**Entity Framework Core Fundamentals**

EF Core can serve as an object-relational mapper (O/RM), enabling .NET developers to work with a database using .NET objects, and eliminating the need for most of the data-access code they usually need to write.

**The Model**

With EF Core, data access is performed using a model. A model is made up of entity classes and a context object that represents a session with the database, allowing you to query and save data.

You can generate a model from an existing database, hand code a model to match your database, or use [EF Migrations](https://docs.microsoft.com/en-us/ef/core/managing-schemas/migrations/index) to create a database from your model, and then evolve it as your model changes over time.

# Connection Strings

Most database providers require some form of connection string to connect to the database. The connection string should be added to your application's App.config/Web.config. If your connection string contains sensitive information, such as username and password, you can protect the contents of the configuration file using the [Secret Manager tool](https://docs.microsoft.com/aspnet/core/security/app-secrets#secret-manager)

<add name="BloggingDatabase" connectionString="Server=(localdb)\mssqllocaldb;Database=Blogging;Trusted\_Connection=True; MultipleActiveResultSets=True " />

The providerName setting is not required on EF Core connection strings stored in App.config because the database provider is configured via code.

//Trusted\_Connection=True, Integrated Security =SSPI, Integrated Security = True all these three property hold same meaning and state that whether or not use windows integrated / windows credentials to log into the database

//If windows credentials are out of scope, then database user name and password must be used explicitly defined in the connection string.

//If connection string include both user name & password and Integrated (Trusted\_Connection=True or Integrated Security =SSPI Or Integrated Security = True ) then windows credential will be used and user name & pwd combination is ignored.

//MultipleActiveResultSets=True -> Before the introduction of Multiple Active Result Sets (MARS), developers had to use either multiple connections or server-side cursors to solve certain scenarios ( for eg:

// read record from an ExecuteReder (where connection will be open until closed), and then for each record, you further query database to do some other action. In this scenario, we need to maintain

//multiple connection at same time.

You can then read the connection string using the ConfigurationManager API in your context's OnConfiguring method.

## ASP.NET Core

In ASP.NET Core the configuration system is very flexible, and the connection string could be stored in appsettings.json, an environment variable, the user secret store, or another configuration source.

The context is typically configured in Startup.cs with the connection string being read from configuration.

public void ConfigureServices(IServiceCollection services)

{

services.AddDbContext<BloggingContext>(options =>

options.UseSqlServer(Configuration.GetConnectionString("BloggingDatabase")));

}

# **Logging**

## ASP.NET Core applications

EF Core integrates automatically with the logging mechanisms of ASP.NET Core whenever AddDbContext or AddDbContextPool is used.

EF Core logging requires an ILoggerFactory which is itself configured with one or more logging providers. Common providers are shipped in the following packages:

* [Microsoft.Extensions.Logging.Console](https://www.nuget.org/packages/Microsoft.Extensions.Logging.Console/): A simple console logger.
* [Microsoft.Extensions.Logging.AzureAppServices](https://www.nuget.org/packages/Microsoft.Extensions.Logging.AzureAppServices/): Supports Azure App Services 'Diagnostics logs' and 'Log stream' features.
* [Microsoft.Extensions.Logging.Debug](https://www.nuget.org/packages/Microsoft.Extensions.Logging.Debug/): Logs to a debugger monitor using System.Diagnostics.Debug.WriteLine().
* [Microsoft.Extensions.Logging.EventLog](https://www.nuget.org/packages/Microsoft.Extensions.Logging.EventLog/): Logs to Windows Event Log.
* [Microsoft.Extensions.Logging.EventSource](https://www.nuget.org/packages/Microsoft.Extensions.Logging.EventSource/): Supports EventSource/EventListener.
* [Microsoft.Extensions.Logging.TraceSource](https://www.nuget.org/packages/Microsoft.Extensions.Logging.TraceSource/): Logs to a trace listener using System.Diagnostics.TraceSource.TraceEvent().

After installing the appropriate package(s), the application should create a singleton/global instance of a LoggerFactory. For example, using the console logger:

# **Connection Resiliency**

Connection resiliency automatically retries failed database commands. The feature can be used with any database by supplying an "execution strategy", which encapsulates the logic necessary to detect failures and retry commands. EF Core providers can supply execution strategies tailored to their specific database failure conditions and optimal retry policies.

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

optionsBuilder

.UseSqlServer(

@"Server=(localdb)\mssqllocaldb;Database=EFMiscellanous.ConnectionResiliency;Trusted\_Connection=True;ConnectRetryCount=0",

options => options.EnableRetryOnFailure());

}

 in Startup.cs for an ASP.NET Core application:

public void ConfigureServices(IServiceCollection services)

{

services.AddDbContext<PicnicContext>(

options => options.UseSqlServer(

"<connection string>",

providerOptions => providerOptions.EnableRetryOnFailure()));

}

## Custom execution strategy

There is a mechanism to register a custom execution strategy of your own if you wish to change any of the defaults.

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

optionsBuilder

.UseMyProvider(

"<connection string>",

options => options.ExecutionStrategy(...));

}

## Execution strategies and transactions

An execution strategy that automatically retries on failures will needs to play back each operation in a retry block that fails. When retries are enabled, each operation you perform via EF Core becomes its own retriable operation. That is, each query and each call to SaveChanges() will be retried as a unit if a transient failure occurs.

However, if your code initiates a transaction using BeginTransaction() you are defining your own group of operations that need to be treated as a unit, and everything inside the transaction would need to be played back upon failure. You will receive an exception like the following if you attempt to do this when using an execution strategy:

*InvalidOperationException: The configured execution strategy 'SqlServerRetryingExecutionStrategy' does not support user initiated transactions. Use the execution strategy returned by 'DbContext.Database.CreateExecutionStrategy()' to execute all the operations in the transaction as a retriable unit.*

The solution is to manually invoke the execution strategy with a delegate representing everything that needs to be executed. If a transient failure occurs, the execution strategy will invoke the delegate again.

using (var db = new BloggingContext())

{

var strategy = db.Database.CreateExecutionStrategy();

strategy.Execute(() =>

{

using (var context = new BloggingContext())

{

using (var transaction = context.Database.BeginTransaction())

{

context.Blogs.Add(new Blog {Url = "http://blogs.msdn.com/dotnet"});

context.SaveChanges();

context.Blogs.Add(new Blog {Url = "http://blogs.msdn.com/visualstudio"});

context.SaveChanges();

transaction.Commit();

}

}

});

}

This approach can also be used with ambient transactions.

using (var context1 = new BloggingContext())

{

context1.Blogs.Add(new Blog { Url = "http://blogs.msdn.com/visualstudio" });

var strategy = context1.Database.CreateExecutionStrategy();

strategy.Execute(() =>

{

using (var context2 = new BloggingContext())

{

using (var transaction = new TransactionScope())

{

context2.Blogs.Add(new Blog { Url = "http://blogs.msdn.com/dotnet" });

context2.SaveChanges();

context1.SaveChanges();

transaction.Complete();

}

}

});

}

## Transaction commit failure and the idempotency issue

In general, when there is a connection failure the current transaction is rolled back. However, if the connection is dropped while the transaction is being committed the resulting state of the transaction is unknown.

By default, the execution strategy will retry the operation as if the transaction was rolled back. But if it's not the case this will result in an exception if the new database state is incompatible or could lead to **data corruption** if the operation does not rely on a particular state, for example when inserting a new row with auto-generated key values.

To deal with above:

### Option 1 - Do (almost) nothing

The likelihood of a connection failure during transaction commit is low so it may be acceptable for your application to just fail if this condition actually occurs.

However, you need to avoid using store (backend)-generated keys in order to ensure that an exception is thrown instead of adding a duplicate row.

Consider using a client-generated GUID value or a client-side value generator.

### Option 2 - Rebuild application state

1. Discard the current DbContext.
2. Create a new DbContext and restore the state of your application from the database.
3. Inform the user that the last operation might not have been completed successfully.

### Option 3 - Add state verification

For most of the operations that change the database state it is possible to add code that checks whether it succeeded. EF provides an extension method to make this easier - IExecutionStrategy.ExecuteInTransaction.

This method begins and commits a transaction and also accepts a function in the verifySucceeded parameter that is invoked when a transient error occurs during the transaction commit.

using (var db = new BloggingContext())

{

var strategy = db.Database.CreateExecutionStrategy();

var blogToAdd = new Blog {Url = "http://blogs.msdn.com/dotnet"};

db.Blogs.Add(blogToAdd);

strategy.ExecuteInTransaction(db,

operation: context =>

{

context.SaveChanges(acceptAllChangesOnSuccess: false);

},

verifySucceeded: context => context.Blogs.AsNoTracking().Any(b => b.BlogId == blogToAdd.BlogId));

db.ChangeTracker.AcceptAllChanges();

}

Here SaveChanges is invoked with acceptAllChangesOnSuccess set to false to avoid changing the state of the Blog entity to Unchanged if SaveChanges succeeds. This allows to retry the same operation if the commit fails and the transaction is rolled back.

### Option 4 - Manually track the transaction

If you need to use store-generated keys or need a generic way of handling commit failures that doesn't depend on the operation performed each transaction could be assigned an ID that is checked when the commit fails.

1. Add a table to the database used to track the status of the transactions.
2. Insert a row into the table at the beginning of each transaction.
3. If the connection fails during the commit, check for the presence of the corresponding row in the database.
4. If the commit is successful, delete the corresponding row to avoid the growth of the table.

using (var db = new BloggingContext())

{

var strategy = db.Database.CreateExecutionStrategy();

db.Blogs.Add(new Blog { Url = "http://blogs.msdn.com/dotnet" });

var transaction = new TransactionRow {Id = Guid.NewGuid()};

db.Transactions.Add(transaction);

strategy.ExecuteInTransaction(db,

operation: context =>

{

context.SaveChanges(acceptAllChangesOnSuccess: false);

},

verifySucceeded: context => context.Transactions.AsNoTracking().Any(t => t.Id == transaction.Id));

db.ChangeTracker.AcceptAllChanges();

db.Transactions.Remove(transaction);

db.SaveChanges();

}

Make sure that the context used for the verification has an execution strategy defined as the connection is likely to fail again during verification if it failed during transaction commit.

**Testing components using EF Core**

You may want to test components using something that approximates connecting to the real database, without the overhead of actual database I/O operations.

There are two main options for doing this:

* [SQLite in-memory mode](https://docs.microsoft.com/en-us/ef/core/miscellaneous/testing/sqlite) allows you to write efficient tests against a provider that behaves like a relational database.
* [The InMemory provider](https://docs.microsoft.com/en-us/ef/core/miscellaneous/testing/in-memory) is a lightweight provider that has minimal dependencies, but does not always behave like a relational database.

SQLite has an in-memory mode that allows you to use SQLite to write tests against a relational database, without the overhead of actual database operations.

## Example testing scenario

Consider the following service that allows application code to perform some operations related to blogs. Internally it uses a DbContext that connects to a SQL Server database.

It would be useful to swap this context to connect to an in-memory SQLite database so that we can write efficient tests for this service without having to modify the code, or do a lot of work to create a test double of the context.

using System.Collections.Generic;

using System.Linq;

namespace BusinessLogic

{

public class BlogService

{

private BloggingContext \_context;

public BlogService(BloggingContext context)

{

\_context = context;

}

public void Add(string url)

{

var blog = new Blog { Url = url };

\_context.Blogs.Add(blog);

\_context.SaveChanges();

}

public IEnumerable<Blog> Find(string term)

{

return \_context.Blogs

.Where(b => b.Url.Contains(term))

.OrderBy(b => b.Url)

.ToList();

}

}

}

## Get your context ready

### **Avoid configuring two database providers**

In your tests you are going to externally configure the context to use the InMemory provider. If you are configuring a database provider by overriding **OnConfiguring** in your context, then you need to add some conditional code to ensure that you only configure the database provider if one has not already been configured.

If you are using ASP.NET Core, then you should not need this code since your database provider is configured outside of the context (in Startup.cs).

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

if (!optionsBuilder.IsConfigured)

{

optionsBuilder.UseSqlServer(@"Server=(localdb)\mssqllocaldb;Database=EFProviders.InMemory;Trusted\_Connection=True;ConnectRetryCount=0");

}

}

### **Add a constructor for testing**

The simplest way to enable testing against a different database is to modify your context to expose a constructor that accepts a **DbContextOptions<TContext>.**

public class BloggingContext : DbContext

{

public BloggingContext()

{ }

public BloggingContext(DbContextOptions<BloggingContext> options)

: base(options)

{ }

DbContextOptions<TContext> tells the context all of its settings, such as which database to connect to. This is the same object that is built by running the OnConfiguring method in your context.

## Writing tests

The key to testing with this provider is the ability to tell the context to use SQLite, and control the scope of the in-memory database. The scope of the database is controlled by opening and closing the connection. The database is scoped to the duration that the connection is open. Typically you want a clean database for each test method.

using BusinessLogic;

using Microsoft.Data.Sqlite;

using Microsoft.EntityFrameworkCore;

using System.Linq;

using Xunit;

namespace EFTesting.TestProject.SQLite

{

public class BlogServiceTests

{

[Fact]

public void Add\_writes\_to\_database()

{

// In-memory database only exists while the connection is open

var connection = new SqliteConnection("DataSource=:memory:");

connection.Open();

try

{

var options = new DbContextOptionsBuilder<BloggingContext>()

.UseSqlite(connection)

.Options;

// Create the schema in the database

using (var context = new BloggingContext(options))

{

context.Database.EnsureCreated();

}

// Run the test against one instance of the context

using (var context = new BloggingContext(options))

{

var service = new BlogService(context);

service.Add("https://example.com");

context.SaveChanges();

}

// Use a separate instance of the context to verify correct data was saved to database

using (var context = new BloggingContext(options))

{

Assert.Equal(1, context.Blogs.Count());

Assert.Equal("https://example.com", context.Blogs.Single().Url);

}

}

finally

{

connection.Close();

}

}

[Fact]

public void Find\_searches\_url()

{

// In-memory database only exists while the connection is open

var connection = new SqliteConnection("DataSource=:memory:");

connection.Open();

try

{

var options = new DbContextOptionsBuilder<BloggingContext>()

.UseSqlite(connection)

.Options;

// Create the schema in the database

using (var context = new BloggingContext(options))

{

context.Database.EnsureCreated();

}

// Insert seed data into the database using one instance of the context

using (var context = new BloggingContext(options))

{

context.Blogs.Add(new Blog { Url = "https://example.com/cats" });

context.Blogs.Add(new Blog { Url = "https://example.com/catfish" });

context.Blogs.Add(new Blog { Url = "https://example.com/dogs" });

context.SaveChanges();

}

// Use a clean instance of the context to run the test

using (var context = new BloggingContext(options))

{

var service = new BlogService(context);

var result = service.Find("cat");

Assert.Equal(2, result.Count());

}

}

finally

{

connection.Close();

}

}

}

}

# **Configuring a DbContext**

This article shows basic patterns for configuring a DbContext via a DbContextOptions to connect to a database using a specific EF Core provider and optional behaviors.

EF Core design-time tools such as [migrations](https://docs.microsoft.com/en-us/ef/core/managing-schemas/migrations/index) need to be able to discover and create a working instance of a DbContext type in order to gather details about the application's entity types and how they map to a database schema. This process can be automatic as long as the tool can easily create the DbContext in such a way that it will be configured similarly to how it would be configured at run-time.

While any pattern that provides the necessary configuration information to the DbContext can work at run-time, tools that require using a DbContext at design-time can only work with a limited number of patterns.

**Configuring DbContextOptions**

DbContext must have an instance of DbContextOptions in order to perform any work. The DbContextOptions instance carries configuration information such as:

* The database provider to use, typically selected by invoking a method such as UseSqlServer or UseSqlite.
* Any necessary connection string or identifier of the database instance
* Any provider-level optional behavior selectors, typically also chained inside the call to the provider selection method
* Any general EF Core behavior selectors, typically chained after or before the provider selector method

The following example configures the DbContextOptions to use the SQL Server provider, a connection contained in the connectionString variable, a provider-level command timeout, and an EF Core behavior selector that makes all queries executed in the DbContext [no-tracking](https://docs.microsoft.com/en-us/ef/core/querying/tracking#no-tracking-queries) by default:

optionsBuilder

.UseSqlServer(connectionString, providerOptions=>providerOptions.CommandTimeout(60))

.UseQueryTrackingBehavior(QueryTrackingBehavior.NoTracking);

Provider selector methods and other behavior selector methods mentioned above are extension methods on DbContextOptions or provider-specific option classes. In order to have access to these extension methods you may need to have a namespace (typically Microsoft.EntityFrameworkCore) in scope and include additional package dependencies in the project.

The DbContextOptions can be supplied to the DbContext by overriding the OnConfiguring method or externally via a constructor argument.

If both are used, OnConfiguring is applied last and can overwrite options supplied to the constructor argument.

public class BloggingContext : DbContext

{

public BloggingContext(DbContextOptions<BloggingContext> options)

: base(options)

{ }

public DbSet<Blog> Blogs { get; set; }

}

Your application can now pass the DbContextOptions when instantiating a context, as follows:

var optionsBuilder = new DbContextOptionsBuilder<BloggingContext>();

optionsBuilder.UseSqlite("Data Source=blog.db");

using (var context = new BloggingContext(optionsBuilder.Options))

{

// do stuff

### }

### **OnConfiguring**

You can also initialize the DbContextOptions within the context itself. While you can use this technique for basic configuration, you will typically still need to get certain configuration details from the outside, e.g. a database connection string. This can be done with a configuration API or any other means.

public class BloggingContext : DbContext

{

public DbSet<Blog> Blogs { get; set; }

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

optionsBuilder.UseSqlite("Data Source=blog.db");

}

}

An application can simply instantiate such a context without passing anything to its constructor:

using (var context = new BloggingContext())

{

// do stuff

}

### **Using DbContext with dependency injection**

EF Core supports using DbContext with a dependency injection container. Your DbContext type can be added to the service container by using the AddDbContext<TContext> method.

AddDbContext<TContext> will make both your DbContext type, TContext, and the corresponding DbContextOptions<TContext> available for injection from the service container.

**Adding the DbContext to dependency injection:**

public void ConfigureServices(IServiceCollection services)

{

services.AddDbContext<BloggingContext>(options => options.UseSqlite("Data Source=blog.db"));

}

This requires adding a [constructor argument](https://docs.microsoft.com/en-us/ef/core/miscellaneous/configuring-dbcontext#constructor-argument) to your DbContext type that accepts DbContextOptions<TContext>.

**Application code (in ASP.NET Core):**

public class MyController

{

private readonly BloggingContext \_context;

public MyController(BloggingContext context)

{

\_context = context;

}

}

**Application code (using ServiceProvider directly, less common):**

using (var context = serviceProvider.GetService<BloggingContext>())

{

// do stuff

}

var options = serviceProvider.GetService<DbContextOptions<BloggingContext>>();

## Avoiding DbContext threading issues

Entity Framework Core **does not support multiple parallel operations** being run on **the same DbContext instance**. This includes both parallel execution of async queries and any explicit concurrent use from multiple threads.

Therefore, **always await async calls immediately**, or **use separate DbContext instances** for operations that execute in parallel.

When EF Core detects an attempt to use a DbContext instance concurrently, you'll see an InvalidOperationException with a message like this:

*A second operation started on this context before a previous operation completed. This is usually caused by different threads using the same instance of DbContext, however instance members are not guaranteed to be thread safe.*

*When concurrent access goes undetected, it can result in undefined behavior, application crashes and data corruption.*

There are common mistakes that can inadvertently cause concurrent access on the same DbContext instance:

### **Forgetting to await the completion of an asynchronous operation before starting any other operation on the same DbContext**

Asynchronous methods enable EF Core to initiate operations that access the database in a non-blocking way. But if a caller does not await the completion of one of these methods, and proceeds to perform other operations on the DbContext, the state of the DbContext can be, (and very likely will be) corrupted.

Always await EF Core asynchronous methods immediately.

### **Implicitly sharing DbContext instances across multiple threads via dependency injection**

The [AddDbContext](https://docs.microsoft.com/dotnet/api/microsoft.extensions.dependencyinjection.entityframeworkservicecollectionextensions.adddbcontext) extension method registers DbContext types with a [scoped lifetime](https://docs.microsoft.com/aspnet/core/fundamentals/dependency-injection#service-lifetimes) by default.

This is safe from concurrent access issues in ASP.NET Core applications because there is only one thread executing each client request at a given time, and because each request gets a separate dependency injection scope (and therefore a separate DbContext instance).

However any code that explicitly executes multiple threads in parallel should ensure that DbContext instances aren't ever accessed concurrently.

Using dependency injection, this can be achieved by either registering the context as scoped and creating scopes (using IServiceScopeFactory) for each thread, or by registering the DbContext as transient (using the overload of AddDbContext which takes a ServiceLifetime parameter).

## DbContext and DbSet

When nullable reference types are enabled, the C# compiler emits warnings for any uninitialized non-nullable property, as these would contain null. As a result, the common practice of defining a non-nullable DbSet on a context will now generate a warning. However, EF Core always initializes all DbSet properties on DbContext-derived types, so they are guaranteed to never be null, even if the compiler is unaware of this. Therefore, it is recommended to keep your DbSet properties non-nullable - allowing you to access them without null checks - and to silence the compiler warnings by explicitly setting them to null with the help of the null-forgiving operator (!):

public class NullableReferenceTypesContext : DbContext

{

public DbSet<Customer> Customers { get; set; } = null!; public DbSet<Order> Orders { get; set; } = null!;

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

=> optionsBuilder

.UseSqlServer(@"Server=(localdb)\mssqllocaldb;Database=EFNullableReferenceTypes;Trusted\_Connection=True;ConnectRetryCount=0");

}

## Non-nullable properties and initialization

Compiler warnings for uninitialized non-nullable reference types are also a problem for regular properties on your entity types.

Required navigation properties present an additional difficulty: although a dependent will always exist for a given principal, it may or may not be loaded by a particular query, depending on the needs at that point in the program.

One way to deal with these scenarios, is to have a non-nullable property with a nullable [backing field](https://docs.microsoft.com/en-us/ef/core/modeling/backing-field):

private Address? \_shippingAddress;

public Address ShippingAddress

{

set => \_shippingAddress = value;

get => \_shippingAddress

?? throw new InvalidOperationException("Uninitialized property: " + nameof(ShippingAddress));

}

Since the navigation property is non-nullable, a required navigation is configured; and as long as the navigation is properly loaded, the dependent will be accessible via the property. If, however, the property is accessed without first properly loading the related entity, an InvalidOperationException is thrown, since the API contract has been used incorrectly.

Note that EF must be configured to always access the backing field and not the property, as it relies on being able to read the value even when unset; consult the documentation on [backing fields](https://docs.microsoft.com/en-us/ef/core/modeling/backing-field) on how to do this

As a terser alternative, it is possible to simply initialize the property to null with the help of the null-forgiving operator (!):

public Product Product { get; set; } = null!;

Collection navigations, which contain references to multiple related entities, should always be non-nullable. An empty collection means that no related entities exist, but the list itself should never be null.

## Navigating and including nullable relationships

When dealing with optional relationships, it's possible to encounter compiler warnings where an actual null reference exception would be impossible. When translating and executing your LINQ queries, EF Core guarantees that if an optional related entity does not exist, any navigation to it will simply be ignored, rather than throwing. However, the compiler is unaware of this EF Core guarantee, and produces warnings as if the LINQ query were executed in memory, with LINQ to Objects. As a result, it is necessary to use the null-forgiving operator (!) to inform the compiler that an actual null value isn't possible:

Console.WriteLine(order.OptionalInfo!.ExtraAdditionalInfo!.SomeExtraAdditionalInfo);

A similar issue occurs when including multiple levels of relationships across optional navigations:

var order = context.Orders

.Include(o => o.OptionalInfo!)

.ThenInclude(op => op.ExtraAdditionalInfo)

.Single();

If you find yourself doing this a lot, and the entity types in question are predominantly (or exclusively) used in EF Core queries, consider making the navigation properties non-nullable, and to configure them as optional via the Fluent API or Data Annotations. This will remove all compiler warnings while keeping the relationship optional; however, if your entities are traversed outside of EF Core, you may observe null values although the properties are annotated as non-nullable.