Modularity is as common Marketing concept, however achieving True Modularity in a Code base is not so easy.

One key aspect of Modularity is to have code that can correctly run regardless of its Application context (or at least be the most possible independent of it). Here we’ll look at one aspect: Connection and Transaction and introduce helpers to solve some issues: CK.SqlServer, CK.SqlServer.Transaction and CK.SqlServer.Dapper.

Contents

[Abstract 1](#_Toc536198314)

[Basics of ADO.Net model 1](#_Toc536198315)

[The Connection/Transaction pair applied to Dapper API 3](#_Toc536198316)

[The transaction scope idea 3](#_Toc536198317)

[CK.SqlServer: ISqlCallContext, ISqlConnectionController & SqlStandardCallContext 3](#_Toc536198318)

[CK.SqlServer.Transaction: ISqlTransactionCallContext 5](#_Toc536198319)

[The final touch: Modularity in practice 6](#_Toc536198320)

# Abstract

By using CK.SqlServer, basic Components that depends on ISqlCallContext are *de facto* compliant with a potential existing transaction.

CK.SqlServer.Dapper gives this transaction transparency to the very good Dapper helper.

Components that have to deal with transactions rely on CK.SqlServer.Transaction so that they can define transactions and nested transactions (and even play with isolation levels) and thanks to the IScopedAutoService marker interface Applications composed of basic (transaction-free) Components and transaction-aware Components work just fine.

# Basics of ADO.Net model

When considering the object model, Transaction and Connection seem to be able to live independently:

* The Connection is the factory of the Transaction (Connection.BeginTransaction() is the only way to actually create a transaction).
* A Connection doesn’t expose any Transaction.
* A Transaction exposes its Connection (its creator).
* A Command exposes Connection and Transaction properties that must be set before its execution.

So far, so good. Unfortunately, this object model is designed in a strange way (this is due – up to me – to the Component Model that enables Connection and Transaction to be “edited” on a “Design Surface” like in Windows Form): it has not been thought for code…

Let’s make simple tests to understand some issues:

1 - Transactions cannot be nested from application code.

[Test]

public void nested\_transactions\_are\_not\_supported()

{

using( var c = TestHelper.CreateOpenedConnection() )

{

using( var t1 = c.BeginTransaction() )

{

c.Invoking( \_ => \_.BeginTransaction() )

.Should().Throw<InvalidOperationException>();

}

}

}

* System.InvalidOperationException : SqlConnection does not support parallel transactions.

This is true for SqlConnection (but not necessarily for all providers). However, nested transaction is a fundamental requirement for modularity:

* Basic Components **doesn’t require a transactional context** BUT **must be able to honor it** if, higher in the call stack, a caller requires one.
* More complex components **must be able to use Transactions** and **at the Isolation level** that fits their needs, regardless of the Components that are using them and the Components they use.

We must handle/correct this.

2 – Connection and Transaction MUST be used together:

[Test]

public void connection\_and\_transaction\_must\_exactly\_match()

{

using( var c1 = TestHelper.CreateOpenedConnection() )

using( var c2 = TestHelper.CreateOpenedConnection() )

using( var cmd = new SqlCommand( "select 1;" ) )

{

var t1 = c1.BeginTransaction();

cmd.Connection = c2;

cmd.Transaction = t1;

cmd.Invoking( \_ => \_.ExecuteScalar() )

.Should().Throw<InvalidOperationException>();

}

}

* System.InvalidOperationException : The transaction is either not associated with the current connection or has been completed.

Let’s try to execute a command on a connection “outside” the transaction currently associated to the connection.

[Test]

public void command\_MUST\_be\_associated\_to\_the\_connection\_transaction()

{

using( var c = TestHelper.CreateOpenedConnection() )

using( var cmd = new SqlCommand( "select 1;" ) )

{

var t = c.BeginTransaction();

cmd.Connection = c;

cmd.Invoking( \_ => \_.ExecuteScalar() )

.Should().Throw<InvalidOperationException>();

}

}

* System.InvalidOperationException : ExecuteScalar requires the command to have a transaction when the connection assigned to the command is in a pending local transaction. The Transaction property of the command has not been initialized.

Conclusion: as soon as a Connection is associated to a Transaction, each and every Command that run on it must be configured with the associated Transaction.

# The Connection/Transaction pair applied to Dapper API

The fact that a Connection doesn’t expose any Transaction property, explains why a typical Dapper extension method looks like this:

public static int Execute( this IDbConnection cnn,

string sql,

object param = null,

IDbTransaction transaction = null )

How Dapper can be used in a code base that MAY be transactional? Where does the transaction comes from?

My bet: no developer actually cares about this optional parameter and as the test above demonstrates it, this will fail miserably if someone opened a transaction…

This where CK.SqlServer.Dapper enters the scene: Dapper is available on the ISqlConnectionController (defined in CK.SqlServer) and its methods don’t need the IDbTransaction.

The previous method becomes:

public static int Execute( this ISqlConnectionController c, string sql, object param = null )

But there is more in the whole picture than this little simplification.

# The transaction scope idea

First, a few words about the “solution” that is currently used by many architects in many applications: the transaction scope.

See here for the official presentation: <https://docs.microsoft.com/en-us/dotnet/framework/data/transactions/implementing-an-implicit-transaction-using-transaction-scope>.

It seems to be THE solution. It is not!

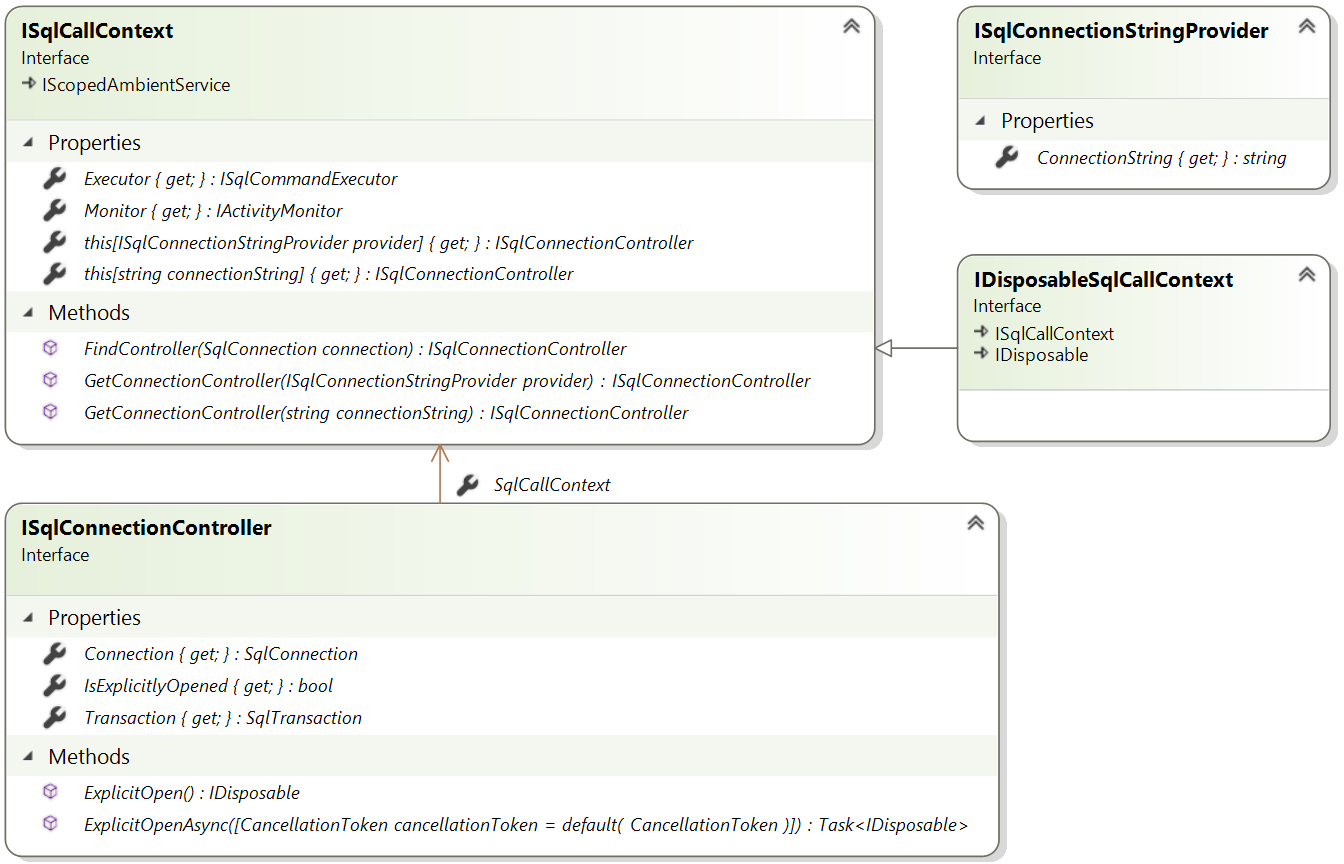
* Transaction scope is meant to transparently escalate to Distributed Transaction. I wish it never happens to you.
  + To coordinate different transactional subsystem, we clearly decide that this be done through compensations or other explicit handling.
* “Implicitness” is the key of this design, and implicit will hurt you.

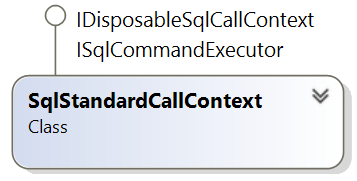
And there are subtle drawbacks that will definitely break modularity: please read this good and short post <https://weblogs.thinktecture.com/pawel/2018/06/entity-framework-core-use-transactionscope-with-caution.html>. In application of its conclusion: we don’t use it. Never.

Now that Transaction scope is evicted, let’s present the CK.SqlServer and CK.SqlServer.Transaction solution.

# CK.SqlServer: ISqlCallContext, ISqlConnectionController & SqlStandardCallContext

CK.SqlServer brings a basic transaction-free model with a root abstraction: the ISqlCallContext interface that gives access to ISqlConnectionController.



One single concrete class is enough to use this model:

* This class holds and manage connections for multiple connection strings: a connection controller is created and cached once for all for each required connection string.
* The design of IDisposable support secures the pattern. Thanks to the IDisposableSqlCallContext, only the root can be disposed.

Typical direct usage pattern in simple (This is a test of CK.SqlServer.Dapper):

[Fact]

public async Task ExecuteReaderClosedAsync()

{

using( var ctx = new SqlStandardCallContext( TestHelper.Monitor ) )

{

ISqlConnectionController conn = ctx[TestHelper.GetConnectionString()];

var dt = new DataTable();

dt.Load( await conn.ExecuteReaderAsync( "select 3 as [three], 4 as [four]" )

.ConfigureAwait( false ) );

Assert.Equal( 2, dt.Columns.Count );

}

}

During the lifetime of the call context, connections live their own life wrapped in controllers. It is not required to open the connection: the connection may be opened/closed for each request, or be pre-opened and kept opened thanks to:

/// <summary>

/// Opens the connection to the database if it were closed.

/// The internal count is always incremented.

/// Returns a IDisposable that will allow the connection to be disposed when disposed.

/// If this IDisposable is not disposed, the connection will be automatically disposed

/// when the root IDisposableSqlCallContext will be disposed.

/// </summary>

/// <returns>A IDisposable that can be disposed.</returns>

IDisposable ExplicitOpen();

Or, until the disposal of the call context, with:

/// <summary>

/// Simple relay to ISqlConnectionController.ExplicitOpen that forgets the returned

/// IDisposable. The connection will remain opened until the holding IDisposableSqlCallContext

/// is disposed.

/// </summary>

/// <param name="ctx">This connection controller.</param>

public static void PreOpen( this ISqlConnectionController ctx )

A minimal set of extension methods are defined on the ISqlConnectionController (see <https://github.com/Invenietis/CK-SqlServer/blob/develop/CK.SqlServer/SqlConnectionControllerExtension.cs>) that helps using it directly without too much pollution of the API, leaving room for more complex helpers (like Dapper).

Let’s now handle the transactions.

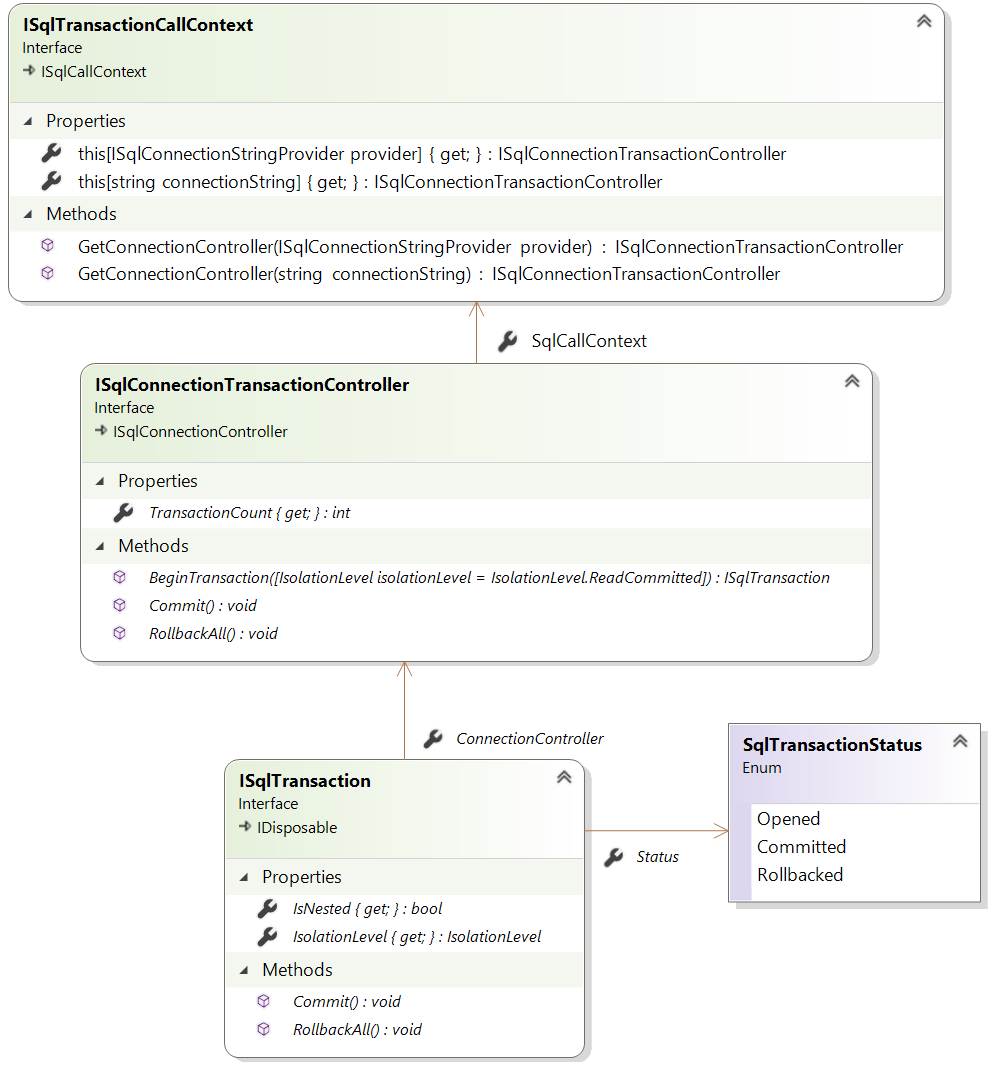
# CK.SqlServer.Transaction: ISqlTransactionCallContext

CK.SqlServer.Transaction exposes interfaces that extends the basic ones and use masking (the new declaration operator) to specialize the API. The ISqlConnectionTransactionController offers the BeginTransaction method that is the factory of ISqlTransaction.

The ISqlTransaction model is quite simple, however it transparently handles nested transactions (even if under, SqlTransaction does not support this). This is achieved by explicitly emitting set transaction level instructions to the engine.

Thanks to this, any Component can always begin a transaction (at a given isolation level) regardless of their caller context.

Playing with transaction level is often considered complicated and this is true. Please read carefully before changing the default ReadCommitted level: <https://docs.microsoft.com/en-us/sql/t-sql/statements/set-transaction-isolation-level-transact-sql?view=sql-server-2017#remarks>



Notice the ISqlTransaction.RollbackAll() method. This clarifies the fact that a Rollback on any opened transaction, whatever its nested level is, actually rolls back the whole transaction. Once a transaction is rolled back all ISqlTransaction.Status of all opened transactions become “Rollbacked”.

This naming is not the standard one (usually it is simply named “Rollback”) and this is on purpose: we plan to support (one day), “Atomic Transactions” where a Rollback will only roll back the nested transaction, living the transactions above opened.

The concrete SqlTransactionCallContext implements the whole model.

# The final touch: Modularity in practice

This is where the magic is and justifies the architecture. Basic components don’t need transactions: they depend on ISqlCallContext and do what their job with it. Basic applications don’t need client transactions: everybody is happy with the small CK.SqlServer package (and may be with the even smaller CK.SqlServer.Dapper): a basic application globally depends on CK.SqlServer.

Now, let’s add a more complex Component to this application that requires transactions: it depends on CK.SqlServer.Transaction and everything works fine:

* ISqlCallContext is marked with the locally defined IScopedAutoService (defined here: <https://github.com/Invenietis/CK-SqlServer/blob/develop/CK.SqlServer/StObjSupport/IScopedAutoService.cs>).
* The resolution will make SqlTransactionCallContext the implementation for both ISqlCallContext and ISqlTransactionCallContext interfaces.
* All injected call contexts are de facto transactional but this aspect doesn’t bother the basic ones (and they may do their job inside a transaction if one has been opened).