

iupORM

small Delphi ORM for mobile

Maurizio Del Magno

mauriziodelmagno@gmail.com

Cos'è lupOrm?

lupOrm, come dice il nome stesso è un O.R.M. (Object Relational Mapping); in pratica è uno strumento software che fornisce, mediante un'interfaccia orientata agli oggetti, servizi inerenti alla persistenza degli oggetti/dati stessi astraendo nel contempo dai dettagli implementativi relativi al RDBMS (database) utilizzato e dalle relative query per il salvataggio e successivo caricamento dei dati e quindi degli oggetti.

Ancora più semplicemente il programmatore crea le classi per rappresentare i dati e i relativi comportamenti (business logic) senza doversi preoccupare di come poi questi oggetti verranno materialmente salvati/recuperati dal database sottostante.

Per quale motivo è stato creato?

Lo sviluppo di lupOrm, almeno inizialmente, è cominciato soprattutto per scopi didattici del sottoscritto; dopo aver frequentato un corso sulla progettazione di architetture software ad alta manutenibilità volevo mettermi alla prova per verificare se riuscivo a realizzare qualcosa dotato di un minimo di complessità separando bene le responsabilità dei vari moduli (classi) di cui è composto.

Successivamente ho pensato che quanto realizzato potesse essermi utile nello sviluppo di applicazioni mobile, soprattutto per velocizzare tutte quelle parti di un software ripetitive relative allo strato di persistenza dei dati come creazione del database locale, query di caricamento/salvataggio/eliminazione di dati.

Durante lo sviluppo di lupOrm sono stato più volte aiutato e indirizzato nella giusta direzione da Daniele Teti, creatore di DORM (the Delphi ORM), progetto sicuramente più grande e completo sotto molti punti di vista da cui ho preso vari spunti. Ringrazio Daniele per non avermi mai negato i suoi consigli.

lupOrm non ha nessuna velleità di competere con DORM in quanto nato volutamente per essere più leggero, snello e specificamente realizzato per la creazione di piccole app per IOS/Android nel modo più veloce e comodo possibile ma sicuramente meno completo e potente in termini di possibilità di mapping e supporto di diversi RDBMS oltre che nelle possibilità di logging.

Quali RDBMS supporta?

Attualmente lupOrm è in grado di operare solo con SQLite. La sua architettura dovrebbe comunque permettere di renderlo compatibile con altri RDBMS con una certa facilità (a me personalmente interesserà soprattutto Firebird/Interbase in un futuro non molto lontano).

Come si configura lupOrm?

(vedi vari esempi forniti con lupOrm, di solito nella unit visibile cliccando su Project->ViewSource)

Una peculiarità di lupOrm, almeno per il momento, è quello di non aver bisogno di alcuna configurazione; non c'è alcun file da configurare. L'unica impostazione possibile, peraltro facoltativa, riguarda la possibilità di specificare un percorso (path) del file del database. Normalmente per applicazioni mobile non è necessario specificare né il percorso né il nome del database, lupOrm farà tutto automaticamente, metterà tutto nella "documenti" locale e privata dell'applicazione nel file "db.db". In ambiente desktop invece, di default, metterà il file nella cartella "Documenti\\lupOrm\\db.db".

Se necessario ci sono due metodi di classe per impostare percorsi diversi:

- ***TlupOrm.SetDBFolder(AFolderName: String);***
- ***TlupOrm.SetDBFolderInDocuments(AFolderName: String);***

Esempi di utilizzo:

- *TlupOrm.SetDBFolderInDocuments('Pizza');*
- *TlupOrm.SetDBFolderInDocuments('lupOrm PhoneContacts Database');*

I due esempi impostano la posizione del database nella sottocartella "Pizza" e "lupOrm PhoneContacts Database" all'interno della cartella "Documenti".

Auto creazione del database

(vedi vari esempi forniti con lupOrm, di solito nella unit visibile cliccando su Project->ViewSource)

lupOrm è in grado di creare automaticamente il database dell'applicazione completo di tabelle con tutti i campi necessari alla memorizzazione delle proprietà delle varie classi che compongono il "Model" e al funzionamento di lupOrm stesso chiamando il seguente metodo:

- *lupOrm.AutoCreateDatabase;*

Chiamando questo metodo lupOrm eseguirà la scansione di tutte le classi del Model, determinerà e creerà tabelle e campi del database tenendo conto anche di eventuali relazioni (HasOne, HasMany e BelongsTo), terrà conto anche dell'ereditarietà tra le classi e di eventuali campi aggiuntivi necessari al suo stesso funzionamento.

Nel caso venissero successivamente aggiunte proprietà ad una classe, richiamando il metodo *lupOrm.AutoCreateDatabase* a database già esistente, lupOrm analizzerà sia il database che le classi e aggiungerà automaticamente il campo relativo al nuovo dato adeguando così il database senza perdita dei dati esistenti.

NB: Se si utilizza la creazione automatica del database è necessario inserire la direttiva `{$STRONGLINKTYPES ON}` nel file principale dell'applicazione (Project.ViewSource).

Preparare le classi del Model per lupOrm

(vedi vari esempi forniti con lupOrm, unit "Model")

Sebbene lupOrm sia in grado di svolgere la funzione di persistenza degli oggetti/classi facenti parte del model dell'applicazione anche senza necessità di alcun parametro/impostazione (grazie ad alcune convenzioni), si possono dare indicazioni per derogare da dette convenzioni attraverso l'uso degli "attributes" che le recenti versioni di Delphi mettono a disposizione.

Ecco un elenco degli attributes disponibili per le classi:

- **[ioTable('TableName')]**
- **[ioClassFromField]**

di seguito invece un elenco degli attributes disponibili per le proprietà:

- **[ioField('FieldName')]**
- **[ioOID]**
- **[ioSkip]**
- **[ioBelongsTo(ChildClassRef)]**
- **[ioHasOne(ChildClassRef, 'ChildPropertyName')]**
- **[ioHasMany(ChildClassRef, 'ChildPropertyName')]**

[ioTable('TableName')]

Per default lupOrm mappa la classe in una tabella con lo stesso nome della classe stessa senza il carattere "T" iniziale; ad esempio la classe TPerson verrebbe mappata su una tabella di nome "Person". Con questo attributo si indica a lupOrm di derogare da questa convenzione e di utilizzare invece una tabella di nome diverso; ad esempio inserendo l'attributo **[ioTable('PEOPLE')]** prima della dichiarazione della classe si indica a lupOrm di mapparla sulla tabella di nome 'PEOPLE'.

ATTENZIONE !!! - Nel caso si intenda utilizzare il metodo TlupOrm.AutoCreateDatabase per la creazione/ristrutturazione automatica del database questo attributo diventa obbligatorio in ogni caso, anche se il nome della tabella coincide con il nome che lupOrm utilizzerebbe di default, questo perché questa funzionalità prende in considerazione solo le classi che hanno l'attributo [ioTable...] esplicito.

[ioClassFromField] (vedi gli esempi "PhoneContacts")

Mi è difficile spiegare l'uso di questo attributo ma ci provo. Poniamo il caso dell'utilizzo della seguente riga di codice:

```
- TlupOrm.Load<TPerson>.ToList<TObjectList<TPerson>>;
```

lupOrm ci restituirà una ObjectList di TPerson e tutto sembra andare bene, ma potremmo essere nella situazione in cui esista una gerarchia di classi derivate da TPerson ad esempio:

- TPerson > TCustomer
- TPerson > TCustomer > TVIPCustomer
- TPerson > TEmployee

Se eseguiamo la stessa riga di codice in questa situazione senza attributo [ioClassFromField] lupOrm (come altri ORM) restituirebbe tutti i record contenuti nella tabella (e questo è giusto visto che sono comunque tutti dei TPerson) ma tutti come TPerson, anche i record relativi a dei TCustomer/TVipCustomer/TEmployee verrebbero restituiti tutti come TPerson rendendo impossibile l'accesso a proprietà e metodi specifici delle sottoclassi.

Inoltre se eseguiamo la stessa riga specificando però il generic come TCustomer ci verrebbero comunque restituiti tutti i record e tutti del tipo specificato nel generic stesso (TCustomer in questo caso) e questo potrebbe essere un problema. Peraltro non viene rispettata nemmeno l'ereditarietà visto che un TPerson e un TEmployee non sono dei TCustomer ma mi verrebbero restituiti ugualmente a meno di non specificare "manualmente" un filtro che me li escluda.

Specificando invece l'attributo [ioClassFromField] invece lupOrm è in grado di gestire correttamente la situazione e restituirebbe nel primo caso sempre una TObjectList<TPerson> contenente gli oggetti relativi a tutti i record ma ognuno della rispettiva classe (TPerson come TPerson, TEmployee come TEmployee ecc.) e persistendoli eventualmente sempre correttamente mappati ciascuno in base alla propria classe. Nel secondo caso invece restituirebbe una TObjectList<TCustomer> contenente solo i TCustomer e i TVIPCustomer (i TPerson e i TEmployee vengono esclusi perché non sono dei TCustomer) e sempre ognuno della propria specifica classe rispettando quindi l'ereditarietà e senza necessità di specificare manualmente alcun filtro.

[ioField('FieldName')]

lupOrm mappa ogni proprietà di una classe in un campo con lo nome della proprietà stessa. Si può indicare di utilizzare un campo dal nome diverso scrivendo questo attributo subito prima della dichiarazione della proprietà e specificanti nel parametro il nome desiderato.

[ioOID]

Questo attributo serve a indicare a lupOrm che la proprietà è l'ID e la primary key della tabella relativa alla classe. Se il nome della proprietà è proprio "ID" è possibile omettere l'attributo.

NB: lupOrm è in grado di gestire solamente campi ID di tipo intero.

[ioSkip]

Serve a indicare allo strumento che la proprietà deve essere ignorata durante la fase di caricamento/salvataggio. Viene usato per le proprietà “calcolate” che non devono essere persistite; ad esempio nel caso la classe TPerson avesse una proprietà “Senior” di tipo boolean che assume valore True nel caso la proprietà “Age” contenga assuma valore superiore a 18 non è necessario che venga persistita (Senior) e quindi si può usare l’attributo per fare in modo che lupOrm non consideri la proprietà durante le fasi di load o di persist.

ioHasMany(ChildClassRef, ‘ChildPropertyName’)] (vedi gli esempi “PhoneContacts”)

Supponiamo che la classe TPerson (e quindi anche tutte le sue discendenti) abbia una proprietà “Phones” di tipo TObjectList<TPhoneNumber> contenente quindi un elenco di numeri di telefono relativi alla persona, si tratta quindi di una relazione master-detail uno a molti (HasMany appunto); utilizzando l’attributo possiamo indicare a lupOrm l’esistenza di questa relazione e permettergli di provvedere automaticamente al caricamento (e alla persistenza) anche gli oggetti di dettaglio.

Esempio: **[ioHasMany(TPhoneNumber, ‘PersonID’)]** indica a lupOrm che la proprietà contiene, con una relazione uno a molti, una lista di oggetti di tipo TPhoneNumber legati alla persona tramite la proprietà “PersonID” (foreign key nella classe dettaglio). E’ importante notare che la relazione viene definita sempre in termini di classe e proprietà astraendo completamente dal database sottostante.

[ioHasMany(ChildClassRef, ‘ChildPropertyName’)]

Vedi quanto spiegato per l’attributo **ioHasMany(ChildClassRef, ‘ChildPropertyName’)]**

[ioBelongsTo(ChildClassRef)]

Vedi quanto spiegato per l’attributo **ioHasMany(ChildClassRef, ‘ChildPropertyName’)]**

Usiamo lupOrm

(vedi vari esempi forniti con lupOrm)

Load "ToObject"

Possiamo chiedere di caricare un oggetto di tipo TPerson, assegnandolo ad una variabile, in questo modo:

APerson := TlupOrm.Load<TPerson>.ByOID(1).ToObject;

Il codice sopra chiede a lupOrm di caricare dal database l'oggetto di tipo TPerson con ID = 1 astruendo completamente dai dettagli implementativi del database.

La prima parte ***TlupOrm.Load<TPerson>*** indica che deve essere caricato un oggetto di tipo TPerson; lupOrm analizza (attraverso l'RTTI di delphi) la classe TPerson e genera automaticamente le query necessarie all'operazione (in questo caso di caricamento).

La parte centrale ***ByOID(1)*** indica allo strumento che l'operazione deve essere relativa all'oggetto/record con ID = 1 anche in questo caso astruendo completamente dal nome del campo sottostante.

La parte finale ***ToObject*** specifica la destinazione dell'operazione, in questo caso indica che il risultato deve essere restituito come singolo oggetto. Se l'oggetto possiede una o più proprietà contenenti altri oggetti o liste di oggetti e, nella definizione della classe, è stato fatto un corretto uso dell'attributo ioHasOne/ioHasMany/ioBelongsTo, verranno caricati anche questi oggetti di dettaglio restituendo in questo modo un oggetto completo al 100%.

Load "ToList"

Possiamo chiedere a lupOrm di caricare una lista di oggetti in modo assolutamente analogo al caricamento di un sincope oggetto sostituendo la parte finale ***ToObject*** con la dicitura ***ToList***.

Es: ***APersonList := TlupOrm.Load<TPerson>.ToList;***

Restituisce una TObjectList<TPerson> contenente tutti gli oggetti TPerson presenti nella corrispondente tabella del database.

In alcuni casi però potremmo avere bisogno di un tipo di lista diversa da quella restituita di default; ad esempio potremmo chiedere di ricevere una TList<T> invece di una TObjectList<T>, lupOrm ci permette di effettuare tale richiesta in questo modo:

APersonList := TlupOrm.Load<TPerson>.ToList<TList<TPerson>>;

Grazie alla possibilità di specificare il tipo della lista risultato in modo completamente indipendente dal tipo di dato oggetto del caricamento, con lupOrm diventa possibile ad esempio chiedere il caricamento di una lista di oggetti che implementano una determinata interfaccia in questo modo:

APersonList := TlupOrm.Load<TPerson>.ToList<TList<IPersonInterface>>;

L'istruzione ritorna una lista di tipo TList<IPersonInterface> che potrebbe anche contenere oggetti di classi diverse (anche con nessun legame di ereditarietà) purché implementino l'interfaccia IPersonInterface.

Se uniamo quanto sopra esposto all'utilizzo dell'attributo ***[ioClassFromField]*** e consideriamo la gerarchia di classi dei nostri esempi precedenti:

- TPerson > TCustomer
- TPerson > TCustomer > TVIPCustomer
- TPerson > TEmployee

supponiamo che la dichiarazione delle classi sia come segue:

```
[ioTable('PERSONS_VIEW')]  
[ioClassFromField]  
TPerson = class  
...  
end;  
  
[ioTable('CUSTOMERS')]  
[ioClassFromField]  
TCustomer = class(TPerson)  
...  
end;  
  
[ioTable('CUSTOMERS')]  
[ioClassFromField]  
TVIPCustomer = class(TCustomer)  
...  
end;  
  
[ioTable('EMPLOYEE')]  
[ioClassFromField]  
TEmployee = class(TPerson)  
...  
end;
```

dove TPerson è la radice della gerarchia ed è mappata sulla vista del database "PERSONS_VIEW" che unisce al suo interno il risultato di più query select sulle tabelle "CUSTOMERS" e "EMPLOYEE" relative alle classi discendenti (CUSTOMERS per le classi TCustomer e TVIPCustomer; EMPLOYEE per la classe TEmployee); eseguendo qualunque delle righe di esempio precedenti otteniamo una lista di oggetti di classi diverse caricate dalla vista "PERSONS_VIEW" e grazie all'attributo ***[ioClassFromField]*** ogni oggetto sarebbe della sua specifica classe inoltre, cosa secondo me interessante, se dopo avere fatto eventuali aggiunte/modifiche/cancellazioni su questa lista chiediamo a lupOrm di persisterla (poi vedremo come), ogni oggetto verrebbe correttamente persistito sulla rispettiva tabella e secondo il suo specifico tipo.

Nei casi esposti lupOrm restituisce sempre una nuova lista ma è possibile anche chiedere di riempire una lista già esistente passandola come parametro in questo modo:

TlupOrm.Load<TEmployee>.ToList(APersonList);

Load “ToListBindSourceAdapter” (vedi gli esempio “Pizza_List”)

Quando usiamo le funzionalità di LiveBindings, di solito, una delle prime cose che scriviamo è l’event handler dell’evento “onCreateAdapter” del BindSource utilizzato (TPrototypeBindSource, TAdapterBindSource ecc.) e scriveremmo una cosa del genere:

APersonList := TlupOrm.Load<TPerson>.ToList;
ABindSourceAdapter := TListBindSourceAdapter<TPerson>.Create(Self, APersonList);

Potremmo però chiedere a lupOrm di restituirci direttamente un TListBindSourceAdapter scrivendo quanto segue:

ABindSourceAdapter := TlupOrm.Load<TPerson>.ToListBindSourceAdapter;

Load “ToActiveListBindSourceAdapter” (vedi esempi terminano con “ioActiveBindSourceAdapter”)

Quando utilizziamo le funzionalità di LiveBindings, oltre a dover ricavare nell’onCreateAdapter del BindSource l’adapter appropriato di solito dobbiamo anche scrivere ulteriore codice, peraltro sempre abbastanza ripetitivo, per effettuare i salvataggi di eventuali cancellazioni e modifiche o nuovi inserimenti degli oggetti contenuti nella lista (Post, Delete, Refresh, Append/Insert ecc.), per non parlare poi se si devono utilizzare due BindSource in relazione Master-Detail tra loro.

lupOrm definisce al suo interno due classi “***TioActiveListBindSourceAdapter***” e “***TioActiveObjectBindSourceAdapter***” che implementano dei BindSourceAdapter contenenti già al loro interno la logica necessaria alle operazioni di persistenza, tramite lupOrm, degli oggetti contenuti. In pratica se nell’evento onCreateAdapter noi scrivessimo:

ABindSourceAdapter := TlupOrm.Load<TPerson>.ToActiveListBindSourceAdapter(Self);

otteniamo un ioActiveBindSourceAdapter con il quale non è necessario scrivere ulteriore codice riguardante la persistenza; quando chiederò il metodo ***Post, Delete, Append, Insert, Persist, Refresh*** del BindSource l’adapter saprà già cosa fare e non sarà necessario scrivere ulteriore codice. Se poi mettessimo sulla form anche un ***TBindNavigator*** non dovremmo nemmeno scrivere le chiamate ai metodi sopra elencati (Post, Delete, Refresh ecc.) perché vengono chiamati automaticamente dal TBindNavigator stesso, poi il BindSource li passa al sottostante ActiveBindSourceAdapter il quale effettuerà sempre automaticamente le corrette chiamate a lupOrm.

La cosa diventa ancora più interessante nel caso ci fosse la necessità di visualizzare su una form dei in relazione **Master-Detail**. Supponiamo di voler visualizzare due TListView, una con l’elenco di TPerson e l’altra contenente i numeri di telefono (TPhoneNumber) della persona selezionata (proprietà “Phones” di TPerson e discendenti di tipo TObjectList<TPhoneNumber>). Usando gli ActiveBindSourceAdapter di lupOrm metteremo ovviamente due TPrototypeBindSource nella form, nell’evento OnCreateAdapter del MasterBindSource scriveremo:

ABindSourceAdapter := TlupOrm.Load<TPerson>.ToActiveListBindSourceAdapter(Self);

nell'OnCreate del DetailBindSource invece scriveremo una riga di questo tipo:

ABindSourceAdapter := MasterBS.lupOrm.GetDetailBindSourceAdapter('Phones');

Con queste due sole righe di codice otteniamo i rispettivi ActiveBindSourceAdapters (Master e Detail) contenenti già tutta la logica necessaria sia per le operazioni di persistenza che per la gestione della relazione HasMany. Se usassimo anche in questo caso due TBindNavigator saremo in grado di scrivere una piccola e basica applicazione senza necessità di scrivere altro codice di presentazione oltre alle due righe sopra (vedremo poi come eliminare anche queste quando tratteremo il componente TioPrototypeBindSource).

A un ActiveBindSourceAdapter master possiamo chiedere tutti gli adapters dettaglio che vogliamo i quali, a loro volta, potrebbero avere ulteriori adapters di dettaglio e così via, tutto gestito automaticamente.

NB: I metodi "Append" e "Persist" e "lupOrm.GetDetailBindSourceAdapter" del TPrototypeBindSource sono aggiunti alla classe TPrototypeBindSource di Delphi da un ClassHelper.

Delete

Per eliminare un oggetto dal database sottostante con lupOrm useremo una riga di codice come questa:

TlupOrm.Delete(APerson);

E' possibile eliminare un singolo oggetto anche con la seguente istruzione grazie al ClassHelper per la classe TObject presente all'interno di lupOrm:

APerson.lupOrm.Delete;

In entrambi i casi verrà generata ed eseguita una query delete che eliminerà il record relativo all'oggetto APerson.

E' possibile anche eliminare un oggetto passando il suo ID come parametro:

TlupOrm.RefTo<TPerson>.ByOID(1).Delete;

La riga elimina dal database l'oggetto con ID = 1.

Possiamo anche eliminare più oggetti in una sola operazione:

TlupOrm.RefTo<TPerson>.Delete;

La riga elimina tutti i TPerson presenti nella tabella, nel caso invece volessimo eliminare, ad esempio, solo tutti i TPerson con il nome che comincia con la lettera "D" e maggiori di 18 anni possiamo scrivere la seguente istruzione:

```
TlupOrm.RefTo<TPerson>._Where._Property('Firstname')._LikeTo('%')._And._Property('Age')._GreaterThan(18).Delete;
```

Oppure in alternativa:

```
TlupOrm.RefTo<TPerson>._Where('[TPerson.FirstName] like "D%" and [TPerson.Age] > 18').Delete;
```

In entrambi i casi lupOrm provvederà poi a "tradurre" i riferimenti alle proprietà nei rispettivi riferimenti ai campi sottostanti nel database mantenendo quindi una totale astrazione tra il codice dell'applicazione e il database sottostante. In realtà nel secondo caso potremmo anche inserire riferimenti espliciti ai campi e alle tabelle del database sottostante perdendo però almeno parte del livello di astrazione di cui sopra.

Persist/PersistCollection e proprietà ObjStatus (vedi gli esempi "Pizza")

Possiamo persistere (Insert/Update/Delete) singoli oggetti oppure liste di oggetti con istruzioni tipo le seguenti:

```
TlupOrm.Persist(APerson);
```

Anche per il comando Persist è possibile usufruire dei servizi dati dei ClassHelpers di lupOrm, la riga sopra potrebbe essere scritta anche così:

```
APerson.lupOrm.Persist;
```

in entrambi i casi verrà effettuato l'insert/update di un singolo oggetto.

Se si necessita di persistere una lista di oggetti scriveremo:

```
TlupOrm.PersistCollection(APersonList);
```

che persiste tutti gli oggetti della lista passata come parametro.

lupOrm capisce se deve fare una operazione Insert oppure Update verificando il valore della proprietà che funge da ID della classe: se tale proprietà contiene il valore 0 (zero che è considerato come null) esegue una Insert e carica il nuovo ID (generato automaticamente) sulla proprietà stessa, altrimenti esegue un Update.

Il comando "Persist" opera in modo diverso in base all'esistenza o meno di una proprietà di nome "**ObjStatus**" di tipo "TlupOrmObjectStatus" nella classe sulla quale si sta operando e può assumere i seguenti valori "osClean", "osDirty", "osDeleted". La presenza di questa proprietà permette in pratica di "marcare" un oggetto (singolo oppure in una lista) come nuovo/modificato oppure eliminato senza persistere immediatamente tali cambiamenti sul database, in questo modo

infatti tutte le operazioni di persistenza dei cambiamenti avvenuti saranno posticipati a quando verrà esplicitamente eseguita l'istruzione **Persist** sul singolo oggetto o sulla lista; lupOrm eseguirà automaticamente tutte le query necessarie a riportare tali cambiamenti sul database all'interno di un'unica transazione. Ovviamente gli oggetti con `ObjStatus = osClean` non genereranno alcuna operazione (query) di persistenza aumentando enormemente, di conseguenza, l'efficienza dell'applicazione.

Gli **ActiveBindSourceAdapters** esposti precedentemente gestiscono automaticamente la proprietà **ObjStatus** se presente e alla chiamata dei metodi **Post** o **Delete** valorizzano opportunamente tale proprietà; alla successiva esecuzione del metodo **Persist** tutte le variazioni saranno persistite.

TioPrototypeBindSource

(vedi vari esempi forniti che terminano con "ioPrototypeBindSource")

All'interno di lupOrm esiste anche un componente TioPrototypeBindSource (sezione lupOrm della ToolPalette) che oltre alle normali caratteristiche del componente standard da cui deriva aggiunge alcune proprietà e metodi che ci consentono di ottenere cose piuttosto interessanti e comode. Ecco una lista:

Proprietà:

- **ioClassName: String;** in questa proprietà inseriremo il nome della classe di cui si vogliono visualizzare/persistere gli oggetti.
- **ioWhere: TStrings;** ci dà la possibilità di inserire delle clausole di filtraggio (where) nel caso necessitassimo di lavorare solo su un sottoinsieme di record presenti che soddisfano determinate condizioni; si utilizza normale linguaggio SQL integrato da speciali tag in cui possiamo fare riferimento ai nomi di classe e proprietà (es: [TPerson.Age]) tra parentesi quadre, in questo il codice scritto astrae dai dettagli implementativi del database sottostante. Ecco un esempio di clausola where: [TPerson.FirstName] like 'D%' and [TPerson.Age] > 18'
- **ioMasterBindSource: TioBaseBindSource;** Nel caso necessitassimo di due (o più) BindSource in relazione Master-Detail tra di loro, imposteremo questa proprietà nel DetailBindSource con il riferimento al BindSource che funge da Master.
- **ioMasterPropertyName: String;** Sempre in caso di due BindSource in relazione Master-Detail, inseriremo in questa proprietà il nome della proprietà master della classe master esposta dal MasterBindSource di cui sopra.

Metodi:

- **Persist(ReloadData: Boolean = False);** Questo metodo viene usato solo se la classe che si sta trattando possiede la proprietà "**ObjStatus**" e, analogamente agli ActiveBindSourceAdapters esposti in precedenza, persiste le eventuali variazioni avvenute in precedenza agli oggetti (Insert/Update/Delete) in un'unica operazione e transazione; anche in questo caso gli oggetti con ObjStatus = osClean non genereranno alcuna operazione (query) di persistenza aumentando enormemente, l'efficienza dell'applicazione. Il componente utilizza al suo interno gli ActiveBindSourceAdapters esposti precedentemente, è quindi in grado di gestire automaticamente la proprietà **ObjStatus** se presente e alla chiamata dei metodi **Post** o **Delete** valorizza opportunamente tale proprietà; alla successiva esecuzione del metodo **Persist** tutte le variazioni saranno persistite. Se il parametro **ReloadData** = True, dopo aver completato tutte le operazioni necessarie a persistere gli oggetti effettuerà un Reload + Refresh completo degli oggetti dal Database aggiornato.

Utilizzando il componente **TioPrototypeBindSourceAdapter** al posto del TPrototypeBindSource di Delphi non sarà più necessario scrivere alcuna riga di codice per l'evento **OnCreateAdapter** al fine di recuperare il BindSourceAdapter in quanto TioPrototypeBindSourceAdapter è in grado di gestire tutto automaticamente, l'unica cosa indispensabile è inserire nella proprietà **ioClassName** il nome della classe da esporre e tutto verrà gestito automaticamente.

Impostando opportunamente le proprietà **ioMasterBindSource** e **ioMasterPropertyName** in un eventuale TioPrototypeBindSource detail, anch'esso ricaverà automaticamente dal Master BindSource l'ActiveBindSourceAdapter di cui necessita già correttamente impostato per gestire in modo completamente automatico le relazioni Master-Detail e le funzioni di persistenza senza scrivere una sola istruzione.

Unendo quindi le caratteristiche del componente `TioPrototypeBindSource` e dei sottostanti `ActiveBindSourceAdapters` otteniamo una gestione completamente automatica sia delle funzionalità di persistenza che della relazione Master-Detail, in pratica potremmo realizzare una piccola applicazione, anche con relazioni Master-Detail, senza scrivere alcuna riga di codice riguardante la logica di presentazione.

Un altro fattore positivo dell'utilizzo di questo componente è che, nel codice della form (view), non è più necessario alcun riferimento diretto al Model, infatti nella relativa sezione "uses" della MainForm degli esempi forniti con `IupOrm` relativi al `TioPrototypeBindSource` non c'è alcun riferimento ad alcuna unit del Model. Il riferimento alla classe model di cui vengono esposti gli oggetti è solo una proprietà string.

Filtri e clausole where

Nelle operazioni di Load e Delete è possibile specificare delle condizioni per limitare il raggio d'azione dell'operazione. Nella parte centrale di una riga di codice che invoca i servizi di lupoOrm si possono specificare tutte le condizioni che si vogliono in due modi:

- 1) *Utilizzando una serie di funzioni (`_Where`, `_Property`, `EqualTo` ecc.) il cui nome comincia sempre con il carattere “_”, in questo modo durante la stesura del codice digitando tale carattere l'editor di Delphi ci mostrerà l'elenco di tutte le funzioni possibili riguardanti appunto le clausole di cui stiamo parlando. Tali funzioni danno la possibilità di fare riferimenti a proprietà, operatori logici e di comparazione. Un esempio potrebbe essere **`TlupoOrm.Load<TPerson>._Where._Property('Age')._GreaterOrEqual(18).ToList`**; come si vede la parte centrale della riga contiene le clausole di restrizione.*
- 2) *Scrivendo le condizioni in normale linguaggio SQL integrato da speciali tags che permettono di fare riferimento alle classi e loro proprietà e non alle tabelle e ai campi del database sottostante. Grazie a questi tags (“`[]`”) quindi è possibile mantenere un certo livello di astrazione dai dettagli implementativi del RDBMS. Un esempio potrebbe essere: **`TlupoOrm.Load<TPerson>._Where(['TPerson.Age'] > 18 and [TPerson.City] = "London")`**.`ToList`; come si vede i riferimenti a `Classe.Proprietà` devono essere scritti tra parentesi quadre. In questo secondo caso è anche possibile fare riferimenti diretti ai campi e tabelle sottostanti (basta scriverli normalmente, non tra parentesi quadre) diminuendo in questo caso il livello di astrazione. Questo modo di scrivere le condizioni di filtraggio è anche quello utilizzato dal componente **`TioPrototypeBindsource`** nella sua proprietà **`ioWhere`**.*