Tortuga Chain

A Fluent ORM for .NET

Getting Started

To get started with Chain, you need to create a data source. This can be done using a connection string or a `SqlConnectionStringBuilder`. Optionally, you can also name your data source. (This has no functional effect, but does assist in logging.)

```
dataSource = new Tortuga.Chain.SqlServerDataSource("Adventure DB", "Server=.;Database=AdventureWorks2014;Trusted_Connection=True;");
```

Or from your app.config file:

```
dataSource = Tortuga.Chain.SqlServerDataSource.CreateFromConfig("AdventureDB");
```

Your data source should be treated as a singleton object; you only need one per unique connection string. This is important because your data source will cache information about your database.

We recommend calling dataSource.Test() when your application starts up. This verifies that you can actually connect to the database.

Connection Management

A major difference between Chain and other ORMs is that you don't need to manage connections or data contexts. A Chain data source is designed to be completely thread safe and will handle connection lifetime for you automatically.

Transactions

Transactions still need to contained within a `using` statement and explicitly committed. You can create one by calling `dataSource.BeginTransaction()`.

Command Chains

Command chains are the primary way of working with Tortuga. Each link in the chain is used to inform the previous link about what actions are desired. Here is a basic example:

```
dataSource.Procedure("uspGetEmployeeManagers", new {@BusinessEntityID = 100}).ToCollection<Manager>().Execute();
```

Breaking this down, we have:

- The data source
- The command being performed
- How the results of the command should be returned
- If the operation should be executed synchronously or asynchronously

Commands

The list of available commands depends on the data source. Most data sources support

- Raw sql
- Table/View queries
- Insert, Update, and Delete operations (some also include 'upserts')

Advanced ones may also include

- Stored procedures and/or Table Value Functions
- Batch insert, a.k.a. bulk copy

Most commands accept a parameter object. The parameter object can be a normal class, a dictionary of type `IDictionary<string, object>`, or a list of appropriate DbParameter objects.

Chain command builders honor .NET's `NotMapped` and `Column` attributes.

Materializers

Materializers are an optional link, you only need them if you want something back from the database.

An interesting feature of the materializer is that it participates in SQL generation. For example, if you use the `ToObject<T>` or `ToCollection<T>` materializer, then it will read the list of properties on class T. That list of properties will be used to generate the SELECT clause, ensuring that you don't pull back more information than you actually need. This in turn means that indexes are used more efficiently and performance is improved.

Materializers call into several categories:

- Scalar: `ToInt`, `ToIntOrNull`, `ToString`
- Row: `ToRow`, `ToDataRow`, `ToObject`
- Table: `ToTable`, `ToDataTable`, `ToCollection`
- Multiple Tables: `ToTableSet`, `ToDataSet`

For better performance, you can use the compiled materializer extension:

- Row: `.Compile().ToObject<TObject>()`
- Table: `.Compile().ToCollection<TObject>()`, `.Compile().ToCollection<TList, TObject>()`

This requires the `Tortuga.Chain.CompiledMaterializers` package, which includes CS-Script as a dependency.

CRUD Operations

By combining commands and materializers, you can perform all of the basic CRUD operations. Here are some examples.

Create

```
var vehicleKey = dataSource.Insert("Vehicle", new { VehicleID = "65476XC54E", Make = "Cadillac", Model = "Fleetwood Series 60", Year = 1955 }).ToInt32().E
```

Read

```
var car = dataSource.GetById("Vehicle", vehicleKey).ToObject<Vehicle>().Execute();
var cars = dataSource.From("Vehicle", new {Make = "Cadillac").ToCollection<Vehicle>().Execute();
```

Update

```
dataSource.Update("Vehicle", new { VehicleKey = vehicleKey, Year = 1957 }).Execute();
```

Delete

```
dataSource.Delete("Vehicle", new { VehicleKey = vehicleKey }).Execute();
```

Appenders

Appenders are links that can change the rules before, during, or after execution. An appender can be added after a materializer or another appender.

Caching appenders include:

- `Cache` : Writes to the cache, overwriting any previous value. (Use with Update and Procedure operations.)
- `ReadOrCache` : If it can read from the cache, the database operation is aborted. Otherwise the value is cached.
- `CacheAllItems` : Cache each item in the result list individually. Useful when using a GetAll style operation.
- `InvalidateCache` : Removes a cache entry. Use with any operation that modifies a record.

Here is an example of CRUD operations using caching.

```
var car = dataSource.GetById("Vehicle", vehicleKey).ToObject<Vehicle>().ReadOrCache("Vehicle " + vehicleKey).Execute();
car = dataSource.Update("Vehicle", new { VehicleKey = vehicleKey, Year = 1957 }).ToObject<Vehicle>().Cache("Vehicle " + vehicleKey).Execute();
dataSource.Delete("Vehicle", new { VehicleKey = vehicleKey }).InvalidateCache("Vehicle " + vehicleKey.Execute();
```

If using SQL Server, you can also use `WithChangeNotification` . This uses SQL Dependency to listen for changes to the table(s) you queried.

When debugging applications, it is often nice to dump the SQL somewhere. This is where the tracing appenders come into play.

- `WithTracing` : Writes to an arbitrary TextWriter style stream.
- `WithTracingToConsole` : Writes to the Console window
- `WithTracingToDebug` : Writes to the Debug window

You can also override DBCommand settings such as the command timeout. For example:

```
ds.Procedure("ExpensiveReport").ToDataSet().SetTimeout(TimeSpan.FromHours(3)).Execute()
```

Execution Modes

The final link in any chain is the execution mode. There are two basic options:

- `Execute()`
- `ExecuteAsync()`

Both options accept a `state` parameter. This has no direct effect, but can be used to facilitate logging. `ExecuteAsync` also accepts an optional cancellation token.

Data Sources

At the beginning of every chain is a `DataSource` . A normal data source represents an abstract connection to the database. The key word here is 'abstract'; this is not an actual connection but rather all of the information needed to create and manage one.

Normally an application will only have one instance of a data source per connection string. This is important, as the data source needs to cache information about your database and application.

Basic Configuration

Most of the time, all of the configuration needed to create a data source can be found with a normal ADO.NET style connection string. The connection string can be passed to the constructor as a `String` or the appropriate subclass of `DbConnectionStringBuilder` . Optionally you can also provide a name for the data source. The name has no effect, but may be useful for logging.

On the .NET Framework, data sources also provide a factory method called `CreateFromConfig` for reading the connection string from the app.config file. (This is not available on UWP.)

Advanced Configuration

For some databases, you may override connection settings beyond those offered by the connection string. For example, for SQL Server you can alter `dataSource.Settings.ArithAbort` and `dataSource.Settings.XactAbort` . These overrides are applied each time a connection is opened by the data source.

The `Settings` object is database specific and will not necessarily be available for all types of data sources.

Data Source Classifications

Chain is primarily designed to take advantage of the capabilities of a database. Rather than catering to the "lowest common denominator" like many ORMs, we'd rather give you full access to everything your database is capable of supporting.

However, we realize that some projects actually do need database portability. For those project we offer a set of data source classifications, which are expressed in terms of interfaces.

IClass1DataSource

A class 1 data source provides the necessary functionality to support a repository pattern using CRUD style operations. Briefly these include:

- Database metadata (tables, views, columns, etc.)
- From
- Insert: Single row inserts

- Update: Single row updates
- Upsert: Automatic insert or update as appropriate
- Delete: Single row deletes
- Sql: Raw SQL statements with optional parameters.

Separate pages will be offered to document each of these operations.

IClass2DataSource

We are still identifying the minimum necessary features to classify something as class 2 data source. The current thought is that it would have to support the following operations:

- Procedure: Execute a stored procedure
- Bulk Insert: For quickly loading a lot of data [Milestone 1.1]
- Function: Query from table-valued functions [Milestone 1.2]

Internals

Command Execution

The database contains an Execute and ExecuteCore method for processing execution tokens. Within these methods the connection is opened and the DBCommand is created. Materializers provide a call-back through which the command is executed and the results are processed.

Commands are executed by the materializer callback because neither the execution token nor the data source know whether you need `ExecuteNonQuery`, `ExecuteScalar`, or `ExecuteReader`. This also gives the materializer the option to set execution mode flags such as `Sequential`.

Metadata

The data source contains metadata about database such as a list of tables and views. While available to the application, this information is primary used by the command builders to generate SQL.

In order to achieve reasonable performance, the data source caches the metadata. This means that the data source can become out of sync with the database if the schema changes while the application is running. Currently there is no way to "reset" the cache, so if that happens you'll need to create a new instance of the data source.

Extension Data

Some Chain extensions such as compiled materializers need to cache data source specific information. For these, there is the `DataSource.GetExtensionData` method. Each extension should only store one object here. Any additional information should hang off that object rather than creating multiple keys.

Data Caching

Data caching is performed through the data source object. Originally the intention was to use the `System.Runtime.Caching` framework, which would have allowed developers to plug in any caching framework they want. However, it turns out that framework has some design flaws and we're going to need to create our own caching interface and matching adapters.

In the meantime, we'll be using .NET's MemoryCache but are not exposing it. That way there won't be a breaking change once we settle on a caching interface.

Universal Windows Platform Limitations

UWP does not support the `System.Runtime.Caching` framework, which in turn means that Chain for UWP doesn't support it either. Again, this will change once we develop our own caching interface and start writing adapters for the more popular caching libraries and servers.

Command Builders

A command builder is used to generate SQL for a specific type of operation. SQL generation does not occur immediately, but rather waits until you've selected a materializer and, optionally, appenders.

The reason for the delay is that materializers often change the way the SQL is generated. For example, when performing an Insert operation you have the option of getting back nothing, just the newly inserted primary key, or an entire object with any defaulted/calculated columns.

Since each command builder works differently, we can't really go into details here.

Internals

Metadata Consumption

When a command builder is created, it may immediately fetch metadata from the data source. This means that you could potentially see blocking operation and/or database error before you actually try to execute a Chain.

In practice this shouldn't be a problem, as it should only occur the first time a given table/view is referenced. But if you are concerned, you can call `dataSource.Metadata.PreloadTables` and `PreloadViews` during startup. The time it takes to run these operations varies with the size of the database schema, so it may negatively impact performance.

Execution

When a chain is executed, the materializer calls `Prepare` on the command builder, which in turn creates an execution token. This token contains the SQL statement and parameters needed to build and execute a `System.Data.Common.DBCommand` object. The materializer and appenders can further modify the execution token from here, but the command builder is no longer in the process.

Materializers

Materializers represent the `SELECT` part of the SQL generation process. They can be used to request the result of an operation be formatted as a scalar value, a list, an object, a collection, or a traditional `DataTable`. The base classes are public so that additional materializers can be created in the same fashion as the built-in ones.

Materializers are always linked directly to command builders. They are usually expressed as `.ToXxx` where Xxx is the type of result desired. There is also a special `AsNonQuery()` command builder for when you don't need anything back (e.g. insert/update operations). Many materializers allow you to set options that further refine their behavior.

Generally speaking, the classes that implement materializers are non-public. Instead you get back an `ILink` or `ILink<T>` wrapper. This is necessary as the concrete implementation of the materializer may change depending on the options you select. It is best to think of an `ILink` in the same way you think of an `IEnumerable` in a LINQ expression.

Sql

All materializers offer a `Sql` method. This returns the SQL that would have been executed.

Note that this is not the only way to access the SQL being generated and is primarily meant to be used when debugging. For other situations, it is typical to access the SQL via the logging events and with a Trace appender.

Internals

When a materializer is executed, it first calls `Prepare` on the command builder it is linked to in order to generate an execution token. Immediately afterwards, it fires an ExecutionTokenPrepared event that appenders can listen for. (If you request the SQL from a materializer, only this step is performed.)

Next, the materializer calls Execute (or ExecuteAsync) on the execution token, passing in a callback. This callback is given a `DBCommand` object on which the materializer can call `ExecuteNonQuery`, `ExecuteScalar`, or `ExecuteReader` as appropriate.

The callback is expected to return an integer which is either null or the number of rows affected by the operation. Here is an example `Execute` method with callback:

```
public override DataTable Execute(object state = null)
{
    DataTable dt = new DataTable();
    ExecuteCore(cmd =>
    {
        using (var reader = cmd.ExecuteReader(CommandBehavior.SequentialAccess))
        {
            dt.Load(reader);
            return dt.Rows.Count;
        }
    }, state);

    return dt;
}
```

Limitations

A strict limitation on materializers is that if they open an `IDataReader` they must close it inside their callback routine. There is no facility for holding open the connection associated with the data reader.

Appenders

In the API, appenders are exposed as `ILink` or `ILink<T>` just like materializers. The difference is that while a materializer hooks onto a command builder, an appender hooks onto other links. This means you can string together as many appenders as you need to get the desired effect.

The mostly commonly used appenders deal with caching. You can also find appenders for tracing SQL calls, modifying command timeouts, and listening to change notification from the database (see SQL Dependency Appenders).

Internals

Appends can be loosely grouped into these categories:

- Before execution
- After execution
- Instead of execution

Appenders can fit into more than one category.

Before execution

A before execution appender is usually involved with the command building process in some way. Here we see one that alters the connection timeout:

```
protected override void OnCommandBuilt(CommandBuiltEventArgs e)
{
    if (e == null)
        throw new ArgumentNullException("e", "e is null.");
    e.Command.CommandTimeout = (int)m_Timeout.TotalSeconds;
}
```

The tracing appenders also respond to this event, but they merely read the command text instead of modifying it.

Another type of before execution appender is InvalidateCache, which you can see here:

```
public override void Execute(object state = null)
{
    DataSource.InvalidateCache(m_CacheKey);

    PreviousLink.Execute(state);
}
```

Note that the ExecuteAsync method will also have to be overridden with nearly identical logic.

After execution

After execution appenders tend to work on the cache. Here is one that caches the result of an operation.

```
public override TResult Execute(object state = null)
{
    var result = PreviousLink.Execute(state);
    DataSource.WriteToCache(new CacheItem(m_CacheKey ?? m_CacheKeyFunction(result), result, null), m_Policy);
    return result;
}
```

Instead of execution

The quintessential instead of appender is `ReadOrCache`. As the name implies, this will read a value from the cache, and if not found executes the query and caches the resulting value. Here is an example:

```
public override TResult Execute(object state = null)
{
    TResult result;
    if (PreviousLink.DataSource.TryReadFromCache(m_CacheKey, out result))
        return result;

    result = PreviousLink.Execute(state);

    DataSource.WriteToCache(new CacheItem(m_CacheKey, result, null), m_Policy);

    return result;
}
```

As you can see, this appender has an effect that occurs both before and optionally after execution.

Creating an Appender

Using the example above and the `Appender`, `Appender<TResult>`, or `Appender<Tin, TOut>` base class, you can implement your own appender. If your appender is database specific, you may need to cast the execution into the correct subtype.

From Command Builder

This generates a SELECT statement against a table or view.

Arguments

If just the table name is provided, all rows are returned.

If an object (model or Dictionary<string, object>) is provided, a WHERE clause is generated in the form of "Column1 = @Value1 AND Column2 = @Value2 [...]".

If a filter string is provided, it is used as the where clause. Optionally, an object can be provided as parameters for the query.

Sorting

To add sorting, use the `.WithSorting(...)` method. This accepts a list of strings or SortExpressions (the latter is only needed for descending sorts). To prevent SQL injection attacks, column names are validated against the database.

Limits

To add limits, use the `.WithLimits(...)` method. The type of limits avaialble vary from database to database. Most provide Top, Offset/Fetch, and one or more forms of random sampling.

Sorting is often required when using Offset/Fetch. It is not allowed when using random sampling.

Filters

Filters can be added (or replaced) using the `.WithFilter(...)` method. This uses the same filters (WHERE clause or filter object) that are available when calling `.From(...)`.

Limitations

When using a filter string with filter parameter object, the SQL generator won't be able to determine which properties/keys are applicable. Instead, all of them will be sent to the database.

When using a filter string with filter parameter object, and one of the filtered columns is an ANSI string (`char` / `varchar`), you may experience performance degradation. To avoid this, pass in an array of `DbParameter` objects with the correct `DbParameter.DbType` value. (Normally DbType is inferred by checking against the column's type.)

SQL Generation

If no columns are desired by the materializer, 'SELECT 1' is returned instead.

If soft deletes are enable for this table, the WHERE clause will filter out deleted rows.

Internals

All databases work the same expect for what types of limits they support.

Roadmap

GetByKey, UpdateByKey, and DeleteByKey Command Builders

This generates a `SELECT`, `UPDATE`, or `DELETE` statement against a table using a key or list of keys.

Arguments

- The `…ByKey` variants accept one or more keys.
- The `…ByKeyList` variants accept a list of keys.

Keys must be scalar values (string, int guid, etc.)

Arguments, `UpdateByKey` Only

The `newValues` object is used to generate the `SET` clause of the `UDPATE` statement in the same manner as the normal `Update` command builder.

Sorting

Rows are returned in a non-deterministic order.

Limitations

This feature only works on tables that have a scalar primary key.

SQL Generation

This will generate a WHERE clause in the form of `PrimaryKey = @Param` or `PrimaryKey in (@Param1, @Param2, ...)`.

Roadmap

Possible enhancement: use tuples to support compound primary keys. This would require an upgrade to the metadata provider, as we would need to know the exact order for the primary key.

Insert Command Builder

The command performs an insert using the provided model.

Arguments

A parameter object is used to construct the INSERT statment. If a property is marked with the IgnoreOnInsert attribute, it will not participate in this step. This attribute is usually applied to defaulted columns such as CreatedDate.

Alternately, a parameter dictionary of type IReadonlyDictionary<string, object> may be used. Again, the primary keys will be read from database metadata.

SQL Generation

If the materializer desires columns, this echos back the newly inserted row.

Limitations

This command is meant to operate on one row at a time.

Internals

Implementation of the insert varies by database.

SQL Server

SQL Server uses an INSERT statement with an optional OUTPUT clause.

SQLite

SQLite uses a separate INSERT and optional SELECT statement.

Roadmap

We should create set-based insert. The biggest question is how to handle the materializer, as we may not necessarily be able to match up the returned identity values with the original objects.

Update Command Builder

The command performs an update using the provided model.

Arguments

The parameter object must contain the primary key(s) necessary to locate the record. The keys are normally read from the database, but you can override this behavior to use properties on the object that use the Key attribute.

The remaining properties on the parameter object are used to construct the UPDATE statment. If a property is marked with the IgnoreOnUpdate attribute, it will not participate in this step. This attribute is usually applied to columns that should never be changed such as CretedBy/CreatedDate.

If you use the ChangedPropertiesOnly option, only properties marked as changed by `IPropertyChangeTracking.ChangedProperties()` will be used. This interface is found in the `Tortuga.Anchor.ComponentModel` namespace.

Alternately, a parameter dictionary of type `IReadonlyDictionary<string, object>` may be used. Again, the primary keys will be read from database metadata.

SQL Generation

If the materializer desires columns, this echos back the inserted/updated row.

Limitations

This command is meant to operate on one row at a time. Set-based operations need to be performed using a SQL or Procedure command.

Internals

Implementation of the delete varies by database.

SQL Server

SQL Server uses an UPDATE statement with an optional OUTPUT clause.

SQLite

SQLite uses a separate UPDATE and optional SELECT statement.

Roadmap

Upsert Command Builder

The command performs a insert or update using the provided model.

Arguments

The parameter object must contain the primary key(s) necessary to locate the record. The keys are normally read from the database, but you can override this behavior to use properties on the object that use the Key attribute.

Upsert will honor the IgnoreOnInsert and IgnoreOnUpdate attributes.

If you use the ChangedPropertiesOnly option, only properties marked as changed by `IPropertyChangeTracking.ChangedProperties()` will be used for the insert part of the operation. This interface is found in the `Tortuga.Anchor.ComponentModel` namespace.

Alternately, a parameter dictionary of type IReadonlyDictionary<string, object> may be used. Again, the primary keys will be read from database metadata.

SQL Generation

If the materializer desires columns, this echos back the row. The UpsertOptions flag determines if the original or new values are returned.

Limitations

This command is meant to operate on one row at a time. Set-based operations need to be performed using a SQL or Procedure command.

Internals

Implementation of the upset varies by database.

SQL Server

Upserts are performed via a MERGE command. This command is inheritently non-atomic and may throw an exception if there is a race condition. If this occurs, the application needs to decide whether or not to retry the operation.

SQLite

This requires a 3 step process:

- Update the existing row (if it exists)
- Insert a new row, silently failing if it already exists
- Select the recently inserted/updated row. (If the correct materializer was selected.)

Roadmap

Delete Command Builder

The command performs a delete using the provided model.

Arguments

The parameter object must contain the primary key(s) necessary to locate the record. The keys are normally read from the database, but you can override this behavior to use properties on the object that use the Key attribute.

Alternately, a parameter dictionary of type IReadonlyDictionary<string, object> may be used. Again, the primary keys will be read from database metadata.

SQL Generation

If the materializer desires columns, this will return the row as it looked before it was deleted.

If soft delete is enabled for this table, an update operation will be generated to set the deleted flag.

Limitations

This command is meant to operate on one row at a time. Set-based operations need to be performed using a SQL or Procedure command.

Internals

Implementation of the delete varies by database.

SQL Server

SQL Server uses an UPDATE statement with an optional OUTPUT clause.

SQLite

SQLite uses a separate DELETE and optional SELECT statement.

Roadmap

SQL Command Builder

This command executes arbitrary SQL. It makes no attempt to pre-validate said SQL.

Arguments

An optional object (model or IDictionary<string, object>) may be provided when executing paramterized SQL.

Limitations

When passing in paramters, the SQL generator won't be able to determine which properties/keys are applicable. Instead, all of them will be sent to the database.

SQL Generation

N/A

Internals

Roadmap

Table Function Command Builder

This generates a SELECT statement against a table-valued function.

Function Arguments

Function arguments are provided as part of the `dataSource.TableFunction(...)` method. All parameters must be accounted for.

Sorting

To add sorting, use the `.WithSorting(...)` method. This accepts a list of strings or SortExpressions (the latter is only needed for descending sorts). To prevent SQL injection attacks, column names are validated against the database.

Limits

To add limits, use the `.WithLimits(...)` method. The type of limits avaialble vary from database to database. Most provide Top, Offset/Fetch, and one or more forms of random sampling.

Sorting is often required when using Offset/Fetch. It is not allowed when using random sampling.

Filters

Filters can be added (or replaced) using the `.WithFilter(...)` method. This uses the same filters (WHERE clause or filter object) that are available when calling `.From(...)`.

If just the table name is provided, all rows are returned.

If an object (model or Dictionary<string, object>) is provided, a WHERE clause is generated in the form of "Column1 = @Value1 AND Column2 = @Value2 [...]".

If a filter string is provided, it is used as the where clause. Optionally, an object can be provided as parameters for the query.

Do not use the same arguments for the filter expression as you use for the table parameters, as this may caused unintended behavior.

Limitations

When using a filter string with filter parameter object, the SQL generator won't be able to determine which properties/keys are applicable. Instead, all of them will be sent to the database.

When using a filter string with filter parameter object, and one of the filtered columns is an ANSI string (`char` / `varchar`), you may experience performance degradation. To avoid this, pass in an array of `DbParameter` objects with the correct `DbParameter.DbType` value. (Normally DbType is inferred by checking against the column's type.)

Some types of random sampling are not available with table-valued functions.

SQL Generation

If no columns are desired by the materializer, 'SELECT 1' is returned instead.

Internals

All databases work the same expect for what types of limits they support.

Roadmap

Procedure Command Builder

This command is used to execute a stored procedure that may return one or more record sets.

Arguments

If an object parameter is provided (model or IDictionary<string, object>), the procedure's paratmeters will be compared to those on the object. Where they match, the value will be passed in.

SQL Generation

SQL Server

This command will use standard `CommandType.StoredProcedure` semantics.

PostgreSQL

This command is only used for functions that return one or more `refcursor` objects.

This command will use standard `CommandType.StoredProcedure` semantics to get a list of cursors. Then for each cursor, `FETCH ALL IN "cursor_name"` will be executed.

Internals

PostgreSQL

When building the command, `PostgreSqlCommandExecutionToken.DereferenceCursors` is set to true. This causes the data sources' execute functions to use cursor dereferencing semantics.

Cursors cannot be dereferenced outside of a transaction. Chain will automatically create a transacton if one doesn't exist.

The `CommandTimeout` is misleading, as it applied twice: once to the function call and once to the call that dereferences the cursors. This effectively doubles the amount of time that a command can theoretically take to execute.

Roadmap

Non-Query Materializers

There are two Non-Query materializers. `AsNonQuery` has no return value, while `AsRowsAffected` returns the number of rows affected.

Limitations

The rows affected count is not necessarily accurate. For example, if a stored procedure uses `SET NOCOUNT ON` then there will be nothing to report.

SQL Generation

These materializers respond with NoColumns when participating in SQL generation.

Internals

These materialziers use DbCommand.ExecuteNonQuery.

Roadmap

Scalar Materializers

Scalar materializers are avaialble for common .NET types such as string, numerics, date/time, etc. Nullable versions are avaialable.

Options

The scalar materializers accepts an optional column name. This may be used by the command builder when generating a SELECT clause.

SQL Generation

As mentioned above, this materailizer may request a specific column. If no column in indicated, then AutoSelectDesiredColumns is returned to the command builder.

Internals

All scalar materializes use DbCommand.ExecuteScalar. This means that only the first column of the first row is evaluated. Everything else is silently ignored.

Roadmap

List Materializers

List materializers are avaialble for common .NET types such as string, numerics, date/time, etc.

Options

The list materializers accepts an optional column name. This may be used by the command builder when generating a SELECT clause.

- DiscardNulls: This flag will discard null values when populating the list.
- IgnoreExtraColumns: Normally an error is raised if multiple columns are returned. This flag will ignore them instead.
- FlattenExtraColumns: If multiple columns are returned, this flag will cause all columns to be included in the resulting list.

FlattenExtraColumns operates left to right, then top to bottom. For example:

```
Resultset:
    [1, 2, 3]
    [4, 5, 6]
List:
    1, 2, 3, 4, 5, 6
```

SQL Generation

As mentioned above, this materailizer may request a specific column. If no column in indicated, then AutoSelectDesiredColumns is returned to the command builder.

Internals

Roadmap

You can pass a columnName parameter to list based materializers. This parameter can be used for SQL generation, but is ignored when parsing the result set. As a result, we can't do interesting things like only read one column from a stored procedure that returns 3 columns. Task #24.

Object/Collection Materializers

- `.ToObect<TObject>`
- `.ToCollection<TObject>`
- `.ToCollection<TObject, TCollection>`
- `.ToImmutableArray<TObject>`
- `.ToImmutableList<TObject>`

Options

The `ToObject` materializer supports the `RowOptions` enumeration. The `ToCollection` materializer supports the `CollectionOptions` enumeration.

The `ToCollection` materializer returns a `List<TCollection>` by default. You can override the collection type with any `ICollection<TObject>` so long as it isn't read-only.

Capabilities

Object/Collection materializers can populate non-public types.

Object/Collection materializers honor the Column attribute, which changes which query result set column the property is mapped to.

Object/Collection materializers honor the `NotMapped` attribute, which prevent the property from being mapped.

Object/Collection materializers honor the `Decompose` attribute. This allows properties on the child object when they match columns in the query result set.

If the desried object implementes `IChangeTracking`, then `AcceptChanges()` will be called automatically.

Non-default Constructors

If you use the `InferConstructor` option or the `WithConstructor` method, the behavior changes. Instead of setting properties, the indicated non-default constructor will be called. (With `InferConstructor`, there can only be one non-default constructor.)

The `Decompose` attribute doesn't apply when using a non-default constructor.

SQL Generation

As per above, mapped properties on the object (and child properties on a decomposed property) will be requested.

In a non-default constructor is chosen, then SQL will only be generated for the columns that match parameter names in the constructor.

Limitations

Object/Collection materializers require that the `Decompose` attribute be applied correctly. The materializer needs to walk the entire object graph, and if there are any cycles represented by decomposed properties then a stack overflow exception will occur.

Object/Collection materializers can only populate public properties. It cannot set fields or non-public properties (unless you use a constructor).

Internals

Object/Collection materializers use reflection to instantiate and populate the object.

Roadmap

Compiled Object/Collection Materializers

Compiled materializers require the `Tortuga.Chain.CompiledMaterializers` package.

- `.Compiled.ToObect<TObject>`
- `.Compiled.ToCollection<TObject>`
- `.Compiled.ToCollection<TCollection, TObject>`

Options

The `ToObject` materializer supports the `RowOptions` enumeration.

The `ToCollection` materializer returns a `List<TCollection>` by default. You can override the collection type with any `ICollection<TObject>` so long as it isn't read-only.

Capabilities

Object/Collection materializers honor the Column attribute, which changes which query result set column the property is mapped to.

Object/Collection materializers honor the `NotMapped` attribute, which prevent the property from being mapped.

Object/Collection materializers honor the `Decompose` attribute. This allows properties on the child object when they match columns in the query result set.

If the desried object implementes `IChangeTracking`, then `AcceptChanges()` will be called automatically.

SQL Generation

See Object/Collection materializers.

Limitations

Object/Collection materializers require that the `Decompose` attribute be applied correctly. The materializer needs to walk the entire object graph, and if there are any cycles represented by decomposed properties then a stack overflow exception will occur.

Object/Collection materializers can only populate public properties. It cannot set fields or non-public properties.

Compiled Object/Collection materializers only support public types. Because it does not use reflection, it cannot sidestep the visibility restrictions like the non-compiled versions can.

Compiled Object/Collection materializers do not support generic types (other than Nullable). This is not a design limitation and will be fixed in a later version. https://github.com/docevaad/Chain/issues/64

Compiled Object/Collection materializers do not support nested types. This is not a design limitation and will be fixed in a later version. https://github.com/docevaad/Chain/issues/63

Compiled Object/Collection materializers are less tolerant of column/property type mis-matches. For example, if you database column is a `Long`, you can't use a property of type `Nullable<int>`.

Internals

Compiled Object/Collection materializers use CS Script to generate the population methods.

Roadmap

Eventually we would like to support non-default constructors so that we can populate immutable objects.

DataRow, DataTable, and DataSet Materializers

These materializers support the legacy DataRow, DataTable, and DataSet types. They are slower than the Row, Table, and TableSet equivalents, but have the advantage of being bindable to data grids in Windows Forms and WebForms.

Options

- `.ToDataRow` supports the `RowOptions` enumeration.
- `.ToDataSet` expects a list of table names.

SQL Generation

These materializers do not request any columns. See the associated command builder for the behavior.

Row, Table, and TableSet Materializers

These are modern equivalents to DataRow, DataTable, and DataSet.

Options

- `.ToRow` supports the `RowOptions` enumeration.
- `.ToTableSet` expects a list of table names.

SQL Generation

These materializers do not request any columns. See the associated command builder for the behavior.

Internals

A Row is implemented as an immutable dictionary. Likewise, Table and TableSet are immutable.

Roadmap

These objects, while faster than the legacy versions, can probably be improved even more if we switch to an array based internal implementation. We're looking for improvements both in performance and, more importantly, memory consumption.

Caching Appenders

- `.Cache(...)`
- `.CacheAllItems(...)`
- `.InvalidateCache(...)`
- `.ReadOrCache(...)`

Cache Keys

Cache keys can be provided as a string or a function. In the latter case, the function accepts the result of the chain and returns a string.

`.CacheAllItems(...)` is a special case. It only operates on lists of objects and generates a cache key for each object seperately.

Example of a Caching Repository

In this example you can see the interplay between caching individual records and caching collections using `CacheAllItems`.

```
public class EmployeeCachingRepository
{

    private const string TableName = "HR.Employee";
    private const string AllCacheKey = "HR.Employee ALL";

    public IClass1DataSource Source { get; private set; }
    public CachePolicy Policy { get; private set; }

    public EmployeeCachingRepository(IClass1DataSource source, CachePolicy policy = null)
    {
        Source = source;
        Policy = policy;
    }

    protected string CacheKey(int id)
    {
        return $"HR.Employee EmployeeKey={id}";
    }

    protected string CacheKey(Employee entity)
    {
        return CacheKey(entity.EmployeeKey.Value);
    }

    public Employee Get(int id)
    {
        return Source.GetByKey(TableName, id).ToObject<Employee>().ReadOrCache(CacheKey(id), policy: Policy).Execute();
    }

    public IList<Employee> GetAll()
    {
        return Source.From(TableName).ToCollection<Employee>().CacheAllItems((Employee x) => CacheKey(x), policy: Policy).ReadOrCache(AllCacheKey, policy
: Policy).Execute();
    }

    public Employee Insert(Employee entity)
    {
        return Source.Insert(TableName, entity).ToObject<Employee>().InvalidateCache(AllCacheKey).Cache((Employee x) => CacheKey(x), policy: Policy).Exec
ute();
    }

    public Employee Update(Employee entity)
    {
        return Source.Update(TableName, entity).ToObject<Employee>().Cache(CacheKey(entity)).InvalidateCache(AllCacheKey).Execute();
    }

    public void Delete(int id)
    {
        Source.DeleteByKey(TableName, id).InvalidateCache(CacheKey(id)).InvalidateCache(AllCacheKey).Execute();
    }
}
```

Internals

The caching appenders use the data source to access a cache. Currently that data source is hard-coded to use .NET's built-in `MemoryCache`.

Roadmap

The current plan is to allow the caching framework to be swapped out.

Tracing Appenders

The tracing appenders are usually used for debugging purposes. For logging, the events on DataSource are more useful.

- `.WithTracing`
- `.WithTracingToDebug`
- `.WithTracingToConsole`

Arguments

The generic `.WithTracing` appender accepts a `TextWriter`, while the others

Internals

These appenders listen for the OnCommandBuilt event.

Roadmap

Change Notification Appenders

SQL Server Only

The `WithChangeNotification` appender is implemented on top of SQL Dependency. It is essential that you read and understand the requirements for using SQL Dependency before attempting to use this appender.

https://msdn.microsoft.com/en-us/library/system.data.sqlclient.sqldependency.aspx

Arguments

This appender accepts an `OnChangeEventHandler` delegate, which will be fired one time if the database detects an associated change. Once it fires, the event will have to be reattached.

Internals

Roadmap

Automatic cache invalidation is planned for the future. https://github.com/docevaad/Chain/issues/23

Audit Rules

Audit rules allow you to drastically reduce the amount of boilerplate code needed for common scenarios such as verifying a record is valid before saving it, setting the created by or updated by column, and honoring soft deletes.

Note that all of these rules are applied at the data source level. You cannot selectively apply them to individual tables, though you can use multiple data sources with different rules.

Validation

Chain supports the following validation interfaces:

- `IDataErrorInfo`
- `INotifyDataErrorInfo`
- `IValidatable` (From Tortuga Anchor)

Validation will only occur if the object used for the insert/update operation implements the indicated interface.

Example:

```
dataSource = dataSource.WithRules(new ValidateWithDataErrorInfo (OperationType.InsertOrUpdate));
```

Triggering Validation

Depending on your object model, you may need to manually trigger validation before IDataErrorInfo or INotifyDataErrorInfo report an error condition.

```
dataSource = dataSource.WithRules(new ValidateWithDataErrorInfo<ModelBase>(OperationType.InsertOrUpdate, x => x.Validate()));
```

Custom Validation

If your object model has its own validation interface, you can subclass `ValidationRule`.

Created By/Updated By Support

Chain can automatically set fields such as “CreatedByKey” and “UpdatedByKey”. There are two steps necessary to do this. First, you need to create a data source with the appropriate rules. As with a normal data source, this should be cached at the application level.

```
dataSource = dataSource.WithRules(
    new UserDataRule("CreatedByKey", "UserKey", OperationType.Insert),
    new UserDataRule("UpdatedByKey", "UserKey", OperationType.InsertOrUpdate));
```

When a request is initiated, you then create a contextual data source with the user object. There are no restrictions on what this object looks like, so long as it has the columns indicated by your audit rules.

```
myDS = dataSource.WithUser(currentUser);
```

When using the contextual data source, the indicated rules will be automatically applied to insert, and update operations. For example:

```
myDS.Update("Customer", customer).Execute();
```

This will automatically apply the user’s `UserKey` value to the `UpdatedByKey` column when performing the update operation. This replaces any value on `customer.UpdatedByKey` and will work even if `customer.UpdatedByKey` is marked as `[IgnoreOnUpdate]`.

Created Date/Updated Date Support

To ensure that `CreatedDate` and `UpdatedDate` are correctly set without using constraints and triggers, you can use `DateTimeRule` or `DateTimeOffsetRule`.

```
dataSource = dataSource.WithRules(
    new DateTimeRule("CreatedDate", DateTimeKind.Local, OperationType.Insert),
    new DateTimeRule("UpdatedDate", DateTimeKind.Local, OperationType.InsertOrUpdate));
```

These rules do not require a context data source.

Soft Delete Support

Soft delete support is enabled by add a `SoftDeleteRule`. This rule can apply to delete and/or select operations. Usually the `SoftDeleteRule` will be combined for rules that set a deleted by and/or deleted date column. For example:

```
dataSource = dataSource.WithRules(
    new SoftDeleteRule("DeletedFlag", 1, OperationType.SelectOrDelete),
    new UserDataRule("DeletedByKey", "UserKey", OperationType.Delete),
    new DateTimeOffsetRule("DeletedDate", OperationType.Delete));
```

When soft deletes are in effect, calling `dataSource.Delete` will automatically switch from a hard to a soft delete if the correct column exists.

Likewise, the `dataSource.From` operation will automatically rewrite the WHERE clause to filter out deleted records.

Transactions

A transactional data source inherits the rules and user value from the data source used to create it. If necessary, you can chain these operations together. For example:

```
using (var trans = dataSource.WithUser(currentUser).BeginTransaction())
```

Dependency Injection

If your DI framework supports it, you should create the contextual data source using the `dataSource.WithUser` function at the beginning of the request. You can then inject that data source instead of the application-wide version to ensure that audit rules are enforced.

Restricted Columns

Chain's restricted column support allows you to limit read/write access to particular columns based on the user. These restrictions can be applied globally or to specific tables.

You setup a restricted column the same way you would any other audit rule. Here is an example that prevents users from granting themselves admin privileges:

```
ExceptWhenPredicate IsAdminCheck = user => ((UserToken)user).IsAdmin;

DataSource = DataSource.WithRules(new RestrictColumn("Users", "IsAdmin", OperationTypes.Insert |  OperationTypes.Update, IsAdminCheck));
```

Whenever an operation is performed against the `Users` table, the `IsAdminCheck` function will be executed. If the check fails, the SQL generator will skip the IsAdmin column while performing insert/update operations.

There are cases where you may need to universally block access to a column. For example, say you find that the CreditCardNumber is being exposed via multiple tables and views. While you still want your payment resolution managers to have access to it, most users shouldn't see it. We can enable using this syntax:

```
ExceptWhenPredicate IsManger = user => ((UserToken)user).HasRoles("PaymentResolutionManager");

DataSource = DataSource.WithRules(new RestrictColumn("CreditCardNumber", OperationTypes.Select, IsManger));
```

An interesting effect of this rule is that it only blocks reads, not writes. So non-managers can update a customer's credit card number even though they can't actually see it themselves after pressing the save button.

Usage Notes

Once you setup a restricted column rule, you need to ensure that all data access is performed within the context of a user. For example:

```
var ds = DataSource.WithUser(currentUser);
//perform operations with ds
```

This can be combined with a transaction:

```
using (var ds = DataSource.WithUser(currentUser).BeginTransaction())
{
    //perform operations with ds
    ds.Commit();
}
```

Implementation

Restricted columns are built on top of the Audit Rules functionality and participate in the SQL generation process. This means that it is possible to bypass the restriction using stored procedures.

Audit rules and restricted columns are not intended to be mixed. For example, if you have an audit rule for setting the UpdatedBy/LastUpdatedDate columns, do not put a restricted column rule on the same columns.

Transactions

Basic transactions are created by calling `dataSource.BeginTransaction()`. This creates a transactional data source which can then be used just like a normal database.

Transactional data sources contain an open connection and transaction object, so they should be managed with a `using` statement just like any other limited resource.

```
using (var trans = dataSource.BeginTransaction())
{
    var newKey = trans.Insert("Customer", customer).AsInt32().Execute();
    address.CustomerKey = newKey;
    trans.Insert("CustomerAddress", address).Execute();
    trans.Commit();
}
```

Attaching to Active Connections and Transactions

There may be times when you need to attach a data source to an existing connection or transaction. This usually happens when mixing Chain with another ORM.

To do this, simply call `CreateOpenDataSource` on a normal data source and pass in the connection object. Optionally, you can also include a transaction object.

```
var openDataSource = dataSource.CreateOpenDataSource( activeConnection, activeTransaction);
```

An open data source does not "own" the connection or transaction associated with it. This means that the open data source cannot close the connection or commit the transaction itself.