**Entity Framework Core**

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");

}

}