# Using Transactions

Article • 01/12/2023 • 6 minutes to read

Transactions allow several database operations to be processed in an atomic manner. If the transaction is committed, all of the operations are successfully applied to the database. If the transaction is rolled back, none of the operations are applied to the database.

**Tip**

You can view this article's **sample** on GitHub.

## Default transaction behavior

By default, if the database provider supports transactions, all changes in a single call to `SaveChanges` are applied in a transaction. If any of the changes fail, then the transaction is rolled back and none of the changes are applied to the database. This means that `SaveChanges` is guaranteed to either completely succeed, or leave the database unmodified if an error occurs.

For most applications, this default behavior is sufficient. You should only manually control transactions if your application requirements deem it necessary.

## Controlling transactions

You can use the `DbContext.Database` API to begin, commit, and rollback transactions. The following example shows two `SaveChanges` operations and a LINQ query being executed in a single transaction:

C#

```csharp
using var context = new BloggingContext();
using var transaction = context.Database.BeginTransaction();

try
{
    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();

    var blogs = context.Blogs
        .OrderBy(b => b.Url)
        .ToList();

    // Commit transaction if all commands succeed, transaction will auto-
rollback
    // when disposed if either commands fails
    transaction.Commit();
}
catch (Exception)
{
    // TODO: Handle failure
}
```

While all relational database providers support transactions, other providers types may throw or no-op when transaction APIs are called.

ⓘ **Note**

Manually controlling transactions in this way is incompatible with implicitly invoked retrying execution strategies. See **Connection Resiliency** for more information.

## Savepoints

When `SaveChanges` is invoked and a transaction is already in progress on the context, EF automatically creates a *savepoint* before saving any data. Savepoints are points within a database transaction which may later be rolled back to, if an error occurs or for any other reason. If `SaveChanges` encounters any error, it automatically rolls the transaction back to the savepoint, leaving the transaction in the same state as if it had never started. This allows you to possibly correct issues and retry saving, in particular when optimistic concurrency issues occur.

⚠ **Warning**

Savepoints are incompatible with SQL Server's Multiple Active Result Sets, and are not used. If an error occurs during `SaveChanges`, the transaction may be left in an unknown state.

It's also possible to manually manage savepoints, just as it is with transactions. The following example creates a savepoint within a transaction, and rolls back to it on failure:

C#

```
using var context = new BloggingContext();
using var transaction = context.Database.BeginTransaction();

try
{
    context.Blogs.Add(new Blog { Url = "https://devblogs.microsoft.com/dot-
net/" });
    context.SaveChanges();

    transaction.CreateSavepoint("BeforeMoreBlogs");

    context.Blogs.Add(new Blog { Url = "https://devblogs.microsoft.com/visu-
alstudio/" });
    context.Blogs.Add(new Blog { Url = "https://devblogs.microsoft.com/asp-
net/" });
    context.SaveChanges();

    transaction.Commit();
}
catch (Exception)
{
    // If a failure occurred, we rollback to the savepoint and can continue
the transaction
    transaction.RollbackToSavepoint("BeforeMoreBlogs");

    // TODO: Handle failure, possibly retry inserting blogs
}
```

## Cross-context transaction

You can also share a transaction across multiple context instances. This functionality is only available when using a relational database provider because it requires the use of `DbTransaction` and `DbConnection`, which are specific to relational databases.

To share a transaction, the contexts must share both a `DbConnection` and a

`DbTransaction`.

## Allow connection to be externally provided

Sharing a `DbConnection` requires the ability to pass a connection into a context when constructing it.

The easiest way to allow `DbConnection` to be externally provided, is to stop using the `DbContext.OnConfiguring` method to configure the context and externally create `DbContextOptions` and pass them to the context constructor.

**Tip**

`DbContextOptionsBuilder` is the API you used in `DbContext.OnConfiguring` to configure the context, you are now going to use it externally to create `DbContextOptions`.

C#

```csharp
public class BloggingContext : DbContext
{
    public BloggingContext(DbContextOptions<BloggingContext> options)
        : base(options)
    {
    }

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

An alternative is to keep using `DbContext.OnConfiguring`, but accept a `DbConnection` that is saved and then used in `DbContext.OnConfiguring`.

C#

```csharp
public class BloggingContext : DbContext
{
    private DbConnection _connection;

    public BloggingContext(DbConnection connection)
    {
      _connection = connection;
    }

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

    protected override void OnConfiguring(DbContextOptionsBuilder
optionsBuilder)
    {
        optionsBuilder.UseSqlServer(_connection);
    }
}
```

## Share connection and transaction

You can now create multiple context instances that share the same connection. Then use the `DbContext.Database.UseTransaction(DbTransaction)` API to enlist both contexts in the same transaction.

C#

```
using var connection = new SqlConnection(connectionString);
var options = new DbContextOptionsBuilder<BloggingContext>()
    .UseSqlServer(connection)
    .Options;

using var context1 = new BloggingContext(options);
using var transaction = context1.Database.BeginTransaction();
try
{
    context1.Blogs.Add(new Blog { Url = "http://blogs.msdn.com/dotnet" });
    context1.SaveChanges();

    using (var context2 = new BloggingContext(options))
    {
        context2.Database.UseTransaction(transaction.GetDbTransaction());

        var blogs = context2.Blogs
            .OrderBy(b => b.Url)
            .ToList();
    }

    // Commit transaction if all commands succeed, transaction will auto-
rollback
    // when disposed if either commands fails
    transaction.Commit();
}
catch (Exception)
```

```
{
    // TODO: Handle failure
}
```

## Using external DbTransactions (relational databases only)

If you are using multiple data access technologies to access a relational database, you may want to share a transaction between operations performed by these different technologies.

The following example, shows how to perform an ADO.NET SqlClient operation and an Entity Framework Core operation in the same transaction.

C#

```
using var connection = new SqlConnection(connectionString);
connection.Open();

using var transaction = connection.BeginTransaction();
try
{
    // Run raw ADO.NET command in the transaction
    var command = connection.CreateCommand();
    command.Transaction = transaction;
    command.CommandText = "DELETE FROM dbo.Blogs";
    command.ExecuteNonQuery();

    // Run an EF Core command in the transaction
    var options = new DbContextOptionsBuilder<BloggingContext>()
        .UseSqlServer(connection)
        .Options;

    using (var context = new BloggingContext(options))
    {
        context.Database.UseTransaction(transaction);
        context.Blogs.Add(new Blog { Url = "http://blogs.msdn.com/dotnet"
});
        context.SaveChanges();
    }

    // Commit transaction if all commands succeed, transaction will auto-
rollback
    // when disposed if either commands fails
    transaction.Commit();
}
catch (Exception)
{
    // TODO: Handle failure
}
```

## Using System.Transactions

It is possible to use ambient transactions if you need to coordinate across a larger scope.

C#

```csharp
using (var scope = new TransactionScope(
           TransactionScopeOption.Required,
           new TransactionOptions { IsolationLevel =
IsolationLevel.ReadCommitted }))
{
    using var connection = new SqlConnection(connectionString);
    connection.Open();

    try
    {
        // Run raw ADO.NET command in the transaction
        var command = connection.CreateCommand();
        command.CommandText = "DELETE FROM dbo.Blogs";
        command.ExecuteNonQuery();

        // Run an EF Core command in the transaction
        var options = new DbContextOptionsBuilder<BloggingContext>()
            .UseSqlServer(connection)
            .Options;

        using (var context = new BloggingContext(options))
        {
            context.Blogs.Add(new Blog { Url = "http://blogs.msdn.com/dot-
net" });
            context.SaveChanges();
        }

        // Commit transaction if all commands succeed, transaction will
auto-rollback
        // when disposed if either commands fails
        scope.Complete();
    }
    catch (Exception)
    {
        // TODO: Handle failure
    }
}
```

It is also possible to enlist in an explicit transaction.

C#

```
using (var transaction = new CommittableTransaction(
           new TransactionOptions { IsolationLevel =
IsolationLevel.ReadCommitted }))
{
    var connection = new SqlConnection(connectionString);

    try
    {
        var options = new DbContextOptionsBuilder<BloggingContext>()
            .UseSqlServer(connection)
            .Options;

        using (var context = new BloggingContext(options))
        {
            context.Database.OpenConnection();
            context.Database.EnlistTransaction(transaction);

            // Run raw ADO.NET command in the transaction
            var command = connection.CreateCommand();
            command.CommandText = "DELETE FROM dbo.Blogs";
            command.ExecuteNonQuery();

            // Run an EF Core command in the transaction
            context.Blogs.Add(new Blog { Url = "http://blogs.msdn.com/dot-
net" });
            context.SaveChanges();
            context.Database.CloseConnection();
        }

        // Commit transaction if all commands succeed, transaction will
auto-rollback
        // when disposed if either commands fails
        transaction.Commit();
    }
    catch (Exception)
    {
        // TODO: Handle failure
    }
}
```

## Limitations of System.Transactions

1. EF Core relies on database providers to implement support for System.Transactions. If a provider does not implement support for System.Transactions, it is possible that calls to these APIs will be completely ignored. SqlClient supports it.

   **Important**

   It is recommended that you test that the API behaves correctly with your provider before you rely on it for managing transactions. You are encouraged to contact the maintainer of the database provider if it does not.

2. Distributed transaction support in System.Transactions was added to .NET 7.0 for Windows only. Any attempt to use distributed transactions on older .NET versions or on non-Windows platforms will fail.