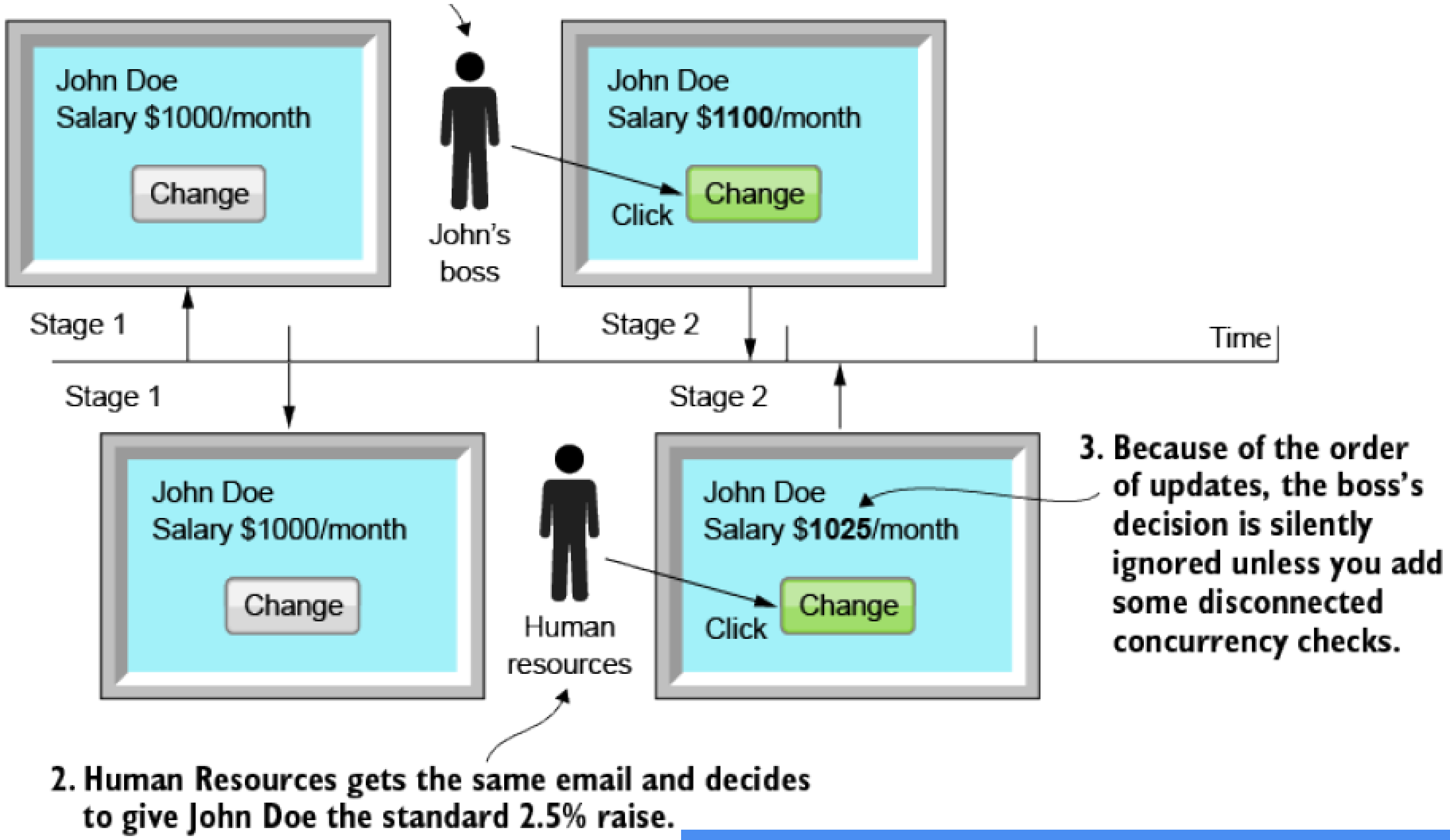
Assumes that a second concurrency issue isn't thrown

```
private static string HandleBookConcurrency(DbContext context, EntityEntry entry)
{
  var book = entry.Entity as Book;
  if (book == null)
    throw new NotSupportedException("Don't know how to handle concurrency conflicts for " +
                                   entry.Metadata.Name);
  var whatTheDatabaseHasNow = context.Set<Book>().AsNoTracking()
    .SingleOrDefault(p => p. BookId == book.BookId);
  if (whatTheDatabaseHasNow == null)
    return "Unable to save changes.The book was deleted by another user.";
  var otherUserData = context.Entry(whatTheDatabaseHasNow);
  foreach (var property in entry.Metadata.GetProperties())
  {
    var theOriginalValue = entry.Property(property.Name).OriginalValue;
    var otherUserValue = otherUserData.Property(property.Name).CurrentValue;
    var whatIWantedItToBe = entry.Property(property.Name).CurrentValue;
    // TODO: Logic to decide which value should be written to database
    if (property.Name == nameof(Book.PublishedOn))
    {
      entry.Property(property.Name).CurrentValue =
        //… your code to pick which PublishedOn to use
    }
    entry.Property(property.Name).OriginalValue =
      otherUserData.Property(property.Name)
        .CurrentValue;
  }
  return null;
}
```

Handling a DbUpdateConcurrencyException


1 Someone else reads in a Book and changes the PublishedOn value to 1/1/2051
1. PublishedOn = 1/1/2050
2. Change to 1/1/2051
3. SaveChanges
3 You get an concurrency exception and your exception handler is called
time
1. PublishedOn = 1/1/2050??
2. Change to 1/1/2052
3. SaveChanges
2 You read in the same book and try to update the PublishedOn to 1/1/2052
Exception Hander
Original value = 1/1/2050
Other user value = 1/1/2051
What I wanted = 1/1/2052
Fix value and call SaveChanges
4 The exception handler decides what to do and fixes the data. It then retries the SaveChanges, which works.

The disconnected concurrent update issue

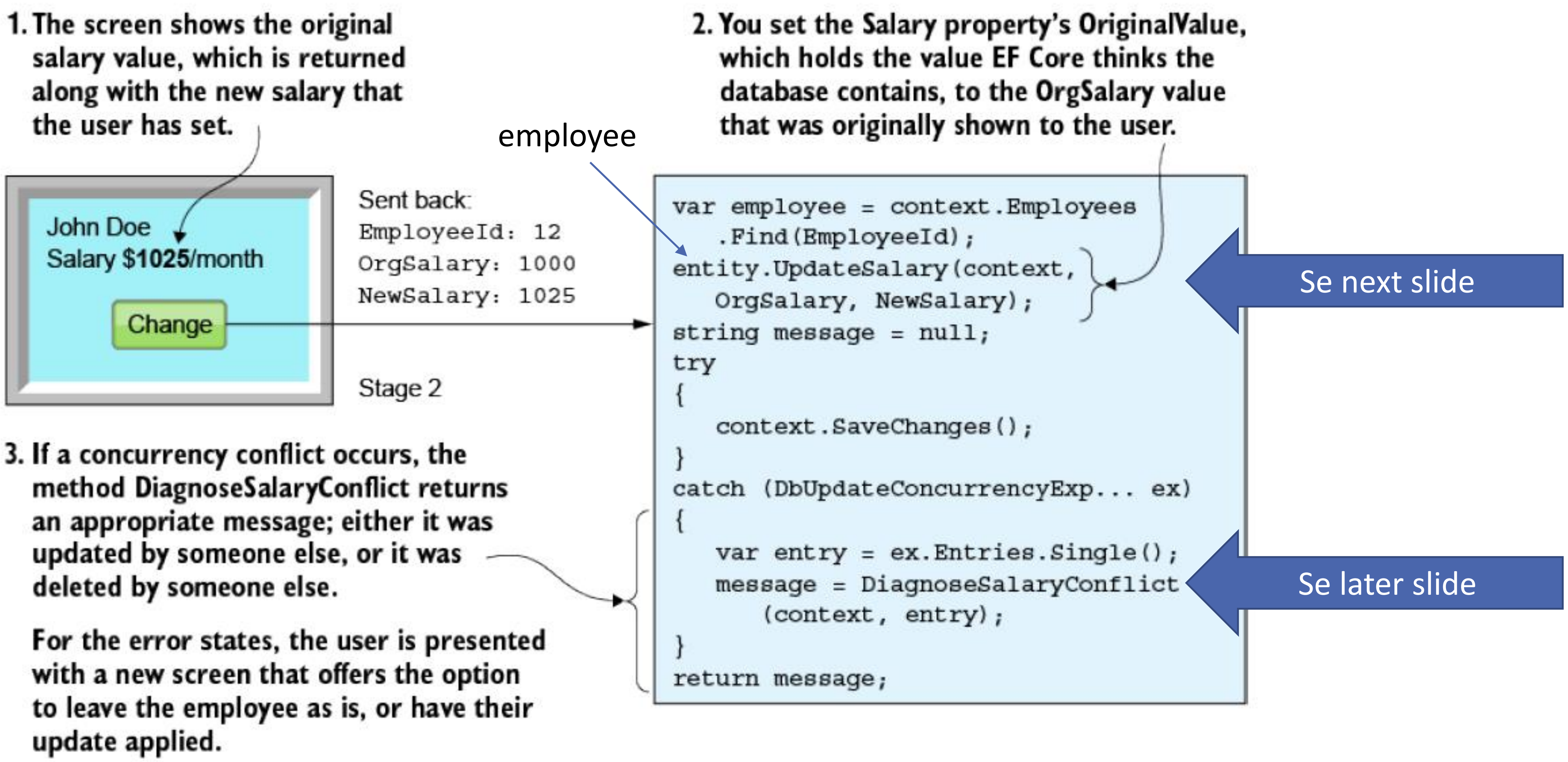

John Doe
Salary $1000/month
Change
John's boss
John Doe
Salary $1100/month
Click
Change
Stage 1
Stage 2
Time
Stage 1
Stage 2
John Doe
Salary $1000/month
Change
Human resources
John Doe
Salary $1025/month
Click
Change
3. Because of the order of updates, the boss's decision is silently ignored unless you add some disconnected concurrency checks.
2. Human Resources gets the same email and decides to give John Doe the standard 2.5% raise.

Although this looks very much like the concurrency conflicts example, the change is in the way a disconnected concurrency conflict is found.

The disconnected concurrent update issue

- Now the time between each person seeing the data and deciding what to do is measured in minutes instead of milliseconds, but if you don’t do anything about it, you can have another *concurrency conflict*, with potentially the wrong data set.
- The way a disconnected concurrency conflict is dealt with is often different.
- Typically, in a human user case, the decision on what should happen is given back to the user.
- If a conflict occurs, the user is presented with a new screen indicating what happened and is given a choice on what should be done.
- The user is then invited to accept the current state, or apply the update, knowing that this overrides the last user’s update.

Handling disconnected concurrent update



1. The screen shows the original salary value, which is returned along with the new salary that the user has set.

2. You set the Salary property's OriginalValue, which holds the value EF Core thinks the database contains, to the OrgSalary value that was originally shown to the user.

```
EmployeeId: 12
OrgSalary: 1000
NewSalary: 1025
```

```
var employee = context.Employees
    .Find(EmployeeId);
entity.UpdateSalary(context,
    OrgSalary, NewSalary);
string message = null;
try
{
    context.SaveChanges();
}
catch (DbUpdateConcurrencyExp... ex)
{
    var entry = ex.Entries.Single();
    message = DiagnoseSalaryConflict
        (context, entry);
}
return message;
```

3. If a concurrency conflict occurs, the method DiagnoseSalaryConflict returns an appropriate message; either it was updated by someone else, or it was deleted by someone else.

For the error states, the user is presented with a new screen that offers the option to leave the employee as is, or have their update applied.

Handling disconnected concurrent update

- The original salary, which was displayed to the user on the first screen, is sent back with the other data and used in the concurrency check when the Salary is updated.
- The entity class (Employee) used for this example, with the Salary property set as a concurrency token.

```
public class Employee
{
  public int EmployeeId { get; set; }
  public string Name { get; set; }
  [ConcurrencyCheck]
  public int Salary { get; set; }
  public void UpdateSalary (DbContext context, int orgSalary, int newSalary)
  {
    Salary = newSalary;
    context.Entry(this).Property(p => p.Salary).OriginalValue = orgSalary;
  }
}
```

Sets the OriginalValue, which holds the data read from the database, to the original value that was shown to the user in the first part of the update

Ask the user what to do

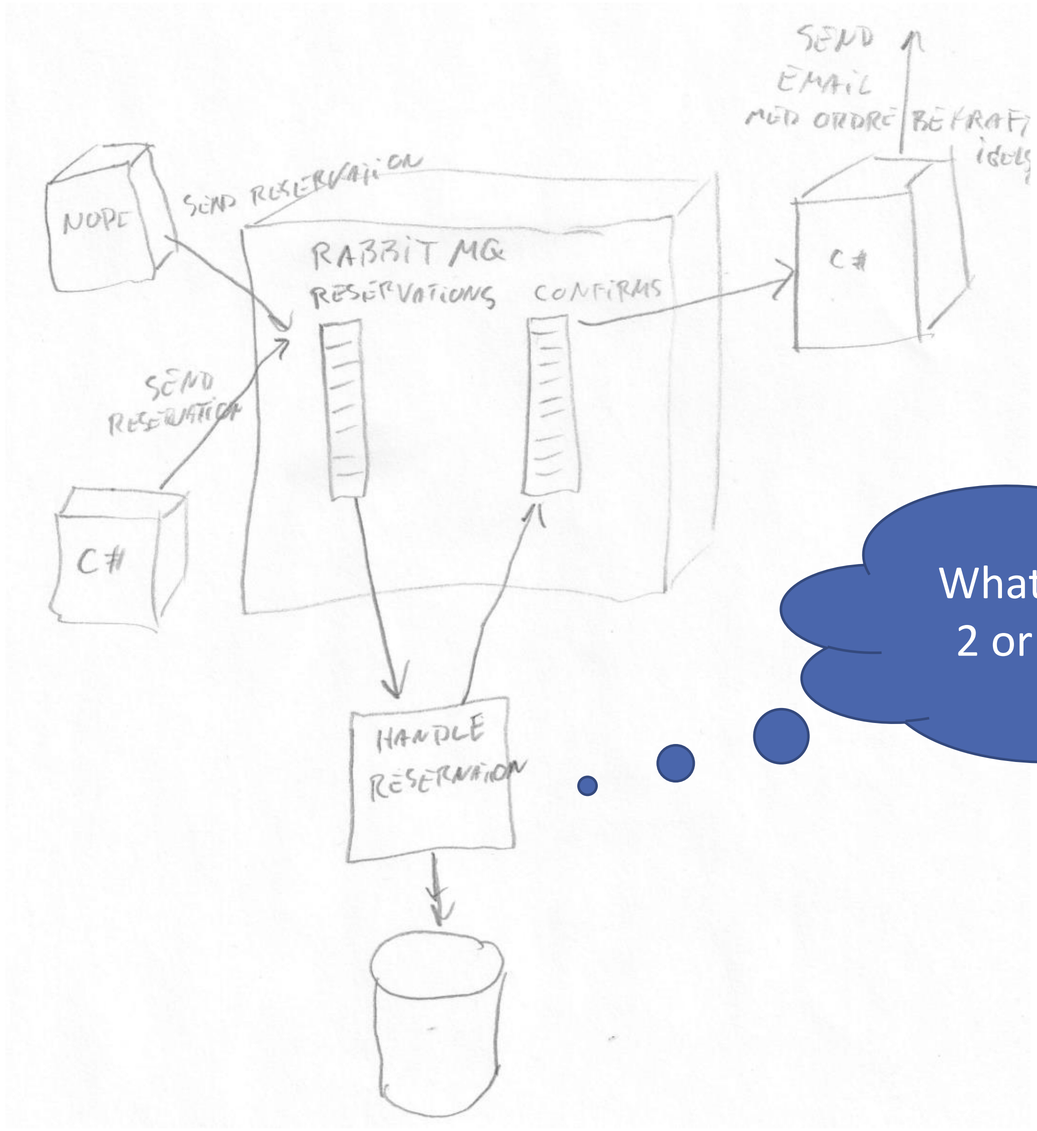
```
private string DiagnoseSalaryConflict(ConcurrencyDbContext context, EntityEntry entry)
{
  var employee = entry.Entity as Employee;
  if (employee == null)
    throw new NotSupportedException("Don't know how to handle concurrency conflicts for " +
                                     entry.Metadata.Name);
  var databaseEntity = context.Employees.AsNoTracking()
      .SingleOrDefault(p => p.EmployeeId == employee.EmployeeId);
  if (databaseEntity == null)
    return $"The Employee {employee.Name} was deleted by another user. " +
           $"Click Add button to add back with salary of {employee.Salary}" +
            " or Cancel to leave deleted.";
  return $"The Employee {employee.Name}'s salary was set to " +
         $"{databaseEntity.Salary} by another user. " +
         $"Click Update to use your new salary of {employee.Salary}" +
         $" or Cancel to leave the salary at {databaseEntity.Salary}.";
}
```

AARHUS UNIVERSITY
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Using timestamp?

- These disconnected concurrency-conflict examples use a concurrency token, but they work equally well with a timestamp.
- To use a timestamp instead of passing the Salary concurrency token used in these examples, you'd pass the timestamp and set the timestamp's original value before any update.

Example



What may happen if we run 2 or more of this process?

References & Links

- Entity Framework Core in Action, Second Edition, Jon P. Smith, Manning
- https://docs.microsoft.com/en-us/aspnet/core/data/ef-mvc/concurrency?view=aspnetcore-6.0#:~:text=Test%20Concurrency%20Conflicts