<https://www.learnentityframeworkcore.com/>

# Welcome To Learn Entity Framework Core

This site provides documentation and tutorials for people looking for help with using Entity Framework Core, Microsoft's recommended data access technology for applications based on the [**.NET Core framework**](https://www.microsoft.com/net/core).

## **What is Entity Framework Core?**

Entity Framework Core (EF Core) is the latest version of the Entity Framework from Microsoft. It has been designed to be lightweight, extensible and to support cross platform development as part of Microsoft's .NET Core framework. It has also been designed to be simpler to use, and to offer performance improvements over previous versions of Entity Framework.

EF Core is an object-relational mapper (ORM). Object-relational mapping is a technique that enables developers to work with data in object-oriented way by performing the work required to **map** between **objects** defined in an application's programming language and data stored in **relational** datasources.

## **Why use an ORM?**

Most development frameworks include libraries that enable access to data from relational databases via recordset-like data structures. The following code sample illustrates a typical scenario where data is retrieved from a database and stored in an ADO.NET DataTable so that it is accessible to the program's code:

1. using(var conn = new SqlConnection(connectionString))
2. using(var cmd = new SqlCommand("select \* from Products", conn))
3. {
4. var dt = new DataTable();
5. using(var da = new SqlDataAdapter(cmd))
6. {
7. da.Fill(dt);
8. }
9. }

The data within the DataTable is accessible via numeric or string indexers and needs to be converted from object to the correct type:

1. foreach(DataRow row in dt.Rows)
2. {
3. int productId = Convert.ToInt32(row[0]);
4. string productName = row["ProductName"].ToString();
5. }

This late-bound or "weakly-typed" approach to data access is prone to error. Problems commonly arise from mistyping the name of a column, or finding that the name of the column has been changed in the database, or from a change to the order in which fields are specified in the SQL statement without a corresponding change being made to the application code. Equally, data type conversions might fail. The code will still compile, but will error at runtime. Consequently, professional developers prefer to work with data in a strongly-typed manner.

## **Strong Typing**

When you take a strongly-typed approach to data, you work with properties of predefined classes that form a [**domain model**](https://www.learnentityframeworkcore.com/model) in an object-oriented way:

1. public class Product
2. {
3. int ProductId { get; set; }
4. string ProductName { get; set; }
5. }
6. int productId = myProduct.ProductId;
7. string productName = myProduct.ProductName;

Work still needs to be done to retrieve and map the data from the database to an instance of the domain object. One option is to write your own code to manage this. However, as the domain model grows, the amount of code required can grow, and will need more and more development time to maintain. This will increase the overall amount of time required to complete an application.

ORMs are pre-written libraries of code that do this work for you. Full-featured ORMs do a lot more too. They can

* map a domain model to database objects
* create databases and maintain the schema in line with changes to the model
* generate SQL and execute it against the database
* manage transactions
* keep track of objects that have already been retrieved

### **Next**

* [**How To Get Entity Framework Core**](https://www.learnentityframeworkcore.com/efcore/how-to-get)

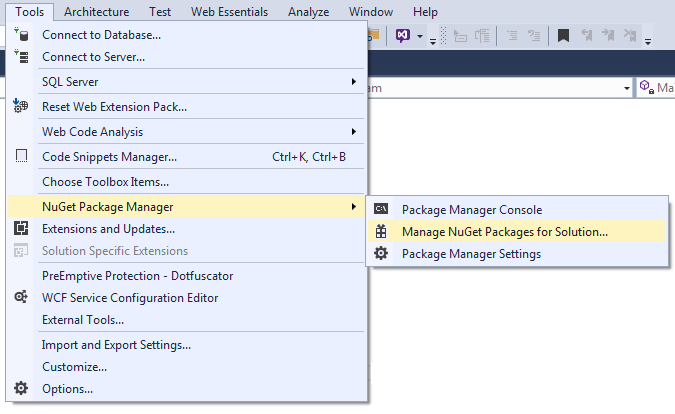
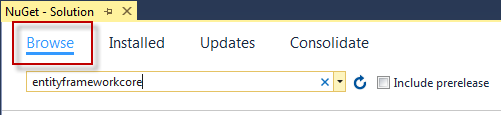
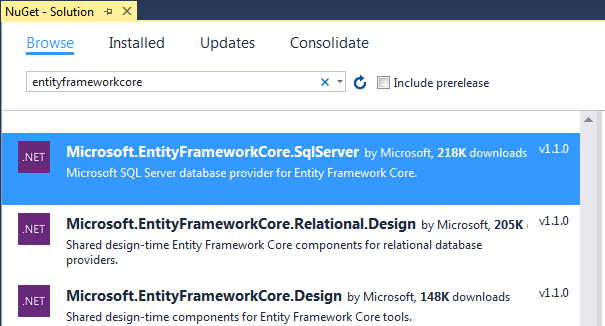
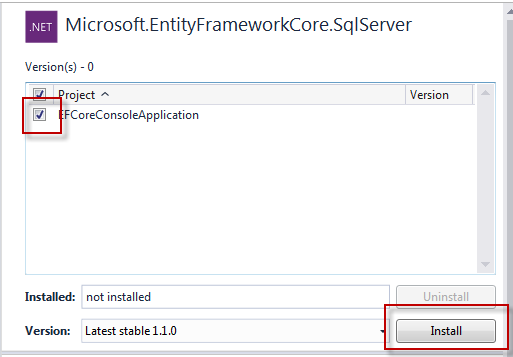
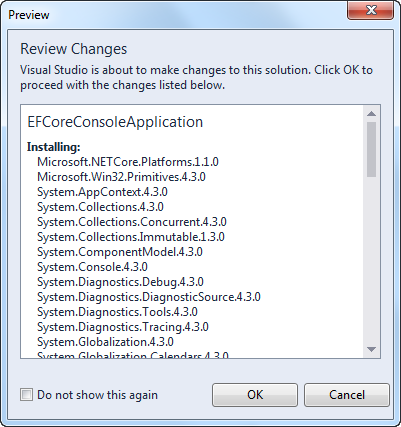
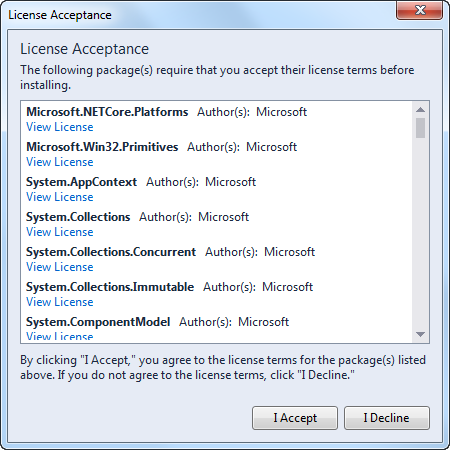
## **How to get Entity Framework Core**

Entity Framework Core is available as a Nuget Package that can be added to your project in a number of ways depending on the project type and the tools available to you.

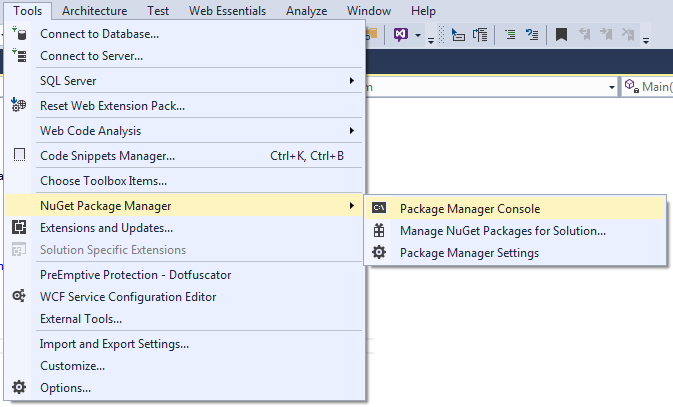
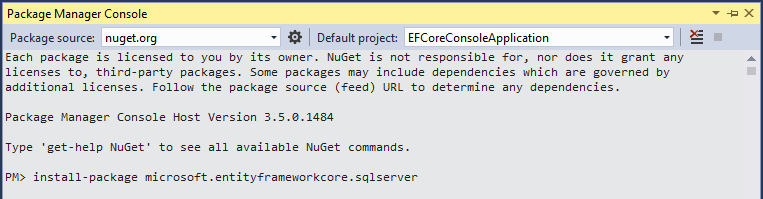
### **Visual Studio Package Manager**

Users of Visual Studio can install Entity Framework Core via one of the package management tool options regardless of the project type (.NET Core or the full .NET Framework): the NuGet Package Manager UI; or the Package Manager Console.

#### **Nuget Package Manager UI**

1. Go to **Tools** » **NuGet Package Manager** » **Manage NuGet Packages For Solution**  
   
2. Ensure that **Browse** is selected and type "entityframeworkcore" into the search box  
   
3. Click on the provider that you want to install. SQL Server is selected in this case.  
   
4. Check the project that you want to install the package into, then click Install  
   
5. Review the changes that are about to be made to your project (unless you have previously ticked the box to prevent this dialog from appearing)  
   
6. Finally, accept the terms of the various licenses associated with the packages that are about to be installed  
   

#### **Package Manager Console**

1. Go to **Tools** » **NuGet Package Manager** » **Package Manager Console**  
   
2. Ensure that the correct project is selected in the "Default Project" dropdown, and type install-package microsoft.entityframeworkcore.sqlserver to install the SQL Server provider.  
   
3. Hit return, and the installation process should start.

### **Command Line Tools**

You can use the .NET Core commmand line tools to install Entity Framework Core. Once you have created your project, navigate to the folder containing the .csproj (or project.json) file and execute the following command:

1. dotnet add package Microsoft.EntityFrameworkCore.SqlServer

You also need the Entity Framework Core tools if you want to make use of EF commands for migrations, scaffolding etc, so type the following command:

1. dotnet add package Microsoft.EntityFrameworkCore.Tools --version 1.1.0-msbuild3-final

### **Project.json**

The project.json file has been deprecated in favour of a .csproj file, but for older .NET Core projects on any platform, you can modify the project.json file to install Entity Framework Core. The following example shows a basic project.json file amended to include the 1.1.0 version of the SQL Server provider:

1. {
2. "version": "1.0.0-\*",
3. "buildOptions": {
4. "emitEntryPoint": true
5. },
6. "dependencies": {
7. "Microsoft.Extensions.Configuration": "1.0.0",
8. "Microsoft.Extensions.Configuration.FileExtensions": "1.0.0",
9. "Microsoft.Extensions.Configuration.Json": "1.0.0",
10. "Microsoft.EntityFrameworkCore.SqlServer": "1.1.0",
11. "Microsoft.EntityFrameworkCore.Tools": "1.1.0-msbuild3-final",
12. "Microsoft.NETCore.App": {
13. "type": "platform",
14. "version": "1.0.0"
15. }
16. },
17. "frameworks": {
18. "netcoreapp1.0": {
19. "imports": "dnxcore50"
20. }
21. },
22. "tools": {
23. "Microsoft.EntityFrameworkCore.Tools": "1.1.0-preview4"
24. }
25. }

A reference to the tooling is also included in both the dependencies section and the tools section. These are required if you want to run [**commands**](https://www.learnentityframeworkcore.com/migrations/commands) for scaffolding or migrations.

If you choose this approach to installing Entity Framework Core while using Visual Studio, packages are automatically restored when you save the changes to the project.json file. Otherwise you may need to manually restore packages, depending on the IDE that you are using and its capabilities. To manually restore the package, use a command prompt to navigate to the folder that houses the project.json file and type dotnet restore. This assumes that you have the [**.NET Core SDK**](https://www.microsoft.com/net/core) installed.

# The Entity Framework Core DbContext

The Entity Framework Core DbContext class represents a session with a database and provides an API for communicating with the database with the following capabilities:

* Database Connections
* Data operations such as querying and persistance
* Change Tracking
* Model building
* Data Mapping
* Object caching
* Transaction management

## **Database Connections**

The DbContext is responsible for opening and managing connections to the database.

## **Data Operations**

The DbContext provides methods for performing the following data operations directly.

* [**Adding data**](https://www.learnentityframeworkcore.com/dbcontext/adding-data)
* [**Modifying data**](https://www.learnentityframeworkcore.com/dbcontext/modifying-data)
* [**Deleting data**](https://www.learnentityframeworkcore.com/dbcontext/deleting-data)

The DbContext also provide [**data querying**](https://www.learnentityframeworkcore.com/dbset/querying-data) capability via the DbSet property.

## **Change Tracking**

The [**Change Tracker**](https://www.learnentityframeworkcore.com/dbcontext/change-tracker) detects changes made to entities and sets the EntityState of an object accordingly. The state of the entity determines the type of operation that the database will be asked to perform upon it, and therefore the SQL that will be generated.

## **Model Building**

The DbContext builds a [**conceptual model**](https://www.learnentityframeworkcore.com/model) based on [**convention**](https://www.learnentityframeworkcore.com/conventions) and [**configuration**](https://www.learnentityframeworkcore.com/configuration), and maps that to the database. The model and its mappings are built at application startup and are persisted in memory for the lifetime of the application.

## **Data Mapping**

The DbContext includes a [**data mapper layer**](http://martinfowler.com/eaaCatalog/dataMapper.html) responsible for mapping the results of SQL queries to entity instances and other types defined by the client application.

## **Object Caching**

The DbContext provides a first-level cache for objects that it is asked to retrieve from the data store. Subsequent requests for the same object will return the cached object instead of executing another database request.

## **Transaction Management**

When the DbContext SaveChanges method is called, a transaction is created and all pending changes are wrapped in it as a single [**unit of work**](http://martinfowler.com/eaaCatalog/unitOfWork.html). If an error occurs when the changes are applied to the database, they are rolled back and the database is left in an unmodified condition.

## **Adding data via the DbContext**

The key methods for adding entities via the DbContext are

* Add<TEntity>(TEntity entity)
* Add(object entity)
* AddRange(IEnumerable<object> entities)
* AddRange(params object[] entities)

These methods are new to the DbContext in Entity Framework Core and have no equivalents in previous version of Entity Framework where the DbContext is available (i.e. EF 4.1 onwards).

Most often, you will use the generic version of Add but omit the type parameter [**because the compiler will infer the type from the argument passed in to the method**](https://msdn.microsoft.com/en-us/library/twcad0zb.aspx). The following two examples are identical:

1. // with type parameter
2. var author = new Author{ FirstName = "William", LastName = "Shakespeare" };
3. context.Add<Author>(author);
4. context.SaveChanges();
5. // without type parameter
6. var author = new Author{ FirstName = "William", LastName = "Shakespeare" };
7. context.Add(author);
8. context.SaveChanges();

Visual Studio 2015 offers helpful advice to omit the type parameter in the first example. The second example should not be confused with the version of Add that takes an object type:

1. object author = new Author{ FirstName = "William", LastName = "Shakespeare" };
2. context.Add(author);
3. context.SaveChanges();

When you use either version of Add the context begins tracking the entity that was passed in to the method and applies an EntityState value of Added to it. The context also applies the same EntityState value of Added to all other objects in the graph that aren't already being tracked by the context. In the next example, the Added state is also applied to the books:

1. var context = new SampleContext();
2. var author = new Author {
3. FirstName = "William",
4. LastName = "Shakespeare",
5. Books = new List<Book>
6. {
7. new Book { Title = "Hamlet"},
8. new Book { Title = "Othello" },
9. new Book { Title = "MacBeth" }
10. }
11. };
12. context.Add(author);
13. context.SaveChanges();

The books are added by virtue of the fact that they are referenced through the Books property of the author. In the next example, the books will **not** be added:

1. var author = new Author { FirstName = "William", LastName = "Shakespeare" };
2. var hamlet = new Book { Title = "Hamlet", Author = author };
3. var othello = new Book { Title = "Othello", Author = author };
4. var macbeth = new Book { Title = "MacBeth", Author = author };
5. context.Add(author);
6. context.SaveChanges();

Although the author has been assigned to the Author property of each of the books that have been instantiated, the author entity is unaware of these relationships. It's Books property is still null and the books are not added to the context.

### **Adding Multiple Records**

The AddRange method is used for adding multiple objects to the database in one method call. The code in the next example is very similar to the previous example, but the AddRange method is used to save all the books and the author to the database in one go:

1. var context = new SampleContext();
2. var author = new Author { FirstName = "Stephen", LastName = "King" };
3. var books = new List<Book> {
4. new Book { Title = "It", Author = author },
5. new Book { Title = "Carrie", Author = author },
6. new Book { Title = "Misery", Author = author }
7. };
8. context.AddRange(books);
9. context.SaveChanges();

This version of the AddRange method takes an IEnumerable<object>. EF Core is clever enough to identify the type of objects being added to the context and will form the appropriate SQL. The author is related to all of the books, so it forms part of the graph and is added too.

The other version of the AddRange method takes a params array, and provides the facility to add a number of unrelated objects to the database in one go:

1. var context = new SampleContext();
2. var author = new Author { FirstName = "William", LastName = "Shakespeare" };
3. var book = new Book { Title = "Adventures of Huckleberry Finn" };
4. context.AddRange(author, book);
5. context.SaveChanges();

When the SaveChanges method is called on the DbContext, all entities with an EntityState of Added will be inserted into the database.

##### **Further Reading**

* [**Adding data via the DbSet**](https://www.learnentityframeworkcore.com/dbset/adding-data)

## **Modifying data via the DbContext**

The approach that you adopt to modifying entities depends on whether the context is currently tracking the entity being modified or not.

In the following example, the entity is obtained by the context, so the context begins tracking it immediately. When you alter property values on a tracked entity, the context changes the EntityState for the entity to Modified and the ChangeTracker records the old property values and the new property values. When SaveChanges is called, an UPDATE statement is generated and executed by the database.

1. var author = context.Authors.First(a => a.AuthorId == 1);
2. author.FirstName = "Bill";
3. context.SaveChanges();

Since the ChangeTracker tracks which properties have been modified, the context will issue a SQL statement that updates only those properties that were changed:

1. exec sp\_executesql N'SET NOCOUNT ON;
2. UPDATE [Authors] SET [FirstName] = @p0
3. WHERE [AuthorId] = @p1;
4. SELECT @@ROWCOUNT;
5. ',N'@p1 int,@p0 nvarchar(4000)',@p1=1,@p0=N'Bill'

### **Disconnected Scenario**

In a disconnected scenario such as an ASP.NET application, changes to an existing entity's property values can take place in a controller or service method, well away from the context. In these cases, the context needs to be informed that the entity is in a modified state. This can be achieved in several ways: setting the EntityState for the entity explicitly; using the DbContext.Update method (which is new in EF Core); using the DbContext.Attach method and then "walking the object graph" to set the state of individual properties within the graph explicitly.

### **Setting EntityState**

You can set the EntityState of an entity via the EntityEntry.State property, which is made available by the DbContext.Entry method.

1. public void Save(Author author)
2. {
3. context.Entry(author).State = EntityState.Modified;
4. context.SaveChanges();
5. }

This approach will result in just the author entity being assigned the Modified state. Any related objects will not be tracked. Since the ChangeTracker is unaware of which properties were modified, the context will issue an SQL statement updating all property values (apart from the primary key value).

### **DbContext Update**

The DbContext class provides Update and UpdateRange methods for working with individual or multiple entities.

1. public void Save(Author author)
2. {
3. context.Update(author);
4. context.SaveChanges();
5. }

As with setting the entity's State, this method results in the entity being tracked by the context as Modified. Once again, the context doesn't have any way of identifying which property values have been changed, and will generate SQL to update all properties. Where this method differs from explicitly setting the State property, is in the fact that the context will begin tracking any related entities (such as a collection of books in this example) in the Modified state, resulting in UPDATE statements being generated for each of them. If the related entity doesn't have a key value assigned, it will be marked as Added, and an INSERT statement will be generated.

### **Attach**

When you use the Attach method on an entity, it's state will be set to Unchanged, which will result in no database commands being generated at all. All other reachable entities with key values defined will also be set to Unchanged. Those without key values will be marked as Added. However, now that the entity is being tracked by the context, you can inform the context which properties were modified so that the correct SQL to update just those values is generated:

1. var context = new TestContext();
2. var author = new Author {
3. AuthorId = 1,
4. FirstName = "William",
5. LastName = "Shakespeare"
6. };
7. author.Books.Add(new Book {BookId = 1, Title = "Othello" });
8. context.Attach(author);
9. context.Entry(author).Property("FirstName").IsModified = true;
10. context.SaveChanges();

The code above will result in the author entity being marked as Modified, and SQL being generated to update just the FirstName property:

1. exec sp\_executesql N'SET NOCOUNT ON;
2. UPDATE [Authors] SET [FirstName] = @p0
3. WHERE [AuthorId] = @p1;
4. SELECT @@ROWCOUNT;
5. ',N'@p1 int,@p0 nvarchar(4000)',@p1=1,@p0=N'William'

### **TrackGraph**

The TrackGraph API provides access to individual entities within an object graph, and enables you to execute customised code against each one individually. This is useful in scenarios where you are dealing with complex object graphs that consist of various related entities in differing states.

The following example replicates a scenario where an object graph is constructed outside of the context. Then the TrackGraph method is used to "walk the graph":

1. var author = new Author {
2. AuthorId = 1,
3. FirstName = "William",
4. LastName = "Shakespeare"
5. };
6. author.Books.Add(new Book { AuthorId = 1, BookId = 1, Title = "Hamlet", Isbn = "1234" });
7. author.Books.Add(new Book { AuthorId = 1, BookId = 2, Title = "Othello", Isbn = "4321" });
8. author.Books.Add(new Book { AuthorId = 1, BookId = 3, Title = "MacBeth", Isbn = "5678" });
9. var context = new TestContext();
10. context.ChangeTracker.TrackGraph(author, e => {
11. if((e.Entry.Entity as Author) != null)
12. {
13. e.Entry.State = EntityState.Unchanged;
14. }
15. else
16. {
17. e.Entry.State = EntityState.Modified;
18. }
19. });
20. context.SaveChanges();

In this scenario, it is assumed that the author entity has not been changed, but the books might have been edited. The TrackGraph method takes the root entity as an argument, and a lambda specifying the action to perform. In this case, the root entity, the author has its EntityState set to UnChanged. Setting the EntityState is required for the context to begin tracking the entity. Only then can related entities be discovered. Books have their EntityState set to Modified, which as in the previous examples, will result in SQL that updates every property on the entity:

1. exec sp\_executesql N'SET NOCOUNT ON;
2. UPDATE [Books] SET [AuthorId] = @p0, [Isbn] = @p1, [Title] = @p2
3. WHERE [BookId] = @p3;
4. SELECT @@ROWCOUNT;
5. UPDATE [Books] SET [AuthorId] = @p4, [Isbn] = @p5, [Title] = @p6
6. WHERE [BookId] = @p7;
7. SELECT @@ROWCOUNT;
8. UPDATE [Books] SET [AuthorId] = @p8, [Isbn] = @p9, [Title] = @p10
9. WHERE [BookId] = @p11;
10. SELECT @@ROWCOUNT;
11. ',N'@p3 int,@p0 int,@p1 nvarchar(4000),@p2 nvarchar(150),@p7 int,@p4 int,@p5 nvarchar(4000),
12. @p6 nvarchar(150),@p11 int,@p8 int,@p9 nvarchar(4000),@p10 nvarchar(150)',
13. @p3=1,@p0=1,@p1=N'1234',@p2=N'Hamlet',
14. @p7=2,@p4=1,@p5=N'4321',@p6=N'Othello',
15. @p113,@p8=1,@p9=N'5678',@p10=N'MacBeth'

Since the SQL updates all properties, they all need to be present and have a valid value assigned, otherwise they will be updated to their default values.

In the next example, the object graph is once again constructed outside of the context, but only the Isbn property of the books is modified. Therefore other properties (apart from the entity key) are omitted:

1. var author = new Author {
2. AuthorId = 1,
3. FirstName = "William",
4. LastName = "Shakespeare"
5. };
6. author.Books.Add(new Book { BookId = 1, Isbn = "1234" });
7. author.Books.Add(new Book { BookId = 2, Isbn = "4321" });
8. author.Books.Add(new Book { BookId = 3, Isbn = "5678" });
9. var context = new TestContext();
10. context.ChangeTracker.TrackGraph(author, e => {
11. e.Entry.State = EntityState.Unchanged; //starts tracking
12. if((e.Entry.Entity as Book) != null)
13. {
14. context.Entry(e.Entry.Entity as Book).Property("Isbn").IsModified = true;
15. }
16. });

This time, the method body of the lambda ensures that all entities are tracked in the UnChanged state, and then indicates that the Isbn property is modified. This results in SQL being generated that only updates the Isbn property value:

1. exec sp\_executesql N'SET NOCOUNT ON;
2. UPDATE [Books] SET [Isbn] = @p0
3. WHERE [BookId] = @p1;
4. SELECT @@ROWCOUNT;
5. UPDATE [Books] SET [Isbn] = @p2
6. WHERE [BookId] = @p3;
7. SELECT @@ROWCOUNT;
8. UPDATE [Books] SET [Isbn] = @p4
9. WHERE [BookId] = @p5;
10. SELECT @@ROWCOUNT;
11. ',N'@p1 int,@p0 nvarchar(4000),@p3 int,@p2 nvarchar(4000),@p5 int,@p4 nvarchar(4000)',
12. @p1=1,@p0=N'1234',
13. @p3=2,@p2=N'4321',
14. @p5=3,@p4=N'5678'

##### **Further Reading**

* [**Entity Framework Core TrackGraph For Disconnected Data**](http://www.mikesdotnetting.com/article/303/entity-framework-core-trackgraph-for-disconnected-data)

## **Deleting data via the DbContext**

The approach that you adopt to deleting entities via the DbContext depends on whether the context is currently tracking the entity being deleted or not.

In the following example, the entity to be deleted is obtained by the context, so the context begins tracking it immediately. The DbContext.Remove method results in the entity's EntityState being set to Deleted.

1. context.Remove(context.Authors.Single(a => a.AuthorId == 1));
2. context.SaveChanges();

When SaveChanges is called, a DELETE statement is generated and executed by the database.

1. exec sp\_executesql N'SET NOCOUNT ON;
2. DELETE FROM [Authors]
3. WHERE [AuthorId] = @p0;
4. SELECT @@ROWCOUNT;
5. ',N'@p0 int',@p0=1

This approach actually results in two SQL statements being executed: one to retrieve the entity from the database, and a second to delete it. You can use a stub to represent the entity to be deleted and thereby stop the entity being retrieved from the database:

1. var context = new SampleContext();
2. var author = new Author { AuthorId = 1 };
3. context.Remove(author);
4. context.SaveChanges();

The only property that the stub requires is the primary key value.

### **Setting EntityState**

You can explicitly set the EntityState of an entity to Deleted via the EntityEntry.State property, which is made available by the DbContext.Entry method.

1. var context = new SampleContext();
2. var author = new Author { AuthorId = 1 };
3. context.Entry(author).State = EntityState.Deleted;
4. context.SaveChanges();

### **Related Data**

If the entity that you want to delete has related data, the approach that you take will depend on how the relationship has been configured. A [**fully defined relationship**](https://www.learnentityframeworkcore.com/conventions/one-to-many-relationship#fully-defined-relationship) will have a cascading referential constraint set to Delete or SetNull, as will a [**relationship which has been configured via the Fluent API**](https://www.learnentityframeworkcore.com/configuration/one-to-many-relationship-configuration#cascading-referential-integrity-constraints). In these cases, you can delete the principal and let the database take care of the dependent rows.

Where the referential constraint action is set to NoAction, you need to take care of any related data explicitly. The next example illustrates a relationship configured on a model that doesn't include a foreign key property:

1. public class Author
2. {
3. public int AuthorId { get; set; }
4. public string FirstName { get; set; }
5. public string LastName { get; set; }
6. public ICollection<Book> Books { get; set; }
7. }
8. public class Book
9. {
10. public int BookId { get; set; }
11. public string Title { get; set; }
12. }

By default, this relationship is configured as optional and the [**referential constraint action option**](https://www.learnentityframeworkcore.com/relationships/referential-constraint-action-options) is configured to NoAction. In addition, EF Core introduces a [**shadow property**](https://www.learnentityframeworkcore.com/model/shadow-properties) to represent the foreign key. It is named AuthorId and is applied to the Book entity, and since the relationship is optional, the AuthorId property is nullable. In order to delete the author, you need to delete the relationship between each book and the author. The code to achieve this is as follows:

1. var context = new SampleContext();
2. var author = context.Authors.Single(a => a.AuthorId == 1);
3. var books = context.Books.Where(b => EF.Property<int>(b, "AuthorId") == 1);
4. foreach (var book in books)
5. {
6. author.Books.Remove(book);
7. }
8. context.Remove(author);
9. context.SaveChanges();

The author its retrieved from the database and then its books are obtained and removed one by one from the author's Book collection. Finally, the author is passed to the Remove method of the context. The result is that the books are updated so that their AuthorId value is modified to null:

1. exec sp\_executesql N'SET NOCOUNT ON;
2. UPDATE [Books] SET [AuthorId] = @p0
3. WHERE [BookId] = @p1;
4. SELECT @@ROWCOUNT;
5. UPDATE [Books] SET [AuthorId] = @p2
6. WHERE [BookId] = @p3;
7. SELECT @@ROWCOUNT;
8. UPDATE [Books] SET [AuthorId] = @p4
9. WHERE [BookId] = @p5;
10. SELECT @@ROWCOUNT;
11. ',N'@p1 int,@p0 int,@p3 int,@p2 int,@p5 int,@p4 int',@p1=1,@p0=NULL,@p3=2,@p2=NULL,@p5=3,@p4=NULL

and the author is deleted:

1. exec sp\_executesql N'SET NOCOUNT ON;
2. DELETE FROM [Authors]
3. WHERE [AuthorId] = @p6;
4. SELECT @@ROWCOUNT;
5. ',N'@p6 int',@p6=1

This approach results in four calls being made to the database: one to select the author; one to select the books, one to update the books and a final one to delete the author. It is therefore alays a good idea to make use of the referential integrity constraints to set foreign keys to null or to delete dependents.

# The Entity Framework Core DbSet

The DbSet<TEntity> class represents a collection for a given entity within the model and is the gateway to database operations against an entity. DbSet<TEntity> classes are added as properties to the DbContext and are mapped by default to database tables that take the name of the DbSet<TEntity> property. The DbSet is an implementation of the [**Repository pattern**](http://martinfowler.com/eaaCatalog/repository.html)

1. public class SampleContext : DbContext
2. {
3. public DbSet<Book> Books { get; set; }
4. public DbSet<Author> Authors { get; set; }
5. }
6. public class Author
7. {
8. public int AuthorId { get; set; }
9. public string FirstName { get; set; }
10. public string LastName { get; set; }
11. public ICollection<Book> Books { get; set; }
12. }
13. public class Book
14. { public int BookId { get; set; }
15. public string Title { get; set; }
16. public Author Author { get; set; }
17. public int AuthorId { get; set; }
18. }

In the example above, two DbSet<TEntity> properties have been added to the DbContext class. The first represents a collection of Book objects, which is mapped by convention to a database table named "Books", after the property name. The second DbSet property represents a collection of Author objects, and is mapped to a table named "Authors".

## **Basic operations**

DbSet objects represent collections of entities in memory. Any changes you make to the contents of a DbSet will only be committed to the database if the SaveChanges method of the DbContext is called.

The DbSet class exposes a number of methods that enable you to perform basic CRUD (**C**reate, **R**ead, **U**pdate, **D**elete) operations against entities.

### **Adding an entity**

To add an new entity to the collection represented by the DbSet, you use the DbSet.Add method:

1. var author = new Author{
2. FirstName = "William",
3. LastName = "Shakespeare"
4. };
5. using (var context = new SampleContext())
6. {
7. context.Authors.Add(author); // adds the author to the DbSet in memory
8. context.SaveChanges(); // commits the changes to the database
9. }

### **Retrieving an entity**

If you wish to retrieve a single instance of an entity, you can use the First or Single method depending on whether you expect there to be more than one row matching the criteria. The Single method will result in an exception being raised if more than one matching row exists. It should only be used when querying the context for entities by a unique key:

1. using (var context = new SampleContext())
2. {
3. var author = context.Authors.Single(a => a.AuthorId == 1);
4. }

If you are not certain of retrieving any data based on the criteria you pass in, you should use the FirstOrDefault or SingleOrDefault methods, which return null in the event that no rows match the search criteria:

1. using (var context = new SampleContext())
2. {
3. var author = context.Authors.FirstOrDefault(a => a.LastName == "Shakespeare");
4. }

A convenience method called Find is available which is used to query the context for an entity by primary key value:

1. using (var context = new SampleContext())
2. {
3. var author = context.Authors.Find(1);
4. }

This method is a wrapper around the SingleOrDefault method and checks to see if an entity with the specified key is currently being tracked by the context, and if so, it will be returned. If not, a database query will be executed and the entity will be returned if found. If it isn't found, the Find method will return null. The Single, First, SingleOrDefault and FirstOrDefault methods result in immediate execution of a query against the database.

Asynchronous versions:SingleAsync,FirstAsync, SingleOrDefaultAsync, FirstOrDefaultAsync, FindAsync.

### **Retrieving multiple entities**

The most commonly method used to return multiple entities is the ToList method:

1. using (var context = new SampleContext())
2. {
3. var author = context.Authors.ToList();
4. }

### **Modifying an entity**

To update an entity, make the required changes to the value of the property or properties and call SaveChanges. The entity must be tracked by the context to be updated:

1. using(var context = new SampleContext())
2. {
3. var author = context.Authors.Find(1); // retrieve the entity
4. author.LastName = "Jones"; // amend properties
5. context.SaveChanges(); // commit the changes
6. }

Asynchronous version: SaveChangesAsync.

### **Deleting an entity**

The DbSet class Remove method is used to delete an entity. The entity must be tracked by the context to be removed:

1. using(var context = new SampleContext())
2. {
3. var author = context.Authors.Find(1);
4. context.Authors.Remove(author);
5. context.SaveChanges();
6. }

This approach may result in two SQL queries being executed: one generated by the Single method to select the entity if it isn't already being tracked by the context, and one generated by the Remove method to delete the entity.

You can instead use a **stub** to prevent the need for a SQL query to retrieve the entity. A stub is representation of the entity that is to be operated on. All that is required for a valid stub is the entity key value.

1. using (var context = new SampleContext())
2. {
3. var author = new Author { AuthorId = 1 };
4. context.Authors.Attach(author);
5. context.Authors.Remove(author);
6. context.Savechanges();
7. }

## **Querying data via the DbSet**

Data querying in Entity Framework Core is performed against the DbSet properties of the DbContext. The DbSet represents a collection of entities of a specific type - the type specified by the type parameter.

Queries are specified using Language Integrated Query (LINQ), a component in the .NET Framework that provides query capability against collections in C# or VB. LINQ queries can be written using [**query syntax or method syntax**](https://msdn.microsoft.com/en-us/library/bb397947.aspx). Query syntax shares a resemblance with SQL. The EF Core provider that you use is responsible for translating the LINQ query into the actual SQL to be executed against the database.

The following example uses query syntax to define a query that retrieves all authors ordered by their last name:

1. var data = from a in Authors select a orderby a.LastName

Method syntax uses chained method calls. Many of the method names also resemble SQL. The next example shows the previous query expressed using method syntax:

1. var data = context.Authors.OrderBy(a => a.LastName);

This guide uses method syntax in query examples.

### **Retrieving a single object**

Queries that return single entities are performed using variations of the First, FirstOrDefault, Single, SingleOrDefault and Find methods:

In addition, there are [**asynchronous**](https://www.learnentityframeworkcore.com/asynchronous) versions of each of the above.

#### **First and FirstOrDefault**

The First, and FirstOrDefault methods are intended to be used to return one result from potentially many matches. If you expect at least one record to match the criteria, you can use the First method. If there is a possiblity of no records matching the criteria, use the FirstOrDefault method, which will return null, the default, in the event of no records being found. Both of these methods result in immediate execution of the query, meaning that the SQL is generated and executed against the database as soon as the method call is reached.

The First method results in a SELECT TOP(1) query fetching all columns from the table that maps to the DbSet:

1. var author = context.Authors.First();

Resulting SQL:

1. SELECT TOP(1) [a].[AuthorId], [a].[FirstName], [a].[LastName]
2. FROM [Authors] AS [a]

#### **Single and SingleOrDefault**

The Single and SingleOrDefault methods are used to return a single record where only one should match the criteria specified. The Single method generates a SELECT TOP(2) query. If more than one result is returned by the query, an InvalidOperationException is generated with the message:

Sequence contains more than one element

For this reason, you are very unlikely to use the Single method without specifying some criteria, usually a unique key or index value. You can specify the criteria as a [**lambda expression**](https://msdn.microsoft.com/en-us/library/bb397687.aspx) in a Where method call, or by passing it directly to the Single method call:

1. var author = context.Authors.Where(a => a.AuthorId == 1).Single();
2. var author = context.Authors.Single(a => a.AuthorId == 1);

Both approaches result in identical SQL being generated:

1. SELECT TOP(2) [a].[AuthorId], [a].[FirstName], [a].[LastName]
2. FROM [Authors] AS [a]
3. WHERE [a].[AuthorId] = 1

If it is possible for the query to generate no matching results, you should use the SingleOrDefault method which will return null in that event.

#### **Find**

The DbSet.Find method is familiar to users of earlier versions of Entity Framework that support the DbSet API. The method takes the key value(s) of the entity to be retrieved as opposed to a lambda expression, providing a less verbose option for retrieving single entities by their key:

1. var author = context.Authors.Find(1);

The Find method is shorthand (syntactic sugar) for the SingleOrDefault method, which is why it requires a key value as a parameter. There should be no possibility of the Find method returning multiple results. However, it is possible for the Find method to return null.

### **Retrieving multiple objects**

Queries for retrieving values relating to multiple objects are only executed against a database when the data is iterated over. This is known as deferred execution. Data is iterated over when you use a foreach loop, or a finalising method on the query such as ToList, Sum or Count. Prior to that, the LINQ method calls represent the definition of the query to be executed. The following example essentially defines a query that will retrieve all products from the database:

1. var products = context.Products; // define query
2. foreach(var product in products) // query executed and data obtained from database
3. {
4. ...
5. }

The query is not executed until the foreach loop is reached. The next example demonstrates the use of ToList to force immediate execution:

1. var products = context.Products.ToList(); // define query and force execution

### **Filtering and Ordering**

The Where method is the principal method for filtering results:

1. var products = context.Products.Where(p => p.CategoryId == 1);

The filter criteria are passed in to a lambda as an expression that returns a boolean. The expression can include multiple conditions:

1. var products = context.Prducts.Where(p => p.CategoryId == 1 && p.UnitsInStock < 10);

The OrderBy, OrderByDescending, ThenOrderBy and ThenOrderByDescending methods are used for specifying the order of results:

1. var products = context.Products.OrderBy(p => p.ProuctName);
2. vat categories = context.Categories.OrderBy(c => cCategoryName).ThenOrderBy(c => c.CategoryId);

### **Grouping**

The GroupBy method is used to group results. the following query produces all products in the database grouped by their CategoryId value:

1. var groups = context.Products.GroupBy(p => p.CategoryId);

This results in a collection of types that implement the IGrouping interface. The types have a Key property, which holds the value of the value that was used for grouping i.e. the CategoryId value in this case. Each group has a collection of the elements that were selected, so they can be iterated:

1. var groups = context.Products.GroupBy(p => p.CategoryId);
2. foreach(var group in groups)
3. {
4. //group.Key is the CategoryId value
5. foreach(var product in group)
6. {
7. // you can access individual product properties
8. }
9. }

If you want to use mutiple properties to group by, you will use an anonymous type to represent the Key:

1. var groups = context.Products.GroupBy(p => new {Supplier = p.SupplierId, Country = p.CountryId});

Now the elements of the grouping criteria become properties of the Key:

1. foreach(var group in groups)
2. {
3. //group.Key.SupplierId is the SupplierId value
4. //group.Key.Country is the CountryId value
5. }

**Note:** Grouping is done in-memory in EF Core versions up to 2.1, which means that in the examples above, the data is obtained from the database and then the grouping is performed in the client application by C# code if you are working with older versions of EF Core. The generated SQL orders by the grouping criteria.

Translation of GroupBy was moved to the database in EF Core 2.1.

### **Returning Non-Entity Types**

If you only want to return a subset of properties (as opposed to effectively executing a SELECT \* command), you can project the data into a new form, either as a non-entity type or as an anonymous type. In this example, a type named ProductHeader is defined specifically to act as a container for a subset of data from the products table:

1. public class ProductHeader
2. {
3. public int ProductId { get; set; }
4. public string ProductName { get; set; }
5. }
6. List<ProductHeader> headers = context.Products.Select(p => new ProductHeader{
7. ProductId = p.ProductId,
8. ProductName = p.ProductName
9. }).ToList();

This snippet illustrates an anonymous type being used as the container for the data, resulting in a SQL query that only retrieves the ProductId and ProductName columns from the database:

1. var headers = context.Products.Select(p => new {
2. ProductId = p.ProductId,
3. ProductName = p.ProductName
4. }).ToList();

As of EF Core 2.1, you can also use [**Query Types**](https://www.learnentityframeworkcore.com/query-types) to return non-entity types.

### **Include related data**

The Include method is used to eagerly load related data. You pass in the navigation property that you want to include in the result set. The following query will retrieve all authors and their books:

1. var authors = context.Authors.Include(a => a.Books).ToList();

You can chain calls to the Include method to load data from multiple relationships:

1. var authors = context.Authors
2. .Include(a => a.Biography)
3. .Include(a => a.Books)
4. .ToList();

You can use the ThenInclude method to retrieve data from second and subsequent level relationships. In the next example, the Book entity is assumed to have a navigation property to a Publisher entity:

1. var authors = context.Authors
2. .Include(a => a.Books)
3. .ThenInclude(b => b.Publisher)
4. .ToList();

The query brings back all authors, and their books, and each book's publisher.

The Include method is only effective if you return entity types. If you attempt to use the Include method that returns a non-entity type (see previous section), the Include will be ignored. It will not form part of the SQL that's generated.

See also [**Lazy Loading**](https://www.learnentityframeworkcore.com/lazy-loading)

### **NoTracking Queries**

Any entities that your query returns are automatically [**tracked by the context**](https://www.learnentityframeworkcore.com/dbcontext/change-tracker). In cases where the data is read-only i.e. it is being used for display purposes on a web page and will not be modified during the current request, it is not necessary to have the context perform the extra work required to set up tracking. The AsNoTracking method stops this work being done and can improve performance of an application:

1. var cars = context.Cars.AsNoTracking().ToList();

If you have a series of read-only queries to perform against the same instance of the context, you can configure the tracking behaviour at context-level instead of using the AsNoTracking method in each query:

1. using (var context = new SampleContext())
2. {
3. context.ChangeTracker.QueryTrackingBehavior = QueryTrackingBehavior.NoTracking;
4. var cars = context.Cars.ToList();
5. var customers = context.Customers.ToList();
6. ...
7. }

Non-entity types are not tracked by the context.

## **Adding data via the DbSet**

The key methods for adding entities via the DbSet are

* Add<TEntity>(TEntity entity)
* AddRange<TEntity>(IEnumerable<TEntity> entities)
* AddRange<TEntity>(params TEntity[] entities)

Most often, you will see examples of the generic version of Add but with the type parameter omitted [**because the compiler will infer it**](https://msdn.microsoft.com/en-us/library/twcad0zb.aspx). The following two examples are identical:

1. // with type parameter
2. var author = new Author{ FirstName = "William", LastName = "Shakespeare" };
3. context.Authors.Add<Author>(author);
4. // without type parameter
5. var author = new Author{ FirstName = "William", LastName = "Shakespeare" };
6. context.Authors.Add(author);

The context begins tracking the entity that was passed in to the Add method and applies an EntityState value of Added to it. The context also applies the same EntityState value of Added to all other objects in the object graph that aren't already being tracked by the context. In the next example, the Added state is also applied to the books:

1. var context = new SampleContext();
2. var author = new Author {
3. FirstName = "William",
4. LastName = "Shakespeare",
5. Books = new List<Book>
6. {
7. new Book { Title = "Hamlet"},
8. new Book { Title = "Othello" },
9. new Book { Title = "MacBeth" }
10. }
11. };
12. context.Authors.Add(author);

The books are added by virtue of the fact that they are referenced through the Books property of the author. In the next example, the books will **not** be added:

1. var author = new Author { FirstName = "William", LastName = "Shakespeare" };
2. var hamlet = new Book { Title = "Hamlet", Author = author };
3. var othello = new Book { Title = "Othello", Author = author };
4. var macbeth = new Book { Title = "MacBeth", Author = author };
5. context.Authors.Add(author);
6. context.SaveChanges();

Although the author has been assigned to the Author property of each of the books that have been instantiated, the author entity is unaware of these relationships. It's Books property is still null and the books are not added to the context.

### **Adding Multiple Records**

The AddRange method is used for adding multiple objects to the database in one method call. The code in the next example is very similar to the previous example, but the AddRange method is used to save all the books and the author to the database in one go:

1. var context = new SampleContext();
2. var author = new Author { FirstName = "Stephen", LastName = "King" };
3. var books = new List<Book> {
4. new Book { Title = "It", Author = author },
5. new Book { Title = "Carrie", Author = author },
6. new Book { Title = "Misery", Author = author }
7. };
8. context.Books.AddRange(books);
9. context.SaveChanges();

The AddRange method takes an IEnumerable<object>. The author forms part of the object graph for at least one of the books, so it is added too.

The other version of the AddRange method takes a params array of entities, providing the facility to add a variable number of entities to the database without the need to create a collection for them:

1. var context = new SampleContext();
2. var author = new Author { FirstName = "William", LastName = "Shakespeare" };
3. var hamlet = new Book { Title = "Hamlet", Author = author };
4. var othello = new Book { Title = "Othello", Author = author };
5. var macbeth = new Book { Title = "MacBeth", Author = author };
6. context.Books.AddRange(hamlet, othello, macbeth);
7. context.SaveChanges();

When the SaveChanges method is called on the DbContext, all entities with an EntityState of Added will be inserted into the database. The ordering of the SQL to insert objects is managed in such a way to ensure that principals are inserted first and their primary key value is then available to be applied to the foreign key of dependent objects.

##### **Further Reading**

* [**Adding data via the DbContext**](https://www.learnentityframeworkcore.com/dbcontext/adding-data)

## **Modifying data via the DbSet**

The approach that you adopt to modifying entities via the DbSet depends on whether the context is currently tracking the entity being modified or not.

In the following example, the entity is obtained by the context, so the context begins tracking it immediately. When you alter property values on a tracked entity, the context changes the EntityState for the entity to Modified and the ChangeTracker records the old property values and the new property values. When SaveChanges is called, an UPDATE statement is generated and executed by the database.

1. var author = context.Authors.Find(1);
2. author.FirstName = "Bill";
3. context.SaveChanges();

Since the ChangeTracker tracks which properties have been modified, the context will issue a SQL statement that updates only those properties that were changed:

1. exec sp\_executesql N'SET NOCOUNT ON;
2. UPDATE [Authors] SET [FirstName] = @p0
3. WHERE [AuthorId] = @p1;
4. SELECT @@ROWCOUNT;
5. ',N'@p1 int,@p0 nvarchar(4000)',@p1=1,@p0=N'Bill'

### **Disconnected Scenario**

In a disconnected scenario such as an ASP.NET application, changes to an existing entity's property values can take place in a controller or service method, well away from the context. In these cases, the context needs to be informed that the entity is in a modified state. This can be achieved by using the DbSet<T>.Update method (which is new in EF Core).

### **DbSet Update**

The DbSet<T> class provides Update and UpdateRange methods for working with individual or multiple entities.

1. public void Save(Author author)
2. {
3. context.Authors.Update(author);
4. context.SaveChanges();
5. }

This method results in the entity being tracked by the context as Modified. The context doesn't have any way of identifying which property values have been changed, and will generate SQL to update all properties. Any related entities (such as a collection of books in this example) will also be tracked in the Modified state, resulting in UPDATE statements being generated for each of them. If the related entity doesn't have a key value assigned, it will be marked as Added, and an INSERT statement will be generated instead.

##### **Further Reading**

* [**Entity Framework Core TrackGraph For Disconnected Data**](http://www.mikesdotnetting.com/article/303/entity-framework-core-trackgraph-for-disconnected-data)

## **Deleting data via the DbSet**

The approach that you adopt to deleting entities via the DbSet depends on whether the context is currently tracking the entity being deleted or not.

In the following example, the entity to be deleted is obtained by the context, so the context begins tracking it immediately. The DbSet<T>.Remove method results in the entity's EntityState being set to Deleted.

1. context.Authors.Remove(context.Authors.Find(1));
2. context.SaveChanges();

When SaveChanges is called, a DELETE statement is generated and executed by the database.

1. exec sp\_executesql N'SET NOCOUNT ON;
2. DELETE FROM [Authors]
3. WHERE [AuthorId] = @p0;
4. SELECT @@ROWCOUNT;
5. ',N'@p0 int',@p0=1

This approach actually results in two SQL statements being executed: one to retrieve the entity from the database, and a second to delete it. You can use a stub to represent the entity to be deleted and thereby stop the entity being retrieved from the database:

1. var context = new SampleContext();
2. var author = new Author { AuthorId = 1 };
3. context.Authors.Remove(author);
4. context.SaveChanges();

The only property that the stub requires is the primary key value.

### **Related Data**

If the entity that you want to delete has related data, the approach that you take will depend on how the relationship has been configured. A [**fully defined relationship**](https://www.learnentityframeworkcore.com/configuration/one-to-many-relationship-configuration#fully-defined-relationship) will have a cascading referential constraint set to Delete or SetNull, as will a [**relationship which has been configured via the Fluent API**](https://www.learnentityframeworkcore.com/configuration/one-to-many-relationship-configuration#cascading-referential-integrity-constraints). In these cases, you can delete the principal and let the database take care of the dependent rows.

Where the referential constraint action is set to NoAction, you need to take care of any related data explicitly. The next example illustrates a relationship configured on a model that doesn't include a foreign key property:

1. public class Author
2. {
3. public int AuthorId { get; set; }
4. public string FirstName { get; set; }
5. public string LastName { get; set; }
6. public ICollection<Book> Books { get; set; }
7. }
8. public class Book
9. {
10. public int BookId { get; set; }
11. public string Title { get; set; }
12. }

By default, this relationship is configured as optional and the [**referential constraint action option**](https://www.learnentityframeworkcore.com/relationships/referential-constraint-action-options) is configured to NoAction. In addition, EF Core introduces a [**shadow property**](https://www.learnentityframeworkcore.com/model/shadow-properties) to represent the foreign key. It is named AuthorId and is applied to the Book entity, and since the relationship is optional, the AuthorId property is nullable. In order to delete the author, you need to delete the relationship between each book and the author. The code to achieve this is as follows:

1. var context = new SampleContext();
2. var author = context.Authors.Find(1);
3. var books = context.Books.Where(b => EF.Property<int>(b, "AuthorId") == 1);
4. foreach (var book in books)
5. {
6. author.Books.Remove(book);
7. }
8. context.Authors.Remove(author);
9. context.SaveChanges();

The author its retrieved from the database and then its books are obtained and removed one by one from the author's Book collection. Finally, the author is passed to the Remove method of the context. The result is that the books are updated so that their AuthorId value is modified to null:

1. exec sp\_executesql N'SET NOCOUNT ON;
2. UPDATE [Books] SET [AuthorId] = @p0
3. WHERE [BookId] = @p1;
4. SELECT @@ROWCOUNT;
5. UPDATE [Books] SET [AuthorId] = @p2
6. WHERE [BookId] = @p3;
7. SELECT @@ROWCOUNT;
8. UPDATE [Books] SET [AuthorId] = @p4
9. WHERE [BookId] = @p5;
10. SELECT @@ROWCOUNT;
11. ',N'@p1 int,@p0 int,@p3 int,@p2 int,@p5 int,@p4 int',@p1=1,@p0=NULL,@p3=2,@p2=NULL,@p5=3,@p4=NULL

and the author is deleted:

1. exec sp\_executesql N'SET NOCOUNT ON;
2. DELETE FROM [Authors]
3. WHERE [AuthorId] = @p6;
4. SELECT @@ROWCOUNT;
5. ',N'@p6 int',@p6=1

This approach results in four calls being made to the database: one to select the author; one to select the books, one to update the books and a final one to delete the author. It is therefore alays a good idea to make use of the referential integrity constraints to set foreign keys to null or to delete dependents.

# The Entity Framework Core Model

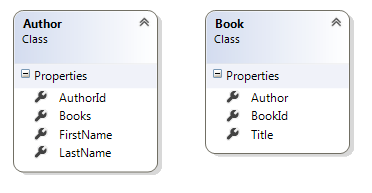
An Entity Framework Core model is a conceptual model of an application's domain. The domain includes all topics relevant to the problem solving areas of interest to the application users. The model includes data and can also include behaviour. Typically, models for CRUD applications don't tend to incorporate a lot of behaviour.

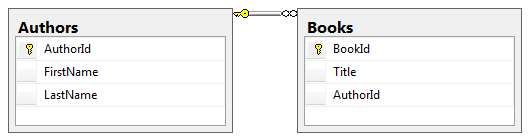
Models are expressed as a collection of classes written in C#. Each class represents an entity (a.k.a. business object, domain object) within the application domain.

The following example illustrates a minimal model for an application that is concerned with books and their authors:

1. public class Author
2. {
3. public int AuthorId { get; set; }
4. public string LastName { get; set; }
5. public List<Book> Titles { get; set; } = new List<Book>();
6. }
7. public class Book
8. {
9. public int BookId { get; set; }
10. public string Title { get; set; }
11. public Author Author { get; set; }
12. }

When working with Entity Framework Core, it is usual for the model structure to closely match the schema of the database.





If the model differs from the database schema, it can be mapped to the target tables and columns using the [**Fluent API configuration**](https://www.learnentityframeworkcore.com/configuration/fluent-api) capability of Entity Framework Core.

The model can be designed by hand coding, or it can be [**reverse engineered from an existing database**](https://www.learnentityframeworkcore.com/walkthroughs/existing-database).

## **Starting with an existing database**

Previous versions of Entity Framework support a Database-First approach to development. In this approach, you reverse-engineer a model from an existing database, resulting in the generation of an EDMX file that contains the model definition and mapping information. Tooling support for the EDMX file was dropped in Entity Framework Core in favour of using commands to reverse-engineer class files for the model from an existing database schema. This approach is known as Code First to an existing database.

### **Command Line Interface**

The following example illustrates how to do use code first to to generate a model from a SQL Server database in a new console application using the CLI tools.

First, create a folder for the project:

1. > mkdir EFCoreScaffoldexample

Then navigate to it:

1. > cd EFCoreScaffoldExample

Then create a new project:

1. > dotnet new console

Add the Entity Framework Core and Tools packages to the project:

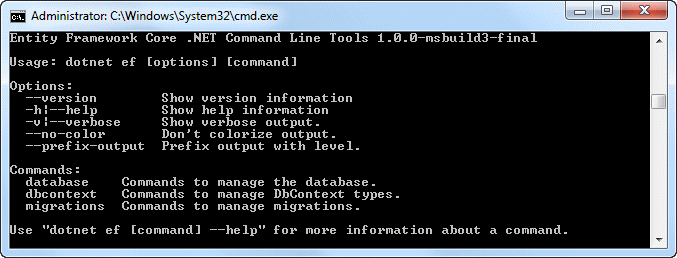
1. > dotnet add package Microsoft.EntityFrameworkCore.SqlServer
2. > dotnet add package Microsoft.EntityFrameworkCore.Design

The first package is the EF Core provider for SQL Server. The second package contains the Entity Framework Core commands. Both of these packages are required for any Entity Framework Core application that targets SQL Server.

Test to see if **[ef commands](https://www.learnentityframeworkcore.com/migrations/commands/cli-commands)** are available to you:

1. dotnet ef -h

This should result in the initial help for the EF tools being displayed:



If you get errors like this:

Could not execute because the specified command or file was not found. Possible reasons for this include:  
\* You misspelled a built-in dotnet command.  
\* You intended to execute a .NET Core program, but dotnet-ef does not exist.  
\* You intended to run a global tool, but a dotnet-prefixed executable with this name could not be found on the PATH.

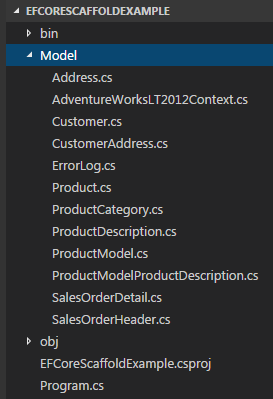
try installing the EF Core tool by executing the following command:

1. dotnet tool install --global dotnet-ef

You use the **[DbContext Scaffold](https://www.learnentityframeworkcore.com/migrations/commands/cli-commands" \l "dbcontext-scaffold)** command to generate the model. The command has two required arguments - a connection string and a provider. The connection string will depend on your environment and database provider. The provider argument is the Entity Framework provider for your chosen database. This example uses the **[AdventureWorks sample database](https://github.com/Microsoft/sql-server-samples/releases/tag/adventureworks)** for SQL server provided by Microsoft.

1. > dotnet ef dbcontext scaffold "Server=.\;Database=AdventureWorksLT2012;Trusted\_Connection=True;" Microsoft.EntityFrameworkCore.SqlServer -o Model

Once you have executed the command, you will see that a folder named Model has been created in the project folder, containing a collection of class files representing the entities in addition to a file for the DbContext class:



The -o option (or alternatively --output-dir) specifies the directory where the class files will be generated. If it is omitted, the class files will be generated in the project directory (where the .csproj file is located).

The DbContext class will take the name of the database plus "Context", You can override this using the -c or --context option e.g.

1. > dotnet ef dbcontext scaffold "Server=.\;Database=AdventureWorksLT2012;Trusted\_Connection=True;" Microsoft.EntityFrameworkCore.SqlServer -o Model -c "AdventureContext"

#### **Model Configuration**

The resulting model is configured using the [**Fluent API**](https://www.learnentityframeworkcore.com/configuration/fluent-api) by default, which is the recommended approach. The configurations are placed in the generated context's OnModelCreating method. However, if you prefer to use [**Data Annotations**](https://www.learnentityframeworkcore.com/configuration/data-annotation-attributes) for configuration, you can use the -d or --data-annotations switch:

1. dotnet ef dbcontext scaffold "Server=.\;Database=AdventureWorksLT2012;Trusted\_Connection=True;" Microsoft.EntityFrameworkCore.SqlServer -d

Since Data Annotations only cover a subset of configuration options, you are likely to find that the Fluent API has also been used to configure the model.

### **Updating the Model**

The recommended approach to keeping changes to the database in sync with the generated model is to use [**migrations**](https://www.learnentityframeworkcore.com/migrations) i.e. to make the changes to the model first, and then use tools to generate code that propagates the modifications to the database. However, depending on your circumstances, this may not always be an option. If you need to re-scaffold the model after database schema changes have been made, you can do so by specifying the -f or --force option e.g.:

1. dotnet ef dbcontext scaffold "Server=.\;Database=AdventureWorksLT2012;Trusted\_Connection=True;" Microsoft.EntityFrameworkCore.SqlServer --force

All of the class files will be overwritten, which means that any amendments that you might have made to them e.g. adding attributes or additional members, will be lost. You can mitigate this by opting to use the [**Fluent API for configuration and using separate configuration classes**](https://www.learnentityframeworkcore.com/configuration/fluent-api#separate-configuration-classes). In addition, you can use partial classes to declare additional properties that don't map to columns in the database tables.

### **Visual Studio**

If you are working with Visual Studio, you can use the [**Package Manager Console commands**](https://www.learnentityframeworkcore.com/migrations/commands/pmc-commands#scaffold-dbcontext) to generate the the code files for the model. The equivalent command to the last CLI command just above is:

1. PM> Scaffold-DbContext "Server=.\;Database=AdventureWorksLT2012;Trusted\_Connection=True;" Microsoft.EntityFrameworkCore.SqlServer -OutputDir Model -Context "AdventureContext" -DataAnnotations

### **Migrations**

In previous versions of Entity Framework, it was possible, once the model has been created, to add a migration that did not affect the schema of the existing database using the -ignorechanges option. This option does not exist in Entity Framework Core, so the workaround is to create a first migration and then to delete or comment out the contents of the Up method prior to applying the migration to the database. This will result in a model and a database schema that match.

## **Entity Framework Core Shadow Properties**

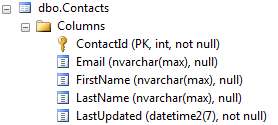
Shadow properties in Entity Framework Core are properties that do not feature as part of the entity class but can be included in the model and are mapped to database columns.

Shadow properties are useful in a number of scenarios. For example, they can be used for extending a model where you do not have access to the source code of the entity classes. They can also be used in cases where you prefer that the class definitions of your entities don't include "relational artifacts" such as foreign key columns or metadata properties like DateCreated or LastUpdated, or a **[rowversion property](https://www.learnentityframeworkcore.com/concurrency" \l "adding-a-rowversion-property)**.

Shadow properties can be configured using the Fluent API in the OnModelCreating method. This example illustrates a LastUpdated shadow property being configured on the Contact entity:

1. public class SampleContext : DbContext
2. {
3. public DbSet<Contact> Contacts { get; set; }
5. protected override void OnModelCreating(ModelBuilder modelBuilder)
6. {
7. modelBuilder.Entity<Contact>()
8. .Property<DateTime>("LastUpdated");
9. }
10. }
11. public class Contact
12. {
13. public int ContactId { get; set; }
14. public string FirstName { get; set; }
15. public string LastName { get; set; }
16. public string Email { get; set; }
17. }

The version of the Property method that takes a string is used, and the data type of the property is specified as a type parameter. The shadow property will be included in migrations, resulting in a column being added to the Contacts table named "LastUpdated".



Once a shadow property has been declared, it can be further configured using the Fluent API just like any other property in the model. The following code illustrates a Version shadow property being added to the Author entity in the OnModelCreating method, and then being configured to take part in concurrency management:

1. public class SampleContext : DbContext
2. {
3. public DbSet<Author> Authors { get; set; }
5. protected override void OnModelCreating(ModelBuilder modelBuilder)
6. {
7. modelBuilder.Entity<Author>()
8. .Property<byte[]>("Version")
9. .IsRowVersion();
10. }
11. }
12. public class Author
13. {
14. public int AuthorId { get; set; }
15. public string FirstName { get; set; }
16. public string LastName { get; set; }
17. public ICollection<Book> Books { get; set; }
18. }

#### **Setting the value of shadow properties**

You can access the shadow property via the DbContext.Entry property and set its value via the CurrentValue property:

1. var context = new SampleContext();
2. var contact = new Contact { FirstName = "John", LastName = "Doe" };
3. context.Add(contact);
4. context.Entry(contact).Property("LastUpdated").CurrentValue = DateTime.UtcNow;
5. context.SaveChanges();

You can also set values via the ChangeTracker API through its Entries() method. This offers a more logical way to set a LastUpdated value by overriding the SaveChanges method:

1. public class SampleCpntext: DbContext
2. {
3. protected override void OnModelBuilding(ModelBuider modelBuilder)
4. {
5. modelBuilder.Entity<Contact>()
6. .Property<DateTime>("LastUpdated");
7. }
8. public override int SaveChanges()
9. {
10. ChangeTracker.DetectChanges();
12. foreach (var entry in ChangeTracker.Entries())
13. {
14. if(entry.State == EntityState.Added || entry.State == EntityState.Modified)
15. {
16. entry.Property("LastUpdated").CurrentValue = DateTime.UtcNow;
17. }
18. }
19. return base.SaveChanges();
20. }
21. public DbSet<Contact> Contacts { get; set; }
22. }
23. public class Contact
24. {
25. public int ContactId { get; set; }
26. public string FirstName { get; set; }
27. public string LastName { get; set; }
28. public string Email { get; set; }
29. }

#### **Querying with shadow properties**

Shadow properties can be referenced in LINQ queries via the static Property method of the EF utility class:

1. var contacts = context.Contacts.OrderBy(contact => EF.Property<DateTime>(contact, "LastUpdated"));

Or, if you prefer to use the C# 6 using static directive:

1. using static Microsoft.EntityFrameworkCore.EF;
2. using static System.Console;
3. ...
4. var contacts = context.Contacts.OrderBy(contact => Property<DateTime>(contact, "LastUpdated"));

#### **Automatic generation of shadow properties**

Shadow properties are generated by EF Core for dependant entities in relationships where there is no foreign key property defined in the dependent entity.

# Introduction to Relationships

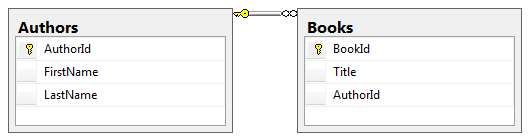
Relational databases are data stores whose structure is based on how items of data are related to each other.

A key benefit to taking a relational view of data is to reduce duplication. For example, you might want to record data about people in a town. If you took a non-relational approach to recording this data, you would store the person's name together with their address, place of work, school and so on as individual data items. Where multiple people live at the same address or go to the same school, you would record the address or school details in multiple places. If the school name ever changed, you would have to update it in every data item in which it appeared, which is a time-consuming and error-prone task.

In a relational database, each entity such as the person, the school, the place of work is stored in separate tables and unique instances if the entity are identified by a Primary Key value. Relationships or associations between entities are defined in a database by the existence of Foreign Keys.

## **One To Many**

The following diagram depicts a relationship between the Books and Authors tables in a database:



Each table has a Primary Key (PK) that uniquely defines each instance of an entity (or row) within the table. The Books table's PK is BookId, and the Authors table's PK is AuthorId. The AuthorId column in the Books table is a Foreign Key (FK), linking a book to its author. Book is the dependent entity in the relationship. It's integrity depends on a valid reference to an author. Author becomes the principal entity. Using foreign keys, you can link one author row in the database to many book rows. This type of relationship is the most commonly found and is known as a **One-To-Many** relationship.

Depending on the diagramming software that you use, the side of the relationship that has a multiplicity of 1 is normally depicted by a figure **1** or a key. The side of the relationship with a multiplicity of many is usually depicted by an asterisk (\*) or an infinity symbol (∞). Multiplicity describes the potential number of items that can be found at one end of a relationship.

## **Navigation Properties**

Relationships between entities in an Entity Framework model are defined by Navigation Properties. A navigation property is one that the database provider being used cannot map to a primitive (or scalar) type. The following code depicts the model representation of the database example above:

1. public class Author
2. {
3. public int AuthorId { get; set; }
4. public string FirstName { get; set; }
5. public string LastName { get; set; }
6. public ICollection<Book> Books { get; set; }
7. }
8. public class Book
9. {
10. public int BookId { get; set; }
11. public string Title { get; set; }
12. public Author Author { get; set; }
13. }

Both classes contain properties whose types can be mapped to existing database types - int, string but they also contain properties that cannot be mapped. There is no equivalent type in databases for the Book type or the Author type. Therefore they are viewed as navigation properties.

The definition of the Book class allows each book to have at most one author through the Author navigation property (a Reference navigation property having a multiplicity of zero or one), and the definition of the Author class allows each author to have many books through the Books navigation property (a Collection navigation property having a multiplicity of many). Together, they define a one-to-many relationship. The principal entity in a one-to-many relationship is the one that has the collection navigation property, and the dependent entity is the one with the reference navigation property.

Navigation properties enable navigation of the association between the two types through code:

1. book.Author = new Author();
2. foreach (var book in author.Books)
3. {
4. ...

#### **Further reading**

* [**Managing One To Many Relationships**](https://www.learnentityframeworkcore.com/relationships/managing-one-to-many-relationships)
* [**Entity Framework One To Many Relationship Conventions**](https://www.learnentityframeworkcore.com/conventions/one-to-many-relationship)
* [**Entity Framework One To Many Relationship Configurations**](https://www.learnentityframeworkcore.com/configuration/one-to-many-relationship-configuration)

## **Many To Many**

The second most common type of relationship is known as a **Many To Many** relationship. The following diagram shows how this appears in a database diagram.

Each book can belong to many categories and each category can contain many books. This type of relationship is managed in a database through the use of a join table (also known among other things as a bridging, junction or linking table). This type of relationship is defined in code by the inclusion of collection properties in each of the entities:

1. public class Book
2. {
3. public int BookId { get; set; }
4. public string Title { get; set; }
5. public Author Author { get; set; }
6. public ICollection<Category> Categories { get; set; }
7. }
8. public class Category
9. {
10. public int CategoryId { get; set; }
11. public string CategoryName { get; set; }
12. public ICollection<Book> Books { get; set; }
13. }

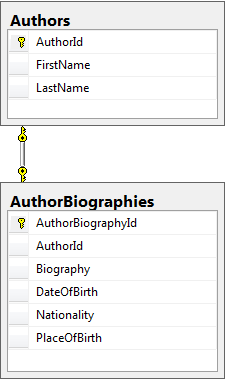
Note: While in previous versions of EF, this approach was sufficient to have EF generate the appropriate tables and form the correct relationships, in EF Core 1.0, it isn't. It is necessary to include an entity within the model to represent the join table.

#### **Further reading**

* [**Entity Framework Core Many To Many Relationship Configuration**](https://www.learnentityframeworkcore.com/conventions/many-to-many-relationship)

## **One To One**

A one to one (or more usually a one to zero or one) relationship exists when only one row of data in the principal table is linked to zero or one row in a dependent table. The following diagram illustrates this relationship between an Authors table and the AuthorBiographies table:



The following example shows how this relationship is modelled in code:

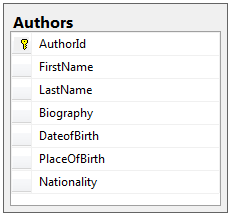
1. public class Author
2. {
3. public int AuthorId { get; set; }
4. public string FirstName { get; set; }
5. public string LastName { get; set; }
6. public AuthorBiography Biography { get; set; }
7. }
8. public class AuthorBiography
9. {
10. public int AuthorBiographyId { get; set; }
11. public string Biography { get; set; }
12. public DateTime DateOfBirth { get; set; }
13. public string PlaceOfBirth { get; set; }
14. public string Nationality { get; set; }
15. public int AuthorId { get; set; }
16. public Author Author { get; set; }
17. }

In this example of a one-to-one relationship, each Author can have one, and only one biography.

One reason for implementing this kind of relationship is when you are working with inheritance. For example, you may have a Vehicle entity, with sub classes such as Car, Truck, Motorcycle etc. Other reasons include database design and/or efficiency. For example, you may want to apply extra database security to the dependent table because it contains confidential information (an employee's health record, for example), or you just want to move data that isn't referenced very often into a separate table to improve search and retrieval times for data that is used all the time.

## **Table Splitting**

EF Core (from version 2.0) also supports table splitting, a technique that enables you to use a single table to represent both entities in a one-to-one relationship where separation into multiple tables is not required. Using this feature, the one-to-one relationship illustrated above will be stored in a database table together:



Each entity needs to be defined separately, and they must share the same primary key value. This type of relationship requires [**configuration**](https://www.learnentityframeworkcore.com/configuration/one-to-one-relationship-configuration#configuring-a-relationship-that-uses-table-splitting)

#### **Further Reading**

* [**Entity Framework One To One Relationship Conventions**](https://www.learnentityframeworkcore.com/conventions/one-to-one-relationship)
* [**Entity Framework One To One Relationship Configurations**](https://www.learnentityframeworkcore.com/configuration/one-to-one-relationship-configuration)

## **Managing One To Many Relationships With Entity Framework Core**

Entity Framework Core offers a number of approaches to the creation and modification of one-to-many relationships.

### **Creating relationships**

This first section explores a number of ways in which relationships can be created between an existing principal entity and newly created dependents.

#### **Add the dependent to the principal's collection property**

In this example, an existing author is referenced using the DbSet.Find method. A newly created book is then added to the author's Books collection navigation property.

1. using(var db = new TestContext())
2. {
3. var book = new Book { Title = "King Lear" };
4. var author = db.Authors.Find(1);
5. author.Books.Add(book);
6. db.SaveChanges();
7. }

This approach potentially results in two calls to the database. The Find method will first check the cache to see if an object of the same type with the same key value is being tracked by the context. If the author is not currently being tracked, a command is issued to select it from the database. Then a separate command inserts the book into the Books table:

1. exec sp\_executesql N'SELECT TOP(1) [e].[AuthorId], [e].[FirstName], [e].[LastName]
2. FROM [Authors] AS [e]
3. WHERE [e].[AuthorId] = @\_\_get\_Item\_0',N'@\_\_get\_Item\_0 int',@\_\_get\_Item\_0=1
4. exec sp\_executesql N'SET NOCOUNT ON;
5. INSERT INTO [Books] ([AuthorId], [Title])
6. VALUES (@p0, @p1);
7. SELECT [BookId]
8. FROM [Books]
9. WHERE @@ROWCOUNT = 1 AND [BookId] = scope\_identity();
10. ',N'@p0 int,@p1 nvarchar(255)',@p0=1,@p1=N'King Lear'

### **Attach a fake representing the principal then add the dependent to it**

A "fake" or "stub" is used to represent the author in this example, which is then attached to the context. EF Core begins to track the fake author in the Unchanged state. When the book is added to the Books collection of the tracked author, the book's state is set to Added.

1. using(var db = new TestContext())
2. {
3. var book = new Book { Title = "As You Like It" };
4. var author = new Author { AuthorId = 1 };
5. db.Attach(author);
6. author.Books.Add(book);
7. db.SaveChanges();
8. }

This approach results in one database operation - inserting the new book into the database.

1. exec sp\_executesql N'SET NOCOUNT ON;
2. INSERT INTO [Books] ([AuthorId], [Title])
3. VALUES (@p0, @p1);
4. SELECT [BookId]
5. FROM [Books]
6. WHERE @@ROWCOUNT = 1 AND [BookId] = scope\_identity();
7. ',N'@p0 int,@p1 nvarchar(255)',@p0=1,@p1=N'As You Like It'

#### **Set the Foreign Key value of the new book explicitly**

Both of the previous examples offer a more object-oriented approach to creating relationships between entities. The next example owes much more to working with relational data in a database. The book is created and has a valid value applied to its foreign key property. There is no reference to an Author entity in this code:

1. using(var db = new TestContext())
2. {
3. var book = new Book { Title = "The Winters Tale", AuthorId = 1 };
4. db.Add(book);
5. db.SaveChanges();
6. }

As with the previous approach, this results in just one database operation:

1. exec sp\_executesql N'SET NOCOUNT ON;
2. INSERT INTO [Books] ([AuthorId], [Title])
3. VALUES (@p0, @p1);
4. SELECT [BookId]
5. FROM [Books]
6. WHERE @@ROWCOUNT = 1 AND [BookId] = scope\_identity();
7. ',N'@p0 int,@p1 nvarchar(255)',@p0=1,@p1=N'The Winters Tale'

### **Modifying Relationships**

The next examples look at modifying relationships between entities.

#### **Use fakes to move the dependent to a different principal's collection**

This example results in an existing book with a key value of 4 being assigned to the author that has a key value of 1. Once again, fakes are attached to the context, telling the context that the entities with the specified keys already exist.

1. using(var db = new TestContext())
2. {
3. var book = new Book {BookId = 4};
4. db.Attach(book);
5. var author = new Author {AuthorId = 1};
6. db.Attach(author);
7. author.Books.Add(book);
8. db.SaveChanges();
9. }

One database operation is generated, updating the foreign key value of the book.

1. exec sp\_executesql N'SET NOCOUNT ON;
2. UPDATE [Books] SET [AuthorId] = @p0
3. WHERE [BookId] = @p1;
4. SELECT @@ROWCOUNT;
5. ',N'@p1 int,@p0 int',@p1=4,@p0=1

#### **Alter the foreign key value of a fake**

In this example, the foreign key value of the fake book is explicity set.

1. using(var db = new TestContext())
2. {
3. var book = new Book { BookId = 4 };
4. db.Attach(book);
5. book.AuthorId = 2;
6. db.SaveChanges();
7. }

Once again, only one database operation is generated.

1. exec sp\_executesql N'SET NOCOUNT ON;
2. UPDATE [Books] SET [AuthorId] = @p0
3. WHERE [BookId] = @p1;
4. SELECT @@ROWCOUNT;
5. ',N'@p1 int,@p0 int',@p1=4,@p0=2

#### **Add the dependent to the new principal's collection property**

The Find method is used to obtain a reference to the book and the author that is is to be assigned to.

1. using(var db = new TestContext())
2. {
3. var book = db.Books.Find(4);
4. var author = db.Authors.Find(2);
5. author.Books.Add(book);
6. db.SaveChanges();
7. }

From a database operation point of view, this approach can be quite expensive, resulting in possibly three SQL commands being executed.

1. exec sp\_executesql N'SELECT TOP(1) [e].[BookId], [e].[AuthorId], [e].[Title]
2. FROM [Books] AS [e]
3. WHERE [e].[BookId] = @\_\_get\_Item\_0',N'@\_\_get\_Item\_0 int',@\_\_get\_Item\_0=4
4. exec sp\_executesql N'SELECT TOP(1) [e].[AuthorId], [e].[FirstName], [e].[LastName]
5. FROM [Authors] AS [e]
6. WHERE [e].[AuthorId] = @\_\_get\_Item\_0',N'@\_\_get\_Item\_0 int',@\_\_get\_Item\_0=2
7. exec sp\_executesql N'SET NOCOUNT ON;
8. UPDATE [Books] SET [AuthorId] = @p0
9. WHERE [BookId] = @p1;
10. SELECT @@ROWCOUNT;
11. ',N'@p1 int,@p0 int',@p1=4,@p0=2

### **Removing relationships**

#### **Delete the principal**

Deleting the principal will ensure that the action specified by the Referential Constraint Action enumeration will be enforced. For required relationships, the dependents will all be deleted. If the relationship is optional, the foreign key values of the dependents will be set to null.

1. using(var db = new TestContext())
2. {
3. var authorToDelete = new Author { AuthorId = 1 };
4. db.Authors.Remove(authorToDelete);
5. db.SaveChanges();
6. }
7. exec sp\_executesql N'SET NOCOUNT ON;
8. DELETE FROM [Authors]
9. WHERE [AuthorId] = @p0;
10. SELECT @@ROWCOUNT;
11. ',N'@p0 int',@p0=2

#### **Set the Foreign Key value to null (optional relationships only)**

If the relationship is optional, you can set the foreign key value to null to delete the relationship

1. using(var db = new TestContext())
2. {
3. var book = context.Books.Find(1);
4. book.AuthorId = null;
5. db.SaveChanges();
6. }

Again, use of the Find method could result in two calls to the database

1. exec sp\_executesql N'SELECT TOP(1) [e].[BookId], [e].[AuthorId], [e].[Title]
2. FROM [Books] AS [e]
3. WHERE [e].[BookId] = @\_\_get\_Item\_0',N'@\_\_get\_Item\_0 int',@\_\_get\_Item\_0=1
4. exec sp\_executesql N'SET NOCOUNT ON;
5. UPDATE [Books] SET [AuthorId] = @p0
6. WHERE [BookId] = @p1;
7. SELECT @@ROWCOUNT;
8. ',N'@p1 int,@p0 int',@p1=1,@p0=NULL

You can also perform this operation using fakes and the DBContext.Entry:

1. var book = new Book { BookId = 2 } ;
2. db.Attach(book);
3. book.AuthorId = null;
4. db.Entry(book).Property(p => p.AuthorId).IsModified = true;
5. db.SaveChanges();
6. exec sp\_executesql N'SET NOCOUNT ON;
7. UPDATE [Books] SET [AuthorId] = @p0
8. WHERE [BookId] = @p1;
9. SELECT @@ROWCOUNT;
10. ',N'@p1 int,@p0 int',@p1=2,@p0=NULL

#### **Remove the dependent from the principal's collection**

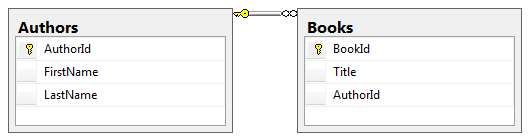
You can remove the dependent from the principal's collection property. Note that this deletes the book in a required relationship, and sets the dependent's foreign key value to null in an optional relationship:

1. using(var db = new TestContext())
2. {
3. var book = db.Books.Find(1);
4. var author = db.Authors.Find(1);
5. author.Books.Remove(book);
6. db.SaveChanges();
7. }

## **Referential Constraint Action Options**

In order to maintain referential integrity of your data, you can specify the action that the database should take in the event that an attempt is made to delete or update a primary key currently being referenced by a foreign key in another table. You can do this via the ON DELETE or ON UPDATE clause when the foreign key constraint is being defined.

Here's a diagram that shows a simple database of books and authors to help illustrate each option



Authors are linked to books through the AuthorId foreign key on the Books table. The SQL for creating the foreign key constraint is as follows:

1. CONSTRAINT [FK\_Books\_Authors\_AuthorId] FOREIGN KEY ([AuthorId]) REFERENCES [Authors] ([AuthorId])

The ON DELETE or ON UPDATE clause is usually appended to this.

#### **On Delete/Update No Action**

This is the default option if ON DELETE or ON UPDATE is not specified. If an attempt is made to delete an Author record that is currently referenced by a Book record, the database will raise an error and the delete will be rolled back. The same will happen if an attempt is made to alter the primary key value of an Author record that is being used as a foreign key in a Book record.

#### **On Delete/Update Restrict**

For the majority of relational database systems, RESTRICT is equivalent to NO ACTION. Where RESTRICT is specifically implemented by a database system, the difference is that RESTRICT results in the constraint being checked at the start of processing instead of at the end, which is the case with NO ACTION

#### **On Delete/Update Set Null**

If an Author record that is currently referenced in one or more Book records is deleted, the AuthorId value in the affected Book records will be updated to NULL. The same will happen if the Author record's primary key value is changed. This operation will only be successful if the AuthorId column in the Books table is nullable.

**Note** This behaviour is not enabled by default. You can use the Fluent API to opt into it by configuration.

#### **On Delete/Update Cascade**

Under this option, when an Author is deleted, the operation will also cascade to dependent tables, resulting in all related books being deleted. When you update the primary key value for an author, all book records containing that value as a foreign key will be updated to the new foreign key value.

#### **On Delete/Update Set Default**

When an Author record is deleted or has its key value amended, foreign key values in the Books table linked to the affected Author record will be changed to their default value. The foreign key fields in the affected tables must either have a default value specified, or be nullable, in which case NULL is considered the default value. Any non-NULL default value must relate to an existing key in the principal table.

# Configuration In Entity Framework Core

Configuration enables you to override Entity Framework Core's default behaviour ([**conventions**](https://www.learnentityframeworkcore.com/conventions)) in respect of mapping entities and their properties and relationships to a relational database. Configuration can be applied in two ways; through decorating classes and properties with [**attributes**](https://www.learnentityframeworkcore.com/configuration/data-annotation-attributes), or by using the [**Fluent API**](https://www.learnentityframeworkcore.com/configuration/fluent-api).

## **Attributes**

Attributes are used to add metadata to classes, properties, methods etc in the .NET Framework. They are applied to the target class or property by placing them in square brackets (angle brackets if using VB) just before the class or property:

1. [MyClassAttribute]
2. public class MyClass
3. {
4. [MyPropertyAttribute]
5. public int MyProperty { get; set; }
7. ...
8. }

You can apply multiple attributes in one of two ways - separately:

1. [MyFirstAttribute]
2. [MySecondAttribute]
3. [MyThirdAttribute]
4. public int MyProperty { get; set; }
5. ...

or as a comma-separated list:

1. [MyFirstAttribute, MySecondAttribute, MyThirdAttribute]
2. public int MyProperty { get; set; }
3. ...

The attributes used by EF Core reside in the System.ComponentModel.DataAnnotations namespace and are often referred to as Data Annotations.

#### **Pros**

* Attributes are applied directly to the domain model, so it is easy to see how the model is configured just by examining the class files.
* Some attributes, such as Required and StringLength are leveraged by client frameworks such as ASP.NET MVC to provide UI-based validation based on the specified configuration.

#### **Cons**

Data Annotation attributes can only provide a subset of configuration options. It may also be necessary to use the Fluent API to provide some configuration which will result in configuration being made in more than one place.

## **Fluent API**

Configuration is applied to entities or their properties via chained methods, which is the identifiying feature of the Fluent API pattern.

#### **Pros**

* The Fluent API provides greater scope for configuration than attributes.
* Configuration can be located in one place, away from the model classes.

#### **Cons**

Really, there aren't any.

#### **Further Reading**

* [**Data Annotation Attributes**](https://www.learnentityframeworkcore.com/configuration/data-annotation-attributes)
* [**Fluent API**](https://www.learnentityframeworkcore.com/configuration/fluent-api)

# Data Annotations Attributes

Configuration enables you to override EF Core's default behaviour. Configuration can be applied in two ways, using the [**Fluent API**](https://www.learnentityframeworkcore.com/configuration/fluent-api), and through DataAnnotation attributes.

Attributes are a kind of tag that you can place on a class or property to specify metadata about that class or property. Entity Framework Core makes use of attributes defined in the System.ComponentModel.DataAnnotations.Schema and System.ComponentModel.DataAnnotations namespaces.

#### **System.ComponentModel.DataAnnotations.Schema attributes**

| **Attribute** | **Description** |
| --- | --- |
| [**Table**](https://www.learnentityframeworkcore.com/configuration/data-annotation-attributes/table-attribute) | The database table and/or schema that a class is mapped to. |
| [**Column**](https://www.learnentityframeworkcore.com/configuration/data-annotation-attributes/column-attribute) | The database column that a property is mapped to. |
| [**ForeignKey**](https://www.learnentityframeworkcore.com/configuration/data-annotation-attributes/foreignkey-attribute) | Specifies the property is used as a foreign key in a relationship. |
| [**DatabaseGenerated**](https://www.learnentityframeworkcore.com/configuration/data-annotation-attributes/databasegenerated-attribute) | Specifies how the database generates values for a property. |
| [**NotMapped**](https://www.learnentityframeworkcore.com/configuration/data-annotation-attributes/notmapped-attribute) | Applied to properties or classes that are to be excluded from database mapping. |
| [**InverseProperty**](https://www.learnentityframeworkcore.com/configuration/data-annotation-attributes/inverseproperty-attribute) | Specifies the inverse of a navigation property |
| [**ComplexType**](https://www.learnentityframeworkcore.com/configuration/data-annotation-attributes/complextype-attribute) | Denotes that the class is a complex type. \*Not currently implemented in EF Core. |

#### **System.ComponentModel.Annotations attributes**

| **Attribute** | **Description** |
| --- | --- |
| [**Key**](https://www.learnentityframeworkcore.com/configuration/data-annotation-attributes/key-attribute) | Identifies one or more properties as a Key |
| [**Timestamp**](https://www.learnentityframeworkcore.com/configuration/data-annotation-attributes/timestamp-attribute) | Specifies the data type of the database column as rowversion |
| [**ConcurrencyCheck**](https://www.learnentityframeworkcore.com/configuration/data-annotation-attributes/concurrencycheck-attribute) | Specifies that the property is included in concurrency checks |
| [**Required**](https://www.learnentityframeworkcore.com/configuration/data-annotation-attributes/required-attribute) | Specifies that the property's value is required |
| [**MaxLength**](https://www.learnentityframeworkcore.com/configuration/data-annotation-attributes/maxlength-attribute) | Sets the maximum allowed length of the property value (string or array) |
| [**StringLength**](https://www.learnentityframeworkcore.com/configuration/data-annotation-attributes/stringlength-attribute) | Sets the maximum allowed length of the property value (string or array) |

Last updated: 7/18/2021 2:46:38 AM

# Fluent API Configuration

EF Core's Fluent API provides methods for configuring various aspects of your model:

* [**Model-wide configuration**](https://www.learnentityframeworkcore.com/configuration/fluent-api/model-configuration)
* [**Type configuration**](https://www.learnentityframeworkcore.com/configuration/fluent-api/type-configuration)
* [**Property configuration**](https://www.learnentityframeworkcore.com/configuration/fluent-api/property-configuration)

Configurations are applied via a number of methods exposed by the Microsoft.EntityFrameworkCore.ModelBuilder class. The DbContext class has a method called OnModelCreating that takes an instance of ModelBuilder as a parameter. This method is called by the framework when your context is first created to build the model and its mappings in memory. You can override this method to add your own configurations:

1. public class SampleContext : DbContext
2. {
3. // Specify DbSet properties etc
4. protected override void OnModelCreating(ModelBuilder modelBuilder)
5. {
6. // add your own configuration here
7. }
8. }

The term Fluent API refers to a pattern of programming where method calls are chained together with the end result being certainly less verbose and arguably more readable than a series of statements:

1. // series of statements
2. modelBuilder.Entity<Order>().Property(t => t.OrderDate).IsRequired();
3. modelBuilder.Entity<Order>().Property(t => t.OrderDate).HasColumnType("Date");
4. modelBuilder.Entity<Order>().Property(t => t.OrderDate).HasDefaultValueSql("GetDate()");
5. // fluent api chained calls
6. modelBuilder.Entity<Order>()
7. .Property(t => t.OrderDate)
8. .IsRequired()
9. .HasColumnType("Date")
10. .HasDefaultValueSql("GetDate()");

In this example, the DateCreated property of the Order entity is configured as Required (not nullable), is mapped to a SQL Server Date type and has a default value applied. Each method call returns a type that exposes its own set of methods which is what enables the chaining of methods. Often, the type returned from the method call is the same instance as the one on which the method is called. For example, the IsRequired method above is exposed by the PropertyBuilder class. The return type of this method is an instance of the PropertyBuilder class, so once the method has performed its task, it ends simply with return this;.

The Fluent API provides a larger range of configuration options than Data Annotation attributes. Fluent API configuration also facilitates cleaner code, in that the configuration can be kept separate from the domain classes.

## **Separate Configuration Classes**

Most examples in this guide show configurations being applied in the OnModelCreating method, but it is recommended to separate configurations out to individual files per entity - especially for larger models or ones that require a lot of configuration.

In pre-.NET Core versions of Entity Framework, this is achieved by creating classes that derive from EntityTypeConfiguration<TEntity>, and then using Fluent API to override conventions in the class constructor. These classes are then added to the DbModelBuilder's configuration in the OnModelCreating method. Prior to version 2.0 of Entity Framework Core, there was no equivalent to this approach and one had to roll one's own solution.

In version 2.0, an new interface was introduced: IEntityTypeConfiguration<TEntity>. This is used in a similar way to EntityTypeConfiguration<TEntity> in that configurations are specified in separate entity-specific classes:

1. public class OrderConfiguration : IEntityTypeConfiguration<Order>
2. {
3. public void Configure(EntityTypeBuilder<Order> builder)
4. {
5. builder.HasKey(o => o.OrderNumber);
6. builder.Property(t => t.OrderDate)
7. .IsRequired()
8. .HasColumnType("Date")
9. .HasDefaultValueSql("GetDate())"
10. }
11. }

This configuration is applied in the OnModelCreating method as follows:

1. protected override void OnModelCreating(ModelBuilder modelBuilder)
2. {
3. modelBuilder.ApplyConfiguration(new OrderConfiguration());
4. }

Note that while this approach follows a similar pattern to Entity Framework 6, there are some differences:

* The configuration class must implement IEntityTypeConfiguration<TEntity> (an interface) rather than EntityTypeConfiguration<TEntity> (a class).
* The configuration is applied in a Configure method rather than in a constructor.
* Configurations are added to the ModelBuilder using an ApplyConfiguration method instead of being added to a Configurations collection.

In the example above, only one entity type configuration was registered. Larger applications will require multiple type configurations, and as the scope of the application grows, the developer will have to remember to register all new type configurations. A new extension method, ApplyConfigurationsFromAssembly, was introduced in 2.2, which scans a given assembly for all types that implement IEntityTypeConfiguration, and registers each one automatically. It is used like this:

1. protected override void OnModelCreating(ModelBuilder modelBuilder)
2. {
3. modelBuilder.ApplyConfigurationsFromAssembly(Assembly.GetExecutingAssembly());
4. }

Once this line of code has been added, you no longer need to remember to add new type configuration registrations to the OnModelCreating method as your model grows.

# Configuring One To Many Relationships in Entity Framework Core

Most one-to-many relationships in an Entity Framework Core model [**follow conventions**](https://www.learnentityframeworkcore.com/conventions/one-to-many-relationship) and require no additional configuration. Where the model does not follow convention, the Fluent API can be used to configure the correct relationship between entities.

## **Has/With pattern**

When configuring relationships with the Fluent API, you will use the [**Has/With pattern**](https://www.learnentityframeworkcore.com/configuration/fluent-api/haswith-pattern). The Has side of the pattern is represented by the HasOne and HasMany methods. The With side of the relationship is represented by the WithOne and WithMany methods.

The following model represents companies and employees with an [**inverse navigation property defined**](https://www.learnentityframeworkcore.com/conventions/one-to-many-relationship#inverse-navigation-property) in the dependent entity (Employee) but no matching foreign key property in the dependent:

1. public class Company
2. {
3. public int Id { get; set; }
4. public string Name { get; set; }
5. public ICollection<Employee> Employees { get; set; }
6. }
7. public class Employee
8. {
9. public int Id { get; set; }
10. public string Name { get; set; }
11. public Company Company { get; set; }
12. }

A company has many employees, each with one company. That relationship is represented as follows:

1. protected override void OnModelCreating(Modelbuilder modelBuilder)
2. {
3. modelBuilder.Entity<Company>()
4. .HasMany(c => c.Employees)
5. .WithOne(e => e.Company);
6. }

It can also be configured by starting with the other end of the relationship:

1. protected override void OnModelCreating(Modelbuilder modelBuilder)
2. {
3. modelBuilder.Entity<Employee>()
4. .HasOne(e => e.Company)
5. .WithMany(c => c.Employees);
6. }

Whether starting with the Company or Employee entity, this configuration will result in an optional relationship. A foreign key [**shadow property**](https://www.learnentityframeworkcore.com/model/shadow-properties) named CompanyId will be introduced by EF Core to the Employee entity which will be nullable. In addition, the [**referential action constraint**](https://www.learnentityframeworkcore.com/relationships/referential-constraint-action-options) on the relationship will be set to NoAction.

## **Required relationship**

You can use the IsRequired method on the relationship to prevent the relationship being optional:

1. protected override void OnModelCreating(Modelbuilder modelBuilder)
2. {
3. modelBuilder.Entity<Company>()
4. .HasMany(c => c.Employees)
5. .WithOne(e => e.Company).
6. .IsRequired();
7. }

The [**referential action constraint**](https://www.learnentityframeworkcore.com/relationships/referential-constraint-action-options) on the relationship will be set to Cascade.

## **Cascading Referential Integrity Constraints**

If the relationship is configured as optional, the default behavour of EF Core is to take no action in respect of the dependent entity if the principal is deleted. Consequently, if you delete the principal, any dependents will be left with foreign key values that reference principals that no longer exist. The default behaviour of a database is to raise an error in this scenario: foreign key values must reference existing primary key values.

You can alter this behaviour through the OnDelete method which takes a DeleteBehaviour enumeration. The following example sets the foreign key value of the dependent entity to null in the event that the principal is deleted:

1. protected override void OnModelCreating(Modelbuilder modelBuilder)
2. {
3. modelBuilder.Entity<Company>()
4. .HasMany(c => c.Employees)
5. .WithOne(e => e.Company).
6. .OnDelete(DeleteBehavior.SetNull);
7. }

This example will result in the dependent entity being deleted:

1. protected override void OnModelCreating(Modelbuilder modelBuilder)
2. {
3. modelBuilder.Entity<Company>()
4. .HasMany(c => c.Employees)
5. .WithOne(e => e.Company).
6. .OnDelete(DeleteBehavior.Delete);
7. }

#### **Further Reading**

* [**Fluent API OnDelete method**](https://www.learnentityframeworkcore.com/configuration/fluent-api/ondelete-method)
* [**Deleting Data**](https://www.learnentityframeworkcore.com/dbcontext/deleting-data)