**Updating many to many relationships in Entity Framework Core**

Last Updated: December 19, 2018 | Created: March 1, 2017

I wrote an article called [Updating many to many relationships in entity framework](https://www.thereformedprogrammer.net/updating-a-many-to-many-relationship-in-entity-framework/" \t "_blank) back on 2014 which is still proving to be popular in 2017. To celebrate the release of my book [Entity Framework Core in Action](http://bit.ly/2m8KRA0" \t "_blank) I am producing an updated version of that article, but for [Entity Framework Core](https://docs.microsoft.com/en-us/ef/core/index) (EF Core).

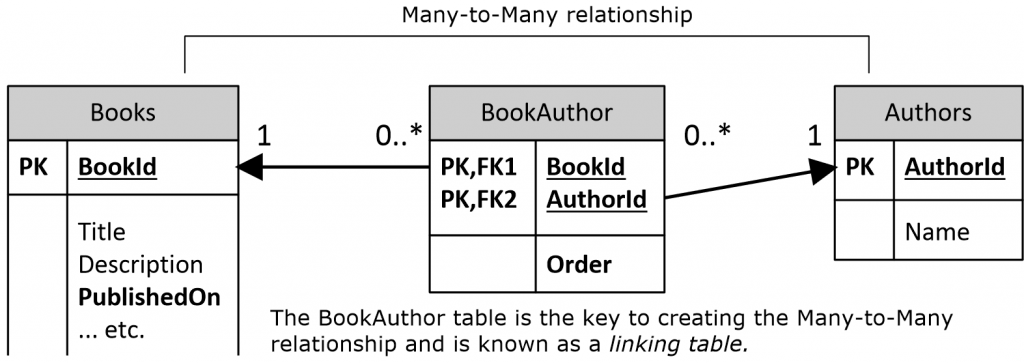
All the information and the code comes from Chapter 2 of my [book](http://bit.ly/2m8KRA0). In the book I use a book selling site, with books linked to authors via a many-to-many relationship. You can see the [live example site](http://efcoreinaction.com/) and if you scroll down you will see books with multiple authors.

All the unit tests and classes I use in this article can be found in the [Chapter03](https://github.com/JonPSmith/EfCoreInAction/tree/Chapter03) branch of the [Git repo](https://github.com/JonPSmith/EfCoreInAction) associated with this book.

*NOTE: If you are in a hurry I have added summaries for each section – just search for the word****Summary****to get the answers without needing to look at the code examples.*

**Creating a many-to-many relationship in EF Core**

The example I am going to use is one taken from the [book I am writing](http://bit.ly/2m8KRA0). It has a **Book** entity linked its **Author**(s) entities via a **BookAuthor** table. The database diagram below shows the three tables, with the foreign key links from the BookAuthor linking table to the other two tables.

[](https://www.thereformedprogrammer.net/wp-content/uploads/2017/03/many-to-many-database-diagram.png)

A linking table like this has foreign keys (FK in the diagram) which link to the primary key(s) (PK in the disgarm) of each end of the relationship – in this case the **BookId**of the **Book** entity and the **AuthorId**of the **Author** entity. These foreign keys then form a composite primary key for the **BookAuthor** table.

EF6.x created this table for you when you defined a many-to-many relationship, but EF Core, which takes a leaner approach, doesn’t – you need to do it. Its not hard, so let me give you the code to define the **BookAuthor** entity class.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | public class BookAuthor  {      public int BookId { get; set; }      public int AuthorId { get; set; }      public byte Order { get; set; }        //-----------------------------      //Relationships        public Book Book { get; set; }      public Author Author { get; set; }  } |

*NOTE: I have a property called****Order****because the order of the authors matter. If the order of items in your many-to-many list don’t matter then you can leave that out.*

EF Core will find the relationships using its *by convention* rules because I have used names that it understands:

* I used the same names, **BookId** and **AuthorId**, of the primary keys in the Book class and the Author class.
* Because I used the classes **Book** and **Author**, which EF Core knows are part of the database, then it knows its a relationship.

Now, the one thing that EF Core can’t work out is what the primary key of the **BookAuthor** table because they don’t follow the normal convension. There are a number of ways to define the primary key(s) of an entity, but I have used EF Core’s Fluent API that I access via the **OnModelCreating** method inside my DbContext, as shown below:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | public class EfCoreContext : DbContext  {      public DbSet<Book> Books { get; set; }      public DbSet<Author> Authors { get; set; }      public DbSet<PriceOffer> PriceOffers { get; set; }        public EfCoreContext(          DbContextOptions<EfCoreContext> options)          : base(options) {}        protected override void          OnModelCreating(ModelBuilder modelBuilder)      {          modelBuilder.Entity<BookAuthor>()              .HasKey(x => new {x.BookId, x.AuthorId});      }  } |

You can see the**.HasKey** method towards the bottom of the code where I define the composite key consisting of the **BookId** and the **AuthorId**.

The easiest approach to setting up relationships is by using EF Core’s conventions, which is what I have done. But if you want to explicitly define the relationships you can, using the**.HasOne/.HasMany** Fluent API commands, as shown below.

*Note: There are loads of configuration rules, which I cover in chapter 6 and 7 of [my book](http://bit.ly/2m8KRA0), or you can look at the*[*EF Core docs*](https://docs.microsoft.com/en-us/ef/core/modeling/relationships)*.*

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | //NOTE: I only show the OnModelCreating part of the DbContext  protected override void      OnModelCreating(ModelBuilder modelBuilder)  {      modelBuilder.Entity<BookAuthor>()         .HasKey(x => new {x.BookId, x.AuthorId});        //If you name your foreign keys correctly, then you don't need this.      modelBuilder.Entity<BookAuthor>()          .HasOne(pt => pt.Book)          .WithMany(p => p.AuthorsLink)          .HasForeignKey(pt => pt.BookId);        modelBuilder.Entity<BookAuthor>()          .HasOne(pt => pt.Author)          .WithMany(t => t.BooksLink)          .HasForeignKey(pt => pt.AuthorId);  } |

*NOTE: You can set the keys, but not the relationships, via attributes. The [Key] and [Order] attributes allow you to define the primary key, but I find the Fluent API is quite clean and easy to understand.*

**Summary – how to add a many-to-many relationship**

To add a many-to-many relationship in EF Core you need to:

1. Create a linking entity class that has the foreign keys(s) of the two entities you want to form a many-to-many link between. In my example these are **BookId** and **AuthorId**.
2. It is efficient (but not necessary) to make these two foreign keys into the composite primary key. This will ensure that there is only ever one link between the two entities you want. It is also more efficient in terms of size of the database table.
   1. If you do want the two foreign keys to be the composite primary key you need to configure this in EF Core, either via Fluent API or by [Key]/[Order] attributes, because it cannot find them using its normal “by convention” method.
   2. If you use properties for the foreign key that don’t  have the same name as the primary key it links to, then you must manually configure the relationship. There are a number of ways to do that, but EF Core’s Fluent API is quick and fairly straightforward to understand.
3. The linking class can, but don’t have to, have relationship links to the two entities it links. In my example my **AuthorLinks** class has a property **Book** of type **Book**, and a property **Author** of type **Author**.
4. You most likely want a collection relationship to the linking table in one or both of the linked entities. In my example I have property called **AuthorLinks** with the type **ICollection<BookAuthor>** in my **Book** class, and a property called **BooksLink** of type **ICollection<BookAuthor>** in my **Author** class.

**Adding a Book many-to-many Author link**

Creating new many-to-many link is fairly easy in EF Core (but not as easy as EF6.x). The code below creates a new book with a new author.

*Note: the variable****context****used at the bottom of the code below is an instance of the application’s DbContext, called****EfCoreContext****in this case (see its definition above).*

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | var book = new Book  {    Title = "Quantum Networking",    Description = "faster-than-light data communications",    PublishedOn = new DateTime(2057, 1, 1),    Price = 220  };  var author = new Author { Name = "Future Person" };  book.AuthorsLink = new List<BookAuthor>  {    new BookAuthor {      Author = author,      Book = book,      Order = 0    }  };    //Now add this book, with all its relationships, to the database  context.Books.Add( book);  context.SaveChanges(); |

**Updating a many-to-many relationship**

It turns out there are two scenarios under which you want to update a relationship. They are:

1. **Connected state:** This is where the load of the data and the update is done in one go, that is within one instance of the application’s DbContext. You find this sort of update that happens in a console application, or inside your business logic (see [this link](https://www.thereformedprogrammer.net/architecture-of-business-layer-working-with-entity-framework-core-and-v6-revisited/) on business logic and EF Core)
2. **Disconnected state:** This is where the update is split into two halves: a) select the entities you want to change, b) and make the change. Each stage has a different instance of the application’s DbContext. This happens on web sites, where in the first stage the user picks what they want done and posts it back. The second stage then has to re-read the data and update it.

I am going to describe these two approaches separately, so you can go to the one that fits the application you are building.

**1. Connected state update**

In the connected state we read in the **Book** entity and update it immediately, i.e. within the same instance of the DbContext. For my Book/Author example I am going to read in the **Book**, with its linking entities, **BookAuthor**, as *tracked entities*. – that means that EF Core will take a copy of the entities we read in so that it can spot any changes when you call **SaveChanges**.

The listing below is one of my unit tests which adds the existing author “Martin Fowler” to the book called “Quantum Networking”, which currently has one author called “Future Person”. After the test has finished the book “Quantum Networking” has two authors, “Future Person” and “Martin Fowler”

*NOTE: I am using a Sqlite, in-memory database, which I seed with four books with known titles and authors.*

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | public void TestAddExtraAuthorOk()  {      //SETUP      var inMemDb = new SqliteInMemory();        using (var context = inMemDb.GetContextWithSetup())      {          context.SeedDatabaseFourBooks();            //ATTEMPT          var book = context.Books              .Include(p => p.AuthorsLink)              .Single(p => p.Title == "Quantum Networking");            var newAuthor = context.Authors              .Single(p => p.Name == "Martin Fowler");            book.AuthorsLink.Add(new BookAuthor          {              Book = book,              Author = newAuthor,              Order = (byte) book.AuthorsLink.Count          });          context.SaveChanges();            //VERIFY          var bookAgain = context.Books              .Include(p => p.AuthorsLink)              .Single(p => p.BookId == book.BookId);          bookAgain.AuthorsLink.ShouldNotBeNull();          bookAgain.AuthorsLink.Count.ShouldEqual(2);          var authorsInOrder = bookAgain.AuthorsLink.OrderBy(p => p.Order).ToList();          authorsInOrder.First().Author.Name.ShouldEqual("Future Person");          authorsInOrder.Last().Author.Name.ShouldEqual("Martin Fowler");      }  } |

The most important parts of the code are lines 11 to 13. Here I load the **Book** entity using an Include method to load the **AuthorLinks** at the same time, as *tracked entities*. The **AuthorLinks** property holds a a collection of **BookAuthor** entities linking to the **Author** entities.

*Note: The default way of loading data in EF Core is as tracked entities, that is, EF Core takes a copy of the loaded data so that it can detect if any changes have happened since they were loaded. You can turn off tracking by including the method .AsNoTracking to your query (useful in read-only queries, as it makes them a little faster).*

In lines 15 to 16 I also load, again as a *tracked entity*, the **Author** entity who’s name is “Martin Fowler” as that the author I want to add to this book.

Then on lines 18 to 23 I create a new **BookAuthor** linking entry linking the Book entity I loaded to the Author entity of “Martin Fowler”.

When **SaveChanges** is called on line 24 it finds that the **AuthorLinks** property  of the book instance it loaded has changed. Because this is a relationship it looks at the entities and finds that there is a new **BookAuthor**entity, which it adds to the database, and a author which is already in the database so it doesn’t have to add it.

**Summary – Connected state many-to-many update**

To update a many-to-many relationship in the connected state:

1. Load an entity at one end of the relationship (doesn’t matter which end) with the collection of links in the many-to-many linking table. In my example I loaded the **Book** entity I wanted to change along with its **AuthorLinks** property, which holds the collection of the linking entities in the **BookAuthor** table.
2. Alter the linking table collection, either by adding or removing entries from the collection. In my example I added a new **BookAuthor** class to the **AuthorLinks** property collection.
   1. Note: if you want to replace the whole collection then you can simply assign a new list to the linking collection (there is an example of this in my [unit tests](https://github.com/JonPSmith/EfCoreInAction/blob/Chapter03/Test/UnitTests/DataLayer/Ch03_ManyToManyUpdate.cs) – see test named TestChangeAuthorsNewListOk).

**2. Disconnected state**

The disconnected state happens when the initial read of the data and update of the data are done separately, i.e. they use different instances of the applications DbContext. This happens in a web application which has two stages:

1. The first stage is where the user is presented with the book, its current author(s) and a list of possible authors to choose from. Once they have chosen the author to add then press a button which takes their choice back to the web application
2. In the second stage the web application needs to re-read the data and execute the database update.

The unit test below does the same as the previous example, that is, it adds the existing author “Martin Fowler” to the book called “Quantum Networking”, which starts out with one author called “Future Person”. After the test has finished the book “Quantum Networking” has two authors, “Future Person” and “Martin Fowler”.

I simulate the disconnected state by having two instances of the application’s DbContext and passing the primary keys of the Book and the new Author to add via a small class called **ChangeAuthorDto**.

*Note: This unit test uses two, separate instances of the DbContext to simulate the disconnected state. With the help of****Daria****(see comment below) I have found a way to use Sqlite in-memory with multiple instances of the DbContext. Daria pointed out that the in-memory part is held in the Sqlite connection, so I create one options (which uses the Sqlite connection) and used that to create two separate DbContext instances  – thanks Daria.*

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50 | [Fact]  public void TestAddAuthorDisconnectedOk()  {      //SETUP      var options = SqliteInMemory.CreateOptions<EfCoreContext>();      ChangeAuthorDto dto;      using (var context = new EfCoreContext(options))      {          context.Database.EnsureCreated();          context.SeedDatabaseFourBooks();          var book = context.Books              .Include(p => p.AuthorsLink)              .Single(p => p.Title == "Quantum Networking");            var newAuthor = context.Authors              .Single(p => p.Name == "Martin Fowler");          dto = new ChangeAuthorDto          {              BookId = book.BookId,              NewAuthorId = newAuthor.AuthorId          };      }        using (var context = new EfCoreContext(options))      {          //ATTEMPT          var book = context.Books              .Include(p => p.AuthorsLink)              .Single(p => p.BookId == dto.BookId);          var newAuthor = context.Authors.Find(dto.NewAuthorId);            book.AuthorsLink.Add(new BookAuthor          {              Book = book,              Author = newAuthor,              Order = (byte)book.AuthorsLink.Count          });          context.SaveChanges();            //VERIFY          var bookAgain = context.Books              .Include(p => p.AuthorsLink).ThenInclude(p => p.Author)              .Single(p => p.BookId == dto.BookId);          bookAgain.AuthorsLink.ShouldNotBeNull();          bookAgain.AuthorsLink.Count.ShouldEqual(2);          var authorsInOrder = bookAgain.AuthorsLink.OrderBy(p => p.Order).ToList();          authorsInOrder.First().Author.Name.ShouldEqual("Future Person");          authorsInOrder.Last().Author.Name.ShouldEqual("Martin Fowler");      }  } |

The important lines are 27 to 30, where I load in the **Book**, with its **BookAuthor** linking collection accessed via the Book’s **AuthorLinks** property, and I read in the **Author** that the user wants to add.

Having loaded that book I can use the same code I had in the connected state, that is, I  create a new **BookAuthor** linking entry linking the Book entity I loaded to the Author entity of “Martin Fowler” and add it to the existing **AuthorLinks** collection – see lines 32 to 37.

**Summary – Disconnected state many-to-many update**

To update a many-to-many relationship in the disconnected state:

1. In the first stage you should return the primary key(s) of the two ends of the many-to-many relationships that you want to link/unlink. In my example I provide the **BookId** and the **AuthorId** I want to add to the book.
2. In stage two you load one end of the relationship (doesn’t matter which end) with the collection of links the the linking table. In my example I loaded the **Book** entity using the **BookId** key along with its **AuthorLinks** property, which held the collection of the linking entities in the **BookAuthor** table.
3. Now load the other end of the relationship. In my example the **Author** entity, using the **AuthorId** returned from the first stage.
4. Alter the linking table collection, either by adding or removing entries from the collection. In my example I added a new **BookAuthor** class to the **AuthorLinks** property collection.
   1. Note: if you want to replace the whole collection then you can simply assign a new list to the linking collection with the new BookAuthor class.

**2a. Another, quicker way to update a many-to-many relationships in the disconnected state**

In chapter 3 of my book, [Entity Framework Core in Action](http://bit.ly/2m8KRA0" \t "_blank) , I point out that you can often change relationships using the foreign keys, and it can me more efficent. This is especially true of many-to-many relationships because you can directly add/delete entries from the linking table, in this case **BookAuthor**.

By directly adding/deleting the linking table entries then it saves you having to load the **Book** entity with its **AuthorLinks** property and the **Author** you want to add. Coupled with the fact that the first stage is going to supply you with the primary keys anyway then it saves quite a few database queries.

The unit test below shows this in action. The first stage of the unit test hands back primary keys of the **Book** and the new **Author** to add via the **ChangeAuthorDto** class. The second stage, which uses a new instance of the DbContext, uses those primary keys to build a new **BookAuthor** linking entity to add to the database.

*NOTE: The one downside of this approach for my Book/Author example is the setting the****Order****property, as it should be set to get the correct order of the authors. For this example know there is one author with an****Order****set to 0 so I simply set it to 1. In real application the first stage would have to define what order the authors should be shown in.*

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44 | [Fact]  public void TestAddAuthorViaForeignKeyOk()  {      //SETUP      var options =          this.NewMethodUniqueDatabaseSeeded4Books();        ChangeAuthorDto dto;      using (var context = new EfCoreContext(options))      {          var book = context.Books              .Single(p => p.Title == "Quantum Networking");            var newAuthor = context.Authors              .Single(p => p.Name == "Martin Fowler");          dto = new ChangeAuthorDto          {              BookId = book.BookId,              NewAuthorId = newAuthor.AuthorId          };      }        using (var context = new EfCoreContext(options))      {          //ATTEMPT          context.Set<BookAuthor>().Add(new BookAuthor          {              BookId = dto.BookId,              AuthorId = dto.NewAuthorId,              Order = 1          });          context.SaveChanges();            //VERIFY          var bookAgain = context.Books              .Include(p => p.AuthorsLink).ThenInclude(p => p.Author)              .Single(p => p.BookId == dto.BookId);          bookAgain.AuthorsLink.ShouldNotBeNull();          bookAgain.AuthorsLink.Count.ShouldEqual(2);          var authorsInOrder = bookAgain.AuthorsLink.OrderBy(p => p.Order).ToList();          authorsInOrder.First().Author.Name.ShouldEqual("Future Person");          authorsInOrder.Last().Author.Name.ShouldEqual("Martin Fowler");      }  } |

The important lines are 26 to 31, where I create a new **BookAuthor** class using the **BookId** and the **AuthorId** of the new author I want to add to the book. If you compare this to the last disconnected example you will see this avoids three database loads:

1. The **Book** we wanted to update
2. The Books **AuthorLinks** collection of **BookAuthor** entities
3. And the **Author** that I wanted to add.

In this case I added an extra author. I could have removed an author by deleting the appropriate BookAuthor entry.

I do recommend this approach for many-to-many relationship updates because it is more efficient and still quite clear in its intent, i.e. its not complex so its obvious what is happening.

**Summary – quicker way to update a many-to-many relationships in the disconnected state**

For updates to many-to-many relationships in the disconnected state you can:

1. In the first stage you should return the primary key(s) of the two ends of the many-to-many relationships that you want to link/unlink. In my example I provide the **BookId** and the **NewAuthorId** I wanted to add to the book.
2. In the second stage you should add/delete entries from the linking table directly. For instance to add a new many-to-many relationship you add a new entry in the linking table. In my example I added a new **BookAuthor** link  with its **BookId** property set from the **BookId** returned by the first stage, and its **AuthorId** property set to the **NewAuthorId** value from the first stage.

**Conclusion**

Setting up and changing many-to-many links in EF Core is harder than in EF6.x. EF 6.x used the same approach as EF Core, but “hid” the linking table and the manipulation of that table from you. That certainly made it easier, but as the continued access to my [original article on updating many-to-many in EF6.x](https://www.thereformedprogrammer.net/updating-a-many-to-many-relationship-in-entity-framework/) shows, it can cause confusion sometimes.

At the moment EF Core makes you configure the linking table and update that linking table yourself. The EF Core roadmap says it will look at automating this in the future, but not for now. But I think there are some benefits in knowing what is going on in the database, and, as my last disconnected example shows, you can sometimes be more efficient that EF6.x is at updating many-to-many relationships.

I cover this an much more in my book [Entity Framework Core in Action](http://bit.ly/2m8KRA0" \t "_blank) and you can access the unit test code you see here via the Chapter03 branch of the Git repo that goes with the book – see the test class [**Ch03\_ManyToManyUpdate**](https://github.com/JonPSmith/EfCoreInAction/blob/Chapter03/Test/UnitTests/DataLayer/Ch03_ManyToManyUpdate.cs).

Happy coding!