{TRYSIL}

Trysil – Delphi ORM

Copyright © David Lastrucci
All rights reserved

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

Versione italiana

```
const
  Volume: Integer = 1;
```

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.

Sommario

Panoramica 1
Benefici 1
Introduzione 2
Versioni Delphi supportate 2
Data Model 2
PODO (Plain Old Delphi Objects) 3
Installazione 4
Git Bash 4
Build Lib 4
Environment Variables 4
Expert 5
New Delphi Project 5
Connessione al database 6
Database supportati 6
TTConnection 6
TTFirebirdSQLConnection 7
TTPostgreSQLConnection 8
TTSQLiteConnection 9
TTSqlServerConnection 10
TTContext 11
Connection pool 11
TTFireDACConnectionPool 12
Identity Map 12

Metadati 13
TTContext.GetMetadata<T>() 13
TTTableMetadata 13
TTColumnMetadata 13
Mapping 14
Attributi 14
TTableAttribute 14
TSequenceAttribute 14
TWhereClauseAttribute 15
TRelationAttribute 15
TPrimaryKeyAttribute 16
TColumnAttribute 16
TDetailColumnAttribute 16
TVersionColumnAttribute 17
Esempio di entità “mappata” 18
TTPrimaryKey 19
TTVersion 19
TTNullable<T> 19
Lazy loading 20
TTLazy<T> 21
Implementation 21
TTLazyList<T> 22
Implementation 22
Costruttori 23

Eventi 24
TTEvent<T> 24
TInsertEventAttribute 25
TUpdateEventAttribute 25
TDeleteEventAttribute 26
Metodi della classe entità 27
TBeforeInsertEventAttribute 27
TAfterInsertEventAttribute 27
TBeforeUpdateEventAttribute 28
TAfterUpdateEventAttribute 28
TBeforeDeleteEventAttribute 29
TAfterDeleteEventAttribute 29
{$RTTI EXPLICIT...} 30
Validazione dei dati 31
Attributi 31
TDisplayNameAttribute 31
TRequiredAttribute 31
TMinLengthAttribute 31
TMaxLengthAttribute 32
TMinValueAttibute 32
TMaxValueAttribute 32
TLessAttribute 32
TGreaterAttribute 33
TRangeAttribute 33

TRegexAttribute 33
TEMailAttribute 33
Messaggio di errore 34
TValidatorAttribute 34
Manipolazione di oggetti 35
CreateEntity<T> 35
Get<T> 35
SelectAll<T> 36
SelectCount<T> 36
Select<T> 37
TTFilter 37
Insert<T> 39
Update<T> 39
Delete<T> 39
Transazioni 40
SupportTransaction 40
CreateTransaction 40
TTTransaction 40
Rollback 41
TTSession<T> 42
CreateSession<T> 42
Insert 42
Update 42
Delete 42

ApplyChanges 43
Log delle operazioni 44
TTLoggerThread 44
TTLoggerItemID 44
TTLogger 45
Licenza 46

Panoramica

Utilizzare un ORM (Object–Relational Mapping) per lo sviluppo di applicazioni significa parlare con il database utilizzando il linguaggio degli oggetti e non raccontare lunghe e noiose frasi SQL.

Benefici

Un ORM è fondamentalmente uno strato di software (una libreria) che permette di non scrivere codice come questo:

```
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]));
```

In favore di codice come questo:

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

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

Introduzione

Versioni Delphi supportate

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

Data Model

Il data model è un insieme di classi che possono essere lette e salvate su un database. Le classi del data model prendono il nome di entità.

Di seguito l'esempio dell'entità TPerson:

```
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)

Le entità, in Trysil, sono oggetti classici, non oggetti speciali: le entità possono essere ereditate direttamente da TObject.

Il termine PODO deriva da POJO (Plain Old Java Object) che fu coniato da Martin Fowler, Rebecca Parsons e Josh MacKenzie, nel 2000, con questa giustificazione:

"*Ci domandavamo perché le persone si opponessero tantissimo ad usare oggetti regolari nei loro sistemi ed abbiamo concluso che era perché gli oggetti semplici non avevano un nome accattivante. Allora gliene abbiamo dato uno che ha preso molto piede*".

Installazione

Una volta fatto il clone del repositoty Trysil presente su GitHub è necessario eseguire alcuni passaggi per poter utilizzare l'ORM.

Git Bash

Per semplicità assumiamo di fare il clone del repository nella cartella C:\

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

Una volta terminato il clone, la cartella C:\Trysil conterrà tutti i sorgenti di Trysil presenti nel repository GitHub.

Build Lib

Aprire "Trysil.groupproj" nella cartella C:\Trysil\Packages\???[1] ed eseguire la Build in tutte le configurazioni e piattaforme necessarie.

Attenzione: se si utilizza l'edizione Community oppure l'edizione Professional di Delphi, non sarà possibile compilare il progetto Trysil.SqlServer???.dproj e non sarà possibile utilizzare connessioni per Microsoft SQL Server.

La cartella C:\Trysil\Lib\???\$(Platform)\$(Config) adesso contiene tutte le Bpl, i Dcp ed i Dcu di Trysil.

Environment Variables

In "Tools -> Options -> Environment Variables" aggiungere una nuova User Variable:

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

[1] ??? rappresenta la versione Delphi che si sta utilizzando. Si veda il paragrafo "Versioni Delphi Supportate"

Expert

Aprire il progetto Trysil.Expert???.dproj presente nella cartella C:\Trysil\Trysil.Expert, e compilarlo.

Chiudere Delphi.

Eseguire regedit.exe e posizionarsi nella cartella

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

Aggiungere un nuovo valore stringa:

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

Avviare di nuovo Delphi.

Nello Splash Screen e nel Menù adesso è presente Trysil.

New Delphi Project

Creare un nuovo progetto Delphi e, in “Project -> Options -> Building -> Delphi Compiler” selezionare “All configurations - All Platforms” e nella “Search Path” scrivere:

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

Il nuovo progetto è adesso pronto per poter utilizzare Trysil.

Connessione al database

Database supportati

<table>
<tr><th rowspan="2">Database</th><th colspan="2">Edizione di Delphi</th></tr>
<tr><th>Community / Professional</th><th>Enterprise / Architect</th></tr>
<tr><td>FirebirdSQL</td><td rowspan="2">✔ [2]</td><td rowspan="4">✔</td></tr>
<tr><td>PostgreSQL</td></tr>
<tr><td>SQLite</td><td>✔</td></tr>
<tr><td>SQL Server</td><td></td></tr>
</table>

TTConnection

Trysil.Data.TTConnection

TTConnection è la connessione astratta al database; da TTConnection ereditano, direttamente o indirettamente tutte le altre tipologie di connessioni.

Il costruttore di TTConnection richiede il nome di una connessione che deve essere registrata precedentemente tramite RegisterConnection.

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

RegisterConnection è un metodo introdotto dai discendenti di TTConnection.

[2] Disponibile solo per connessioni a localhost (limite imposto da FireDAC)

TTFirebirdSQLConnection

Trysil.Data.FireDAC.FirebirdSQL.TTFirebirdSQLConnection

Connessione ad un database di tipo FirebirdSQL. È possibile registrare una connessione nei seguenti modi:

```
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

Connessione ad un database di tipo PostgreSQL. È possibile registrare una connessione nei seguenti modi:

```
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

Connessione ad un database di tipo SQLite. È possibile registrare una connessione nei seguenti modi:

```
 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

Connessione ad un database di tipo SQL Server. È possibile registrare una connessione nei seguenti modi:

```
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 è la classe che permette di dialogare con il database.

All'interno di TTContext viene utilizzata TTProvider per le operazioni di lettura e TTResolver per quelle di scrittura.

TTProvider e TTResolver non devono essere utilizzati direttamente.

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

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

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

Connection pool

Il pool riduce il numero di volte in cui è necessario aprire nuove connessioni, gestendone e mantenendone un set per ogni configurazione.

Quando l'applicazione apre una connessione, il pool verifica la presenza di una disponibile e se disponibile la restituisce al chiamante invece di crearne una nuova.

Quando l'applicazione chiude una connessione, il pool la restituisce al set di quelle disponibili invece di distruggerla realmente.

Una volta restituita al pool, la connessione può essere usata nuovamente nella successiva apertura.

TTFireDACConnectionPool

Trysil.Data.FireDAC.ConnectionPool.TTFireDACConnectionPool

È possibile attivare o disattivare il pool di connessioni di Trysil in questo modo:

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

Identity Map

Trysil.IdentityMap.TTIdentityMap

TTIdentityMap è una cache delle entità affidata a TTContext.

Se l'entità richiesta è già stata letta dal database, tramite l'utilizzo di TTIdentityMap, viene restituita la stessa istanza dell'entità letta precedentemente.

L'utilizzo di TTIdentityMap garantisce comunque dati aggiornati.

TTContext, per default, utilizza identity map che può essere disabilitato durante la creazione dello stesso.

Di seguito i costruttori di TTContext:

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

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

Metadati

Le operazioni di Insert, Update e Delete sul database utilizzano i parametri, sia per garantire sicurezza che per assicurare efficienza. Per questo motivo Trysil ha bisogno dei metadati.

TTContext.GetMetadata<T>()

Tramite il metodo GetMetadata<T> di TTContext possiamo accedere ai metadati delle entità:

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

TTTableMetadata

Trysil.Metadata.TTTableMetadata

Contiene i metadati della tabella collegata ad una entità:

- **TableName** il nome della tabella del database
- **PrimaryKey** il nome della colonna chiave primaria della tabella del database
- **Columns** la lista delle colonne del database

TTColumnMetadata

Trysil.Metadata.TTColumnMetadata

Contiene i metadati della colonna della tabella collegata ad una entità:

- **ColumnName** il nome della colonna della tabella del database
- **DataType** il tipo (TFieldType) della colonna della tabella del database
- **DataSize** la dimensione della colonna della tabella del database

Mapping

Il mapping viene utilizzato per istruire Trysil ad interagire con il database: come leggere, inserire, aggiornare ed eliminare i dati utilizzando una entità.

Attributi

TTableAttribute

Trysil.Attributes.TTableAttribute

Si applica alla classe e definisce il nome della tabella del database:

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

TSequenceAttribute

Trysil.Attributes.TSequenceAttribute

Si applica alla classe e definisce il nome della sequenza[3] del database che sarà utilizzata per la chiave primaria dell'entità:

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

[3] SQLite non metta a disposizione niente di simile alle sequenze. Al suo posto viene utilizzata la colonna ROWID (un concetto molto simile a MAX + 1). Per questo motivo è sconsigliato l'utilizzo di SQLite in ambiente multi utente.

TWhereClauseAttribute

Trysil.Attributes.TWhereClauseAttribute

È possibile decorare la classe con l'attributo TWhereClauseAttribute per filtrare i dati della tabella del database sui quali l'entità andrà a lavorare.

Aggiungiamo ad esempio la colonna PersonType alla nostra tabella Persons e definiamo che potrà contenere i valori 1 (manager) e 2 (impiegato).

Possiamo andare a definire le nostre entità TManager e TEmployee in questo modo:

```
type

{ TManager }

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

{ TEmployee }

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

TRelationAttribute

Trysil.Attributes.TRelationAttribute

Si applica alla classe e definisce la relazione dell'entità con un'altra tabella del database:

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

TRelationAttribute richiede tre parametri:

- **TableName** il nome della tabella in relazione
- **ColumnName** il nome della colonna in relazione
- **IsCascade** indica se la relazione è di tipo cascade o meno

TPrimaryKeyAttribute

Trysil.Attributes.TPrimaryKeyAttribute

Si applica ad un field della classe entità e definisce la chiave primaria della tabella del database:

```
[TPrimaryKey]
```

TColumnAttribute

Trysil.Attributes.TColumnAttribute

Si applica ad un field della classe entità e definisce il nome della colonna sulla tabella del database:

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

TDetailColumnAttribute

Trysil.Attributes.TDetailColumnAttribute

Si applica ad un field della classe entità (di solito ad una colonna di tipo TTLazyList<T>) e definisce un dettaglio di entità (master/detail) :

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

TVersionColumnAttribute

Trysil.Attributes.TVersionColumnAttribute

Trysil, per gestire la concorrenza sui dati, utilizza una Colonna di tipo versione. La colonna versione è un Int32 che viene incrementata ad ogni aggiornamento (Update).

L'attributo TVersionColumnAttribute si applica al field della classe che rappresenta la versione del record:

```
[TVersionColumn]
```

Per le colonne di tipo versione su utilizza un field di tipo TTVersion.

Esempio di entità "mappata"

Di seguito l'entità TPerson "mappata" con gli attributi di Trysil:

```
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 è il tipo da utilizzare per la chiave primaria delle entità. È un alias di Int32.

TTVersion

Trysil.Types.TTVersion

TTVersion è il tipo da utilizzare per la colonna versione delle entità ed anche lui è un alias di Int32.

TTNullable<T>

Trysil.Types.TTNullable<T>

I database supportano le colonne NULL; per poter gestire questo tipo di colonne, in Trysil, sono stati introdotti i tipi TTNullable<T>.

TTNullable<T> è un record che implementa una serie di "class operator" in modo da permettere, ad esempio, l'assegnazione di un nullable al suo corrispettivo tipo e viceversa.

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

Lazy loading

Tramite il meccanismo del "lazy loading", le entità vengono lette dal database solo quando ne abbiamo la necessità. Analizziamo uno dei primi esempi di questo documento e vediamo cosa succede dietro le quinte:

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

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

- L'istruzione *FContext.Get<TInvoice>(1)* esegue la lettura dal database della fattura con ID uguale a 1
- L'istruzione *LInvoice.Customer* esegue la lettura dal database del cliente collegato alla fattura precedentemente letta
- L'istruzione *LInvoice.Customer.Country* esegue la lettura dal database della nazione collegata al cliente precedentemente letto

TTLazy<T>

Trysil.Lazy.TTLazy<T>

TTLazy<T> è il tipo di colonna che rappresenta una “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> è il tipo di colonna che rappresenta una “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;
```

Costruttori

Come già detto, le entità possono essere di tipo PODO e quindi, possono essere ereditate direttamente da TObject.

L'unico vincolo che abbiamo è il costruttore; dobbiamo sceglierne uno tra:

- Costruttore di default, quello senza parametri
- Costruttore con un solo parametro di tipo TTContext

Eventi

Gli eventi fanno parte del modello. Per ogni entità è possibile definire gli eventi prima (before) e dopo (after) l'inserimento (INSERT), l'aggiornamento (UPDATE) e la cancellazione (DELETE).

Gli eventi possono essere definiti in due modi:

- Creando una classe che eredita da TTEvent<T>
- Definendo i metodi direttamente all'interno della classe entità

TTEvent<T>

Trysil.Events.TTEvent<T>

TTEvent<T> definisce due metodi virtuali che possono essere sovrascritti (override):

- 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

L'attributo TInsertEventAttribute si applica alla classe e definisce l'evento da eseguire durante l'inserimento di una nuova entità:

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

TUpdateEventAttribute

Trysil.Events.Attributes.TUpdateEventAttribute

L'attributo TUpdateEventAttribute si applica alla classe e definisce l'evento da eseguire durante l'aggiornamento di una entità:

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

TDeleteEventAttribute

Trysil.Events.Attributes.TDeleteEventAttribute

L'attributo TDeleteEventAttribute si applica alla classe e definisce l'evento da eseguire durante la cancellazione di una entità:

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

Metodi della classe entità

TBeforeInsertEventAttribute

Trysil.Events.Attributes.TBeforeInsertEventAttribute

L'attributo TBeforeInsertEventAttribute si applica al metodo da eseguire prima dell'inserimento dell'entità:

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

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

TAfterInsertEventAttribute

Trysil.Events.Attributes.TAfterInsertEventAttribute

L'attributo TAfterInsertEventAttribute si applica al metodo da eseguire dopo l'inserimento dell'entità:

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

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

TBeforeUpdateEventAttribute

Trysil.Events.Attributes.TBeforeUpdateEventAttribute

L'attributo TBeforeUpdateEventAttribute si applica al metodo da eseguire prima dell'aggiornamento dell'entità:

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

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

TAfterUpdateEventAttribute

Trysil.Events.Attributes.TAfterUpdateEventAttribute

L'attributo TAfterUpdateEventAttribute si applica al metodo da eseguire dopo l'aggiornamento dell'entità:

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

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

TBeforeDeleteEventAttribute

Trysil.Events.Attributes.TBeforeDeleteEventAttribute

L'attributo TBeforeDeleteEventAttribute si applica al metodo da eseguire prima della cancellazione dell'entità:

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

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

TAfterDeleteEventAttribute

Trysil.Events.Attributes.TAfterDeleteEventAttribute

L'attributo TAfterDeleteEventAttribute si applica al metodo da eseguire dopo la cancellazione dell'entità:

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

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

{$RTTI EXPLICIT…}

Per impostazione predefinita, Delphi non genera le informazioni RTTI per i metodi non pubblici.

È necessario aggiungere la direttiva {$RTTI EXPLICIT…} alla classe dell'entità perché i metodi sono stati definiti privati, e, in questa situazione, Trysil non sarebbe in grado di sapere quali metodi invocare al verificarsi degli eventi.

Abbiamo quindi due possibilità:

- Aggiungere la direttiva {RTTI EXPLICIT…}
- Definire i metodi di tipo public

Personalmente preferisco istruire Delphi per fargli generare le informazioni RTTI dei metodi privati piuttosto che renderli pubblici a tutta l'applicazione.

Validazione dei dati

Trysil fornisce modi semplici e diretti per convalidare le entità prima che vengano persistite sul database.

Aggiungendo attributi specifici ai fields dell'entità, è possibile assicurarsi che quest'ultima venga sempre salvata nel database in uno stato corretto.

Attributi

TDisplayNameAttribute

Trysil.Validation.Attributes.TDisplayNameAttribute

L'attributo TDisplayNameAttribute definisce il nome da utilizzare in caso di errore di validazione:

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

TRequiredAttribute

Trysil.Validation.Attributes.TRequiredAttribute

L'attributo TRequiredAttribute definisce che quella colonna è necessaria, obbligatoria, e non può essere lasciata vuota:

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

TMinLengthAttribute

Trysil.Validation.Attributes.TMinLengthAttribute

L'attributo TMinLengthAttribute definisce la lunghezza minima del valore della colonna:

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

TMaxLengthAttribute

Trysil.Validation.Attributes.TMaxLengthAttribute

L'attributo TMaxLengthAttribute definisce la lunghezza massima del valore della colonna:

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

TMinValueAttibute

Trysil.Validation.Attributes.TMinValueAttribute

L'attributo TMinValueAttribute definisce il valore minimo per la colonna:

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

TMaxValueAttribute

Trysil.Validation.Attributes.TMaxValueAttribute

L'attributo TMaxValueAttribute definisce il valore massimo per la colonna:

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

TLessAttribute

Trysil.Validation.Attributes.TLessAttribute

L'attributo TLessAttribute definisce che il valore della colonna deve essere inferiore di:

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

TGreaterAttribute

Trysil.Validation.Attributes.TGreaterAttribute

L'attributo TGreaterAttribute definisce che il valore della colonna deve essere superiore a:

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

TRangeAttribute

Trysil.Validation.Attributes.TRangeAttribute

L'attributo TRangeAttribute definisce che il valore della colonna deve essere compreso tra:

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

TRegexAttribute

Trysil.Validation.Attributes.TRegexAttribute

L'attributo TRegexAttribute definisce che il valore della colonna deve essere valido per l'espressione regolare di Delphi:

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

TEMailAttribute

Trysil.Validation.Attributes.TEMailAttribute

L'attributo TEMailAttribute, ereditato da TRegexAttribute, definisce che il valore della colonna deve contenere un indirizzo email "valido":

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

Messaggio di errore

Per tutti gli attributi visti fino a qui, eccetto che per TDisplayNameAttribute, è possibile definire un messaggio di errore personalizzato:

```
[TDisplayName('Cognome')]
[TRequired('%0:s non può essere vuoto.')]
FLastname: String;
```

In questo caso, il risultato del messaggio di errore sarà: *"Cognome non può essere vuoto."*.

TValidatorAttribute

Trysil.Validation.Attributes.TValidatorAttribute

L'attributo TValidatorAttribute serve per decorare un metodo o più metodi della classe dell'entità che si occupano di eseguire la validazione:

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

I metodi di validazione possono essere definiti in tre diversi modi. In base alle necessità possiamo scegliere tra:

```
[TValidator]
procedure Validate();


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

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

Manipolazione di oggetti

Trysil.Context.TTContext

Di seguito i metodi pubblici di TTContext per la manipolazione degli oggetti.

CreateEntity<T>

CreateEntity<T> deve essere utilizzata per la creazione di nuove entità. Oltre alla creazione dell'oggetto, viene calcolata la chiave primaria tramite la sequenza e vengono mappate eventuali colonne di tipo Lazy.

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

Non utilizzare TPerson.Create!

LPerson deve essere distrutto? Sì, nel caso in cui abbiamo deciso di non fare gestire Identity Map al TTContext.

Get<T>

Get<T> è utilizzata per la lettura di una entità dal database:

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

SelectAll<T>

SelectAll<T> è utilizzata per la lettura di tutte le entità dal database:

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

SelectCount<T>

SelectCount<T> è utilizzata per contare quante entità sono presenti sul database specificando un filtro:

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

Select<T>

Select<T> è utilizzata per la lettura delle entità dal database utilizzando un filtro:

```
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 è un record con i seguenti costruttori:

```
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);
```

I parametri dei costruttori hanno il seguente significato:

- **AWhere** è il WHERE che sarà applicato alla selezione
- **AMaxRecord** è il numero massimo di record ritornati dalla query

- **AStart** è il record da cui iniziare a ritornare i record
- **ALimit**, come AMaxRecord, è il numero massimo di record ritornati dalla query
- **AOrderBy** è l'ordinamento dei dati il numero massimo di record ritornati dalla query

AOrderBy è necessario se si utilizza AMaxRecord, AStart e ALimit. AOrderBy non garantisce che i dati siano ritornati nell'ordine indicato, ma garantisce che la selezione dei dati sia eseguita sulla base di quell'ordinamento.

Insert<T>

Insert<T> esegue l'inserimento di una entità sul database creando un nuovo record:

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

Update<T>

Update<T> esegue l'aggiornamento di una entità sul database:

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

Delete<T>

Delete<T> esegue la cancellazione di una entità dal database:

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

Transazioni

Trysil supporta le transazioni del database (se il database le supporta).

SupportTransaction

È una proprietà del TTContext che indica se la TTConnection a lui associata supporta le transazioni.

CreateTransaction

È un metodo del TTContext che crea un oggetto TTTransaction; alla sua creazione inizia una transazione sul database.

TTTransaction

Trysil.Transaction.TTTransaction

È la classe che gestisce le transazioni del database. Per default, Trysil, termina la transazione con un commit.

Rollback

È un metodo di TTTransaction che esegue il Rollback della transazione del database.

```
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>

È una transazione locale, una cosa simile ai CachedUpdates già visti in varie occasioni con Delphi.

CreateSession<T>

È un metodo del TTContext che crea un oggetto TTSession<T> per l'inizio di una transazione locale.

Insert

Inserisce una nuova entità alla sessione.

Update

Aggiorna una entità della sessione.

Delete

Elimina una entità dalla sessione.

ApplyChanges

Applica le modifiche apportate alle entità all’interno della sessione sul 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;
```

Log delle operazioni

Trysil permette di eseguire il log delle operazioni.

TTLoggerThread

Trysil.Logger.TTLoggerThread

Per attivare il log di Trysil è sufficiente ereditare la classe TTLoggerThread ed implementare i seguenti metodi astratti:

```
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 è un record che contiene la ConnectionID ed il ThreadID.

TTLogger

Trysil.Logger.TTLogger

Una volta creata la nostra classe per il logger, è sufficiente registrarla nel seguente modo:

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

Oppure:

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

Dove 5 indica pool di thread: il numero di thread di tipo TLoggerThread che devono essere creati. Per default ne viene creato uno soltanto.

Licenza

La ridistribuzione e l'utilizzo in formato sorgente e binario, con o senza modifiche, sono consentiti a condizione che siano soddisfatte le seguenti condizioni:

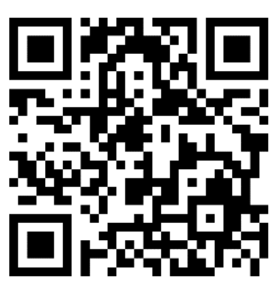
- Le ridistribuzioni del codice sorgente devono mantenere l'avviso di copyright di cui sopra, questo elenco di condizioni e la seguente dichiarazione di non responsabilità.
- Le ridistribuzioni in formato binario devono riprodurre l'avviso di copyright di cui sopra, questo elenco di condizioni e la seguente dichiarazione di non responsabilità nella documentazione e/o in altri materiali forniti con la distribuzione.
- Né il nome di questa libreria né i nomi dei suoi collaboratori possono essere utilizzati per avallare o promuovere prodotti derivati da questo software senza previa autorizzazione scritta specifica.

Questo software viene fornito dai detentori del copyright e dai collaboratori così com'è e si declina qualsiasi garanzia esplicita o implicita, incluse, a titolo esemplificativo ma non esaustivo, le garanzie implicite di commerciabilità e idoneità per uno scopo particolare.

In nessun caso il titolare del copyright o i collaboratori saranno responsabili per qualsiasi danno diretto, indiretto, incidentale o speciale.

sta utilizzando una
versione modificata di
“Trysil – Delphi ORM”
per scrivere il suo nuovo ERP

14 febbraio 2024



David Lastrucci
R&D Manager
Open Source Italia S.r.l.
https://www.ositalia.com
david.lastrucci@ositalia.com