Trysil – Delphi ORM

https://github.com/davidlastrucci/trysil/

English version

Trysil – Operation ORM (World War II)

https://codenames.info/operation/orm/

‘ORM’ was a British operation by the Special Operations Executive to establish a reception base centred on Trysil in the eastern part of German-occupied Norway (3 March/9 May 1945).

The initial two-man party of B. Hansen and B. Sætre was delivered by parachute on 3 March and established radio contact with the UK. A four-man party with the base leader, Captain Aasen, then arrived overland from neutral Sweden. Supplies were initially limited, and a strong German presence meant that much of the work was undertaken from across the Swedish border. Co-operation with the Milorg military resistance organisation was good, and more supplies and men arrived from Stockholm.

At the time of the German surrender, the 'Orm' leadership and 75 men crossed the border, and on 9 May they accepted the surrender of the local German commandant.

Summary

Overview

Using an ORM (Object–Relational Mapping) for application development means talking to the database using object language and not telling long, boring SQL sentences.

Benefits

An ORM is basically a layer of software (a library) that allows you not to write code like this:

```
FQuery.SQL.Text :=
  'SELECT I.ID AS InvoiceID, I.Number, ' +
  'C.ID AS CustomerID, C.Name AS CustomerName, ' +
  'U.ID AS CountryID, U.Name AS CountryName ' +
  'FROM Invoices AS I ' +
  'INNER JOIN Customers AS C ON C.ID = I.CustomerID ' +
  'INNER JOIN Countries AS U ON U.ID = C.CountryID ' +
  'WHERE I.ID = :InvoiceID';
FQuery.ParamByName('InvoiceID').AsInteger := 1;
FQuery.Open;

ShowMessage(
  Format('Invoice No: %d, Customer: %s, Country: %s', [
    FQuery.FieldByName('Number').AsInteger,
    FQuery.FieldByName('CustomerName').AsString,
    FQuery.FieldByName('CountryName').AsString]));
```

Instead of code like this:

```
LInvoice := FContext.Get<TInvoice>(1);

ShowMessage(
  Format('Invoice No: %d, Customer: %s, Country: %s', [
    LInvoice.Number,
    LInvoice.Customer.Name,
    LInvoice.Customer.Country.Name]));
```

Introduction

Delphi supported version

- Delphi 10.3 – Rio (260)
- Delphi 10.4 – Sydney (270)
- Delphi 11 – Alexandria (280)
- Delphi 12 – Athens (290)

Data Model

Data model is a set of classes that can be read and saved to a database. Data model classes are called entities.

Follow the example of the TPerson entity:

```
type

{ TPerson }

  TPerson = class
  strict private
    FFirstname: String;
    FLastname: String;
    FEmail: String;
  public
    property Firstname: String
      read FFirstname write FFirstname;
    property Lastname: String read FLastname write FLastname;
    property Email: String read FEmail write FEmail;
  end;
```

PODO (Plain Old Delphi Objects)

Entities, in Trysil, are classic objects, not special objects: entities can be inherited directly from TObject.

The term PODO derives from POJO (Plain Old Java Object) which in 2000 has been coined by Martin Fowler, Rebecca Parsons and Josh MacKenziewith this justification:

"*We wondered why people were so against using regular objects in their systems and concluded that it was because simple objects lacked a fancy name. So we gave them one, and it's caught on very nicely.*".

Setup

Once you have cloned the Trysil repositoty on GitHub, you need to perform a few steps to be able to use the ORM.

Git Bash

To simplify we assume that we are cloning the repository in the C:\ folder

```
git clone https://github.com/davidlastrucci/Trysil.git
```

Once the clone is finished, the C:\Trysil folder will contain all the Trysil sources in the GitHub repository.

Build Lib

Open "Trysil.groupproj" in C:\Trysil\Packages\???[1] folder and Build in all necessary configurations and platforms.

Warning: If you are using the Community edition or the Professional edition of Delphi, you will not be able to build the Trysil.SqlServer???.dproj project and you will not be able to use connections for Microsoft SQL Server.

C:\Trysil\Lib\???\$(Platform)\$(Config) directory now contains all Trysil BPLs, DCPs and DCUs.

Environment Variables

In "Tools -> Options -> Environment Variables" add a new User Variable:

```
Variable name: Trysil
Variable value: C:\Trysil\Lib\???
```

[1] ??? represents the Delphi version you are using. See the paragraph "Supported Delphi Versions"

Expert

Open the Trysil.Expert???.dproj project present in the C:\Trysil\Trysil.Expert folder, and compile it.

Close Delphi.

Run regedit.exe and navigate to the folder

```
HKCU\SOFTWARE\Embarcadero\BDS\???\Expert
```

Add a new string value:

```
Name: Trysil
Value: C:\Trysil\Trysil.Expert\Win32\Trysil.Expert???.dll
```

Start Delphi again.

Trysil is now present on the Splash Screen and in the Menu.

New Delphi Project

Create a new Delphi project and, in "Project -> Options -> Building -> Delphi Compiler" select "All configurations - All Platforms" and in the "Search Path" write:

```
$(Trysil)\$(Platform)\$(Config)
```

The new project is now ready to use Trysil.

Database connection

Supported databases

Database	Edizione di Delphi	
	Community / Professional	Enterprise / Architect
FirebirdSQL	✔ [2]	✔
PostgreSQL		
SQLite	✔	
SQL Server		

TTConnection

Trysil.Data.TTConnection

TTConnection is the abstract connection to the database; from TTConnection they inherit, directly or indirectly, all other types of connections.

The TTConnection constructor requires the name of a connection that must be previously registered via RegisterConnection.

```
constructor Create(const AConnectionName: String);
```

RegisterConnection is a method introduced by descendants of TTConnection.

[2] Available only for connections to localhost (limit imposed by FireDAC)

TTFirebirdSQLConnection

Trysil.Data.FireDAC.FirebirdSQL.TTFirebirdSQLConnection

Connection to a FirebirdSQL database. You can register a connection in the following ways:

```
class procedure RegisterConnection(
 const AConnectionName: String;
 const AServer: String;
 const ADatabaseName: String);

class procedure RegisterConnection(
 const AConnectionName: String;
 const AServer: String;
 const AUsername: String;
 const APassword: String;
 const ADatabaseName: String);

class procedure RegisterConnection(
 const AConnectionName: String;
 const AParameters: TStrings);
```

TTPostgreSQLConnection

Trysil.Data.FireDAC.PostgreSQL.TTPostgreSQLConnection

Connection to a PostgreSQL database. You can register a connection in the following ways:

```
class procedure RegisterConnection(
  const AConnectionName: String;
  const AServer: String;
  const AUsername: String;
  const APassword: String;
  const ADatabaseName: String);

class procedure RegisterConnection(
  const AConnectionName: String;
  const AServer: String;
  const APort: Integer;
  const AUsername: String;
  const APassword: String;
  const ADatabaseName: String);

class procedure RegisterConnection(
  const AConnectionName: String;
  const AParameters: TStrings);
```

TTSQLiteConnection

Trysil.Data.FireDAC.SQLite.TTSQLiteConnection

Connection to a SQLite database. You can register a connection in the following ways:

```
 class procedure RegisterConnection(
  const AConnectionName: String;
  const ADatabaseName: String);

class procedure RegisterConnection(
  const AConnectionName: String;
  const AUsername: String;
  const APassword: String;
  const ADatabaseName: String);

class procedure RegisterConnection(
  const AConnectionName: String;
  const AParameters: TStrings);
```

TTSqlServerConnection

Trysil.Data.FireDAC.SqlServer.TTSqlServerConnection

Connection to a SQL Server database. You can register a connection in the following ways:

```
class procedure RegisterConnection(
  const AConnectionName: String;
  const AServer: String;
  const ADatabaseName: String);

class procedure RegisterConnection(
  const AConnectionName: String;
  const AServer: String;
  const AUsername: String;
  const APassword: String;
  const ADatabaseName: String);

class procedure RegisterConnection(
  const AConnectionName: String;
  const AParameters: TStrings);
```

TTContext

Trysil.Context.TTContext

TTContext is the class that allows you to communicate with the database.

Inside TTContext, TTProvider is used for reading operations and TTResolver for writing operations.

TTProvider and TTResolver should not be used directly.

```
...
strict private
  FConnection: TTConnection;
  FContext: TTContext;
...
  TTSqlServerConnection.RegisterConnection(
    'MyConnection', '127.0.0.1', 'TestDB');

  FConnection := TTSqlServerConnection.Create(
   'MyConnection');

  FContext := TTContext.Create(FConnection);
...
```

Connection pool

Pooling reduces the number of times you need to open new connections by managing and maintaining a set of them for each configuration.

When the application opens a connection, the pool checks for an available one and if available returns it to the caller instead of creating a new one.

When the application closes a connection, the pool returns it to the available set instead of actually destroying it.

Once returned to the pool, the connection can be used again on the next open.

TTFireDACConnectionPool

Trysil.Data.FireDAC.ConnectionPool.TTFireDACConnectionPool

You can turn Trysil connection pooling on or off like this:

```
TTFireDACConnectionPool.Instance.Config.Enabled := True;
```

Identity Map

Trysil.IdentityMap.TTIdentityMap

TTIdentityMap is an entity cache powered by TTContext.

If the requested entity has already been read from the database, through the use of TTIdentityMap, the same instance of the previously read entity is returned.

However, the use of TTIdentityMap guarantees updated data.

TTContext, by default, uses identity map which can be disabled during its creation.

Below are the constructors of TTContext:

```
constructor Create(const AConnection: TTConnection);

constructor Create(
  const AConnection: TTConnection;
  const AUseIdentityMap: Boolean);
```

Metadata

Insert, Update and Delete operations on the database use parameters, both to ensure security and to ensure efficiency. For this reason Trysil needs metadata.

TTContext.GetMetadata<T>()

Through the GetMetadata<T> method of TTContext we can access the metadata of the entities:

```
var
  LTableMetadata: TTTableMetadata;
begin
  LTableMetadata := FContext.GetMetadata<TPerson>();
  ...
```

TTTableMetadata

Trysil.Metadata.TTTableMetadata

Contains the metadata of the table linked to an entity:

- **TableName** the name of the database table
- **PrimaryKey** the name of the primary key column of the database table
- **Columns** the list of database columns

TTColumnMetadata

Trysil.Metadata.TTColumnMetadata

Contains the metadata of the table column linked to an entity:

- **ColumnName** the name of the database table column
- **DataType** the type (TFieldType) of the database table column
- **DataSize** the size of the database table column

Mapping

The mapping is used to instruct Trysil to interact with the database: how to read, insert, update and delete data using an entity.

Attributes

TTableAttribute

Trysil.Attributes.TTableAttribute

It applies to the class and defines the name of the database table:

```
[TTable('Persons')]
TPerson = class
```

TSequenceAttribute

Trysil.Attributes.TSequenceAttribute

It applies to the class and defines the database sequence name that will be used for the entity's primary key[3]:

```
[TSequence('PersonsID')]
TPerson = class
```

[3] SQLite doesn't provide anything like sequences. The ROWID column is used instead (a concept very similar to MAX + 1). For this reason, the use of SQLite in a multi-user environment is not recommended.

TWhereClauseAttribute

Trysil.Attributes.TWhereClauseAttribute

You can decorate the class with the TWhereClauseAttribute attribute to filter the database table data on which the entity will work.

For example, let's add the PersonType column to our Persons table and define that it can contain the values 1 (manager) and 2 (employee).

We can define our TManager and TEmployee entities like this:

```
type

{ TManager }

  [TWhereClause('PersonType = 1')]
  TManager = class(TPerson)
  end;

{ TEmployee }

  [TWhereClause('PersonType = 2')]
  TEmployee = class(TPerson)
  end;
```

TRelationAttribute

Trysil.Attributes.TRelationAttribute

It applies to the class and defines the entity's relationship with another database table:

```
[TRelation('Companies', 'EmployeeID', False)]
TEmployee = class
...
```

TRelationAttribute requires three parameters:

- **TableName** the name of the related table
- **ColumnName** the name of the related column
- **IsCascade** indicates whether the relationship is cascade or not

TPrimaryKeyAttribute

Trysil.Attributes.TPrimaryKeyAttribute

It applies to a field of the entity class and defines the primary key of the database table:

```
[TPrimaryKey]
```

TColumnAttribute

Trysil.Attributes.TColumnAttribute

It applies to a field of the entity class and defines the name of the column on the database table:

```
[TColumn('Firstname')]
```

TDetailColumnAttribute

Trysil.Attributes.TDetailColumnAttribute

It applies to a field of the entity class (usually to a column of type TTLazyList<T>) and defines an entity detail (master/detail):

```
TCompany = class
strict private
  ...
  [TDetailColumn('ID', 'CompanyID')]
  FEmployees: TTLazyList<TEmployee>;
  ...
```

TVersionColumnAttribute

Trysil.Attributes.TVersionColumnAttribute

Trysil, to manage data concurrency, uses a version type Column. The version column is an Int32 which is incremented at each update (Update).

The TVersionColumnAttribute attribute applies to the field of the class that represents the version of the record:

```
[TVersionColumn]
```

For columns of type version on use a field of type TTVersion.

"Mapped" entity sample

Below is the TPerson entity "mapped" with Trysil attributes:

```
type

{ TPerson }

  [TTable('Persons')]
  [TSequence('PersonsID')]
  TPerson = class
  strict private
    [TPrimaryKey]
    [TColumn('ID')]
    FID: TTPrimaryKey;

    [TColumn('Firstname')]
    FFirstname: String;

    [TColumn('Lastname')]
    FLastname: String;

    [TColumn('Email')]
    FEmail: String;

    [TVersionColumn]
    [TColumn('VersionID')]
    FVersionID: TTVersion;
  public
    property ID: TTPrimaryKey read FID;
    property Firstname: String
      read FFirstname write FFirstname;
    property Lastname: String read FLastname write FLastname;
    property Email: String read FEmail write FEmail;
    property VersionID: TTVersion read FVersionID;
  end;
```

TTPrimaryKey

Trysil.Types.TTPrimaryKey

TTPrimaryKey is the type to use for the primary key of entities. It is an alias of Int32.

TTVersion

Trysil.Types.TTVersion

TTVersion is the type to use for the entity version column and it is also an alias of Int32.

TTNullable<T>

Trysil.Types.TTNullable<T>

Databases support NULL columns; to be able to manage this type of columns, TTNullable<T> types have been introduced in Trysil.

TTNullable<T> is a record that implements a series of "class operators" in order to allow, for example, the assignment of a nullable to its corresponding type and vice versa.

```
var
  LNullable: TTNullable<String>;
  LString: String;
begin
  LNullable := 'David';
  LString := LNullable;
  ...
```

Lazy loading

Through the "lazy loading" mechanism, the entities are read from the database only when we need them. Let's analyze one of the first examples of this document and see what happens behind the scenes:

```
LInvoice := FContext.Get<TInvoice>(1);

ShowMessage(
  Format('Invoice No: %d, Customer: %s, Country: %s', [
    LInvoice.Number,
    LInvoice.Customer.Name,
    LInvoice.Customer.Country.Name]));
```

- *FContext.Get<TInvoice>(1)* instruction reads the invoice with ID equal to 1 from the database
- *LInvoice.Customer* instruction reads from the customer database linked to the previously read invoice
- *LInvoice.Customer.Country* instruction reads from the database the country connected to the previously read customer

TTLazy<T>

Trysil.Lazy.TTLazy<T>

TTLazy<T> is the type of column that represents a "foreign entity":

```
type

{ TEmployee }

  TEmployee = class
  strict private
    ...
    [TColumn('CompanyID')]
    FCompany: TTLazy<Company>;
    ...
    function GetCompany: TCompany;
    procedure SetCompany(const AValue: TCompany);
  public
    ...
    property Company: TCompany
      read GetCompany write SetCompany;
    ...
  end;
```

Implementation

```
function TEmployee.GetCompany: TCompany;
begin
  result := FCompany.Entity;
end;

procedure TEmployee.SetCompany(const AValue: TCompany);
begin
  FCompany.Entity := AValue;
end;
```

TTLazyList<T>

Trysil.Lazy.TTLazyList<T>

TTLazyList<T> is the type of column that represents an "entity detail":

```
type

{ TCompany }

  TCompany = class
  strict private
    ...
    [TDetailColumn('ID', 'CompanyID')]
    FEmployees: TTLazyList<TEmployee>;
    ...
    function GetEmployees: TTList<TEmployee>;
  public
    ...
    property Employees: TTList<TEmployee> read GetEmployees;
    ...
  end;
```

Implementation

```
function TEmployee. GetEmployees: TTList<TEmployee>;
begin
  result := FEmployees.List;
end;
```

Constructors

As already mentioned, entities can be of type PODO and therefore, can be inherited directly from TObject.

The only constraint we have is the manufacturer; we have to choose one among:

- Default constructor, the one without parameters
- Constructor with a single parameter of type TTContext

Events

Events are part of the model. For each entity it is possible to define the events before (before) and after (after) insertion (INSERT), updating (UPDATE) and deletion (DELETE).

Events can be defined in two ways:

- By creating a class that inherits from TTEvent<T>
- By defining the methods directly within the entity class

TTEvent<T>

Trysil.Events.TTEvent<T>

TTEvent<T> defines two virtual methods that can be overridden:

- DoBefore
- DoAfter

```
type

{ TPersonInsertEvent }

  TPersonInsertEvent = class(TTEvent<TPerson>)
  public
    procedure DoBefore; override;
    procedure DoAfter; override;
  end;

{ TPersonUpdateEvent }

  TPersonUpdateEvent = class(TTEvent<TPerson>)
  public
    procedure DoBefore; override;
    procedure DoAfter; override;
  end;
```

```
{ TPersonDeleteEvent }

   TPersonDeleteEvent = class(TTEvent<TPerson>)
   public
     procedure DoBefore; override;
     procedure DoAfter; override;
   end;
```

TInsertEventAttribute

Trysil.Events.Attributes.TInsertEventAttribute

The TInsertEventAttribute attribute applies to the class and defines the event to execute when inserting a new entity:

```
[TInsertEvent(TPersonInsertEvent)]
TPerson = class
...
```

TUpdateEventAttribute

Trysil.Events.Attributes.TUpdateEventAttribute

The TUpdateEventAttribute attribute applies to the class and defines the event to execute when updating an entity:

```
[TUpdateEvent(TPersonUpdateEvent)]
TPerson = class
...
```

TDeleteEventAttribute

Trysil.Events.Attributes.TDeleteEventAttribute

The TDeleteEventAttribute attribute applies to the class and defines the event to execute when deleting an entity:

```
[TDeleteEvent(TPersonDeleteEvent)]
TPerson = class
...
```

Methods of entity class

TBeforeInsertEventAttribute

Trysil.Events.Attributes.TBeforeInsertEventAttribute

The TBeforeInsertEventAttribute attribute applies to the method to execute before inserting the entity:

```
{$RTTI EXPLICIT METHODS([vcPrivate..vcPublished])}

TPerson = class
strict private
  [TBeforeInsertEvent]
  procedure BeforeInsert();
  ...
```

TAfterInsertEventAttribute

Trysil.Events.Attributes.TAfterInsertEventAttribute

The TAfterInsertEventAttribute attribute applies to the method to execute after the entity is inserted:

```
{$RTTI EXPLICIT METHODS([vcPrivate..vcPublished])}

TPerson = class
strict private
  [TAfterInsertEvent]
  procedure AfterInsert();
  ...
```

TBeforeUpdateEventAttribute

Trysil.Events.Attributes.TBeforeUpdateEventAttribute

The TBeforeUpdateEventAttribute attribute applies to the method to run before the entity is updated:

```
{$RTTI EXPLICIT METHODS([vcPrivate..vcPublished])}

TPerson = class
strict private
  [TBeforeUpdateEvent]
  procedure BeforeUpdate();
  ...
```

TAfterUpdateEventAttribute

Trysil.Events.Attributes.TAfterUpdateEventAttribute

The TAfterUpdateEventAttribute attribute applies to the method to run after the entity is updated:

```
{$RTTI EXPLICIT METHODS([vcPrivate..vcPublished])}

TPerson = class
strict private
  [TAfterUpdateEvent]
  procedure AfterUpdate();
  ...
```

TBeforeDeleteEventAttribute

Trysil.Events.Attributes.TBeforeDeleteEventAttribute

The TBeforeDeleteEventAttribute attribute applies to the method to execute before deleting the entity:

```
{$RTTI EXPLICIT METHODS([vcPrivate..vcPublished])}

TPerson = class
strict private
  [TBeforeDeleteEvent]
  procedure BeforeDelete();
  ...
```

TAfterDeleteEventAttribute

Trysil.Events.Attributes.TAfterDeleteEventAttribute

The TAfterDeleteEventAttribute attribute applies to the method to execute after the entity is deleted:

```
{$RTTI EXPLICIT METHODS([vcPrivate..vcPublished])}

TPerson = class
strict private
  [TAfterDeleteEvent]
  procedure AfterDelete();
  ...
```

{$RTTI EXPLICIT...}

By default, Delphi does not generate RTTI information for non-public methods.

It is necessary to add the {$RTTI EXPLICIT...} directive to the entity class because the methods have been defined as private, and, in this situation, Trysil would not be able to know which methods to invoke when events occur

We therefore have two possibilities:

- Add the {RTTI EXPLICIT...} directive
- Define methods of type public

Personally, I prefer to instruct Delphi to generate RTTI information for private methods rather than making them public to the entire application.

Data validation

Trysil provides simple and straightforward ways to validate entities before they are persisted to the database.

By adding specific attributes to the entity's fields, you can ensure that the entity is always saved in the database in a correct state.

Attributes

TDisplayNameAttribute

Trysil.Validation.Attributes.TDisplayNameAttribute

The TDisplayNameAttribute attribute defines the name to use in case of a validation error:

```
[TDisplayName('Cognome')]
FLastname: String;
```

TRequiredAttribute

Trysil.Validation.Attributes.TRequiredAttribute

The TRequiredAttribute attribute defines that that column is necessary, mandatory, and cannot be left empty:

```
[Required]
FLastname: String;
```

TMinLengthAttribute

Trysil.Validation.Attributes.TMinLengthAttribute

The TMinLengthAttribute attribute defines the minimum length of the column value:

```
[TMinLength(1)]
FLastname: String;
```

TMaxLengthAttribute

Trysil.Validation.Attributes.TMaxLengthAttribute

The TMaxLengthAttribute attribute defines the maximum length of the column value:

```
[TMaxLength(100)]
FLastname: String;
```

TMinValueAttibute

Trysil.Validation.Attributes.TMinValueAttribute

The TMinValueAttribute attribute defines the minimum value for the column:

```
[TMinValue(1)]
FAge: Integer;
```

TMaxValueAttribute

Trysil.Validation.Attributes.TMaxValueAttribute

The TMaxValueAttribute attribute defines the maximum value for the column:

```
[TMaxValue(100)]
FAge: Integer;
```

TLessAttribute

Trysil.Validation.Attributes.TLessAttribute

The TLlessAttribute attribute defines that the column value must be less than:

```
[TLess(1000000)]
FPrice: Double;
```

TGreaterAttribute

Trysil.Validation.Attributes.TGreaterAttribute

The TGreaterAttribute attribute defines that the column value must be greater than:

```
[TGreater(0)]
FPrice: Double;
```

TRangeAttribute

Trysil.Validation.Attributes.TRangeAttribute

The TRangeAttribute attribute defines that the column value must be between:

```
[TRange(1, 1000000)]
FPrice: Double;
```

TRegexAttribute

Trysil.Validation.Attributes.TRegexAttribute

The TRegexAttribute attribute defines that the column value must be valid for the Delphi regular expression:

```
[TRegex('...')]
FLastname: String;
```

TEMailAttribute

Trysil.Validation.Attributes.TEMailAttribute

The TEMailAttribute attribute, inherited from TRegexAttribute, defines that the column value must contain a "valid" email address:

```
[TEMail]
FEmail: String;
```

Error message

For all the attributes seen so far, except for TDisplayNameAttribute, it is possible to define a custom error message:

```
[TDisplayName('Last Name')]
[TRequired('%0:s cannot be empty.')]
FLastname: String;
```

In this case, the result of the error message will be: *"Last Name cannot be empty."*.

TValidatorAttribute

Trysil.Validation.Attributes.TValidatorAttribute

The TValidatorAttribute attribute is used to decorate a method or more methods of the entity class that perform validation:

```
[TValidator]
procedure Validate();
```

Validation methods can be defined in three different ways. Based on needs we can choose between:

```
[TValidator]
procedure Validate();


[TValidator]
procedure Validate(const AErrors: TTValidationErrors);

[TValidator]
procedure Validate(
  const AContext: TTContext;
  const AErrors: TTValidationErrors);
```

Manipulating Objects

Trysil.Context.TTContext

Below are the public methods of TTContext for manipulating objects.

CreateEntity<T>

CreateEntity<T> should be used for creating new entities. In addition to creating the object, the primary key is calculated through the sequence and any Lazy columns are mapped.

```
var
  LPerson: TPerson;
begin
  LPerson := FContext.CreateEntity<TPerson>();
  ...
```

Don't use TPerson.Create!

Should Persona be destroyed? Yes, in case we decided not to let TTContext manage Identity Map.

Get<T>

Get<T> is used to read an entity from the database:

```
var
  LPerson: TPerson;
begin
  LPerson := FContext.Get<TPerson>(1);
  ...
```

SelectAll<T>

SelectAll<T> is used to read all entities from the database:

```
var
  LPersons: TTList<TPerson>;
begin
  LPersons := TTList<TPerson>.Create;
  try
    FContext.SelectAll<TPerson>(LPersons);
    ...
  finally
    LPersons.Free;
  end;
  ...
```

SelectCount<T>

SelectCount<T> is used to count how many entities are present in the database by specifying a filter:

```
var
  LCount: Integer;
begin
  LCount := FContext.SelectCount<TPerson>(
    TTFilter.Create('ID <= 10'));
  ...
```

Select<T>

Select<T> is used for reading entities from the database using a filter:

```
var
  LPersons: TTList<TPerson>;
begin
  LPersons := TTList<TPerson>.Create;
  try
    FContext.SelectAll<TPerson>(
      LPersons, TTFilter.Create('ID <= 10'));
    ...
  finally
    LPersons.Free;
  end;
  ...
```

TTFilter

Trysil.Filter.TTFilter

TTFilter is a record with the following constructors:

```
constructor Create(const AWhere: String);

constructor Create(
  const AWhere: String;
  const AMaxRecord: Integer;
  const AOrderBy: String);

constructor Create(
  const AWhere: String;
  const AStart: Integer;
  const ALimit: Integer;
  const AOrderBy: String);
```

The constructor parameters have the following meaning:

- **AWhere** is the WHERE that will be applied to the selection
- **AMaxRecord** is the maximum number of records returned by the query

- **AStart** is the record from which to start returning records
- **ALimit** , like AMaxRecord, is the maximum number of records returned by the query
- **AOrderBy** is sorting the data by the maximum number of records returned by the query

AOrderBy is required if you use AMaxRecord, AStart and ALimit. AOrderBy does not guarantee that the data is returned in the order indicated, but it guarantees that the data selection is performed based on that ordering.

Insert<T>

Insert<T> inserts an entity into the database by creating a new record:

```
var
  LPerson: TPerson;
begin
  LPerson := FContext.CreateEntity<TPerson>();
  LPerson.Firstname := 'David';
  LPerson.Lastname := 'Lastrucci';
  FContext.Insert<TPerson>(LPerson);
  ...
```

Update<T>

Update<T> updates an entity on the database:

```
var
  LPerson: TPerson;
begin
  LPerson := FContext.Get<TPerson>(1);
  LPerson.Email := 'david.lastrucci@trysil.com';
  FContext.Update<TPerson>(LPerson);
  ...
```

Delete<T>

Delete<T> deletes an entity from the database:

```
var
  LPerson: TPerson;
begin
  LPerson := FContext.Get<TPerson>(1);
  FContext.Delete<TPerson>(LPerson);
  ...
```

Transactions

Trysil supports database transactions (if the database supports them).

SupportTransaction

It is a property of the TTContext that indicates whether the associated TTConnection supports transactions.

CreateTransaction

It is a method of the TTContext that creates a TTTransaction object; upon its creation a transaction begins on the database.

TTTransaction

Trysil.Transaction.TTTransaction

It is the class that manages database transactions. By default, Trysil ends the transaction with a commit.

Rollback

It is a method of TTTransaction that rolls back the database transaction.

```
var
  LTransaction: TTTransaction;
begin
  LTransaction := nil;
  if FContext.SupportTransaction then
    LTransaction := FContext.CreateTransaction();
  try
    try
      ...
    except
      if Assigned(LTransaction) then
        LTransaction.Rollback;
      raise;
    end;
  finally
    if Assigned(LTransaction) then
      LTransaction.Free;
  end;
end;
```

TTSession<T>

Trysil.Session.TTSession<T>

It is a local transaction, something similar to the CachedUpdates already seen on various occasions with Delphi.

CreateSession<T>

It is a method of the TTContext that creates a TTSession<T> object for the start of a local transaction.

Insert

Inserts a new entity to the session.

Update

Updates a session entity.

Delete

Delete an entity from the session.

ApplyChanges

Applies changes made to entities within the session to the database.

```
var
  LSession: TTSession<TPerson>;
  LPerson1, LPerson2, LPerson3: TPerson;
begin
  LSession := FContext.CreateSession<TPerson>();
  try
    ...
    LSession.Insert(LPerson1);
    LSession.Update(LPerson2);
    LSession.Delete(LPerson3);
    ...

    LSession.ApplyChanges();
  finally
    LSession.Free;
  end;
end;
```

Operations Log

Trysil allows you to log operations.

TTLoggerThread

Trysil.Logger.TTLoggerThread

To enable Trysil's log, simply inherit the TTLoggerThread class and implement the following abstract methods:

```
type

 { TLoggerThread }

   TLoggerThread = class(TTLoggerThread)
   strict protected
     procedure LogStartTransaction(const AID: TTLoggerItemID); override;
     procedure LogCommit(const AID: TTLoggerItemID); override;
     procedure LogRollback(const AID: TTLoggerItemID); override;

     procedure LogParameter(
       const AID: TTLoggerItemID;
       const AName: String;
       const AValue: String); override;

     procedure LogSyntax(
       const AID: TTLoggerItemID; const ASyntax: String); override;
     procedure LogCommand(
       const AID: TTLoggerItemID; const ASyntax: String); override;

     procedure LogError(
       const AID: TTLoggerItemID; const AMessage: String); override;
     ...
```

TTLoggerItemID

Trysil.Logger.TTLoggerItemID

TTLoggerItemID is a record that contains the ConnectionID and ThreadID.

TTLogger

Trysil.Logger.TTLogger

Once you've created your logger class, simply register it as follows:

```
TTLogger.Instance.RegisterLogger<TLoggerThread>();
```

Or:

```
TTLogger.Instance.RegisterLogger<TLoggerThread>(5);
```

Where 5 is thread pool: the number of threads of type TLoggerThread that need to be created. By default, only one is created.

Licence

David Lastrucci
https://www.lastrucci.net
david.lastrucci@gmail.com