



# EntityDAC

ORM e supporto LINQ per Delphi



# CLAUDIO PIFFER

PC SOFT

CLAUDIO.PIFFER @GMAIL.COM



# AGENDA

## → Introduzione

- Panoramica di EntityDAC
- Edizioni e compatibilità
- Database supportati
- Lista componenti

## → Concetti base di EntityDAC

- Code-first Database-first Model-first
- Entity Developer
- Code Mapped Entities
- Attribute-mapped entities
- XML-mapped entities
- Attribute-mapped objects
- Operazioni CRUD (Create, Read, Update, Delete)
- Entity
- Entity customization

## → Esecuzione delle query

- Utilizzo del linguaggio LINQ

## → Ottimizzazione delle prestazioni

- Utilizzo della cache dei dati

## → Utilizzo delle transazioni con EntityDAC

## → SQL Executing

# Introduzione: Panoramica di EntityDAC

- ➔ EntityDAC è un framework ORM (Object-Relational Mapping) per Delphi che semplifica l'accesso ai dati.
- ➔ Mapping oggetto-relazionale: EntityDAC mappa automaticamente le classi delle tue entità nel database, consentendo di lavorare con i dati come oggetti fortemente tipizzati. Supporta il mapping di entità complesse, relazioni uno-a-uno, uno-a-molti e molti-a-molti.
- ➔ Query LINQ: EntityDAC supporta LINQ (Language INtegrated Query), che consente di eseguire query complesse sui dati utilizzando un linguaggio simile a SQL direttamente in codice Delphi. Ciò semplifica la scrittura delle query, validazione a compile time e rende il codice più leggibile e manutenibile.
- ➔ Persistenza dei dati: EntityDAC offre metodi per creare, leggere, aggiornare ed eliminare (CRUD) le entità nel database. Supporta transazioni per garantire l'integrità dei dati e la coerenza delle operazioni.
- ➔ Supporto multiplatforma: EntityDAC è compatibile con diverse versioni di Delphi e funziona su tutte le piattaforme: Windows, macOS, Linux, Android e iOS.

# Introduzione: edizioni e compatibilità

- EntityDAC è disponibile in tre edizioni: Express , Standard e Professional .
  - Express Edition è una versione gratuita di EntityDAC che include i provider di dati Standard e Devart e alcune delle funzionalità di EntityDAC per la valutazione.
  - Standard Edition è una soluzione conveniente per gli sviluppatori alla ricerca di un ORM ad alte prestazioni e ricco di funzionalità per Delphi.
  - La Professional Edition estende la Standard Edition con diverse importanti funzionalità in fase di progettazione e componenti data-aware.
  - Puoi ottenere l'accesso al codice sorgente di tutte le classi di componenti in EntityDAC acquistando la speciale EntityDAC Professional Edition con codice sorgente
- <https://www.devart.com/entitydac/editions.html>
- EntityDAC è installabile da Delphi 2007 in poi e supporta tutte le edizioni (Community/Professional/Enterprise/Architect)

# Introduzione: database supportati

→ EntityDAC supporta i seguenti database

→ SQL Server 2000 e versioni successive

→ MySQL 4.1 e versioni successive

→ Oracle 9 e versioni successive

→ PostgreSQL 8 e versioni successive






→ SQLite 3

→ Firebird 2 e versioni successive

→ DB2 9.5 e versioni successive



# Introduzione: lista componenti

## → Componenti base

	EntityConnection	Consente di configurare e controllare le connessioni a diversi server. Utilizzato anche per il controllo delle transazioni sulle sessioni e per l'esecuzione di query SQL su un database.
	TEntityXMLModel	Rappresenta il meta-modello in fase di progettazione. Utilizzato per configurare i componenti del set di dati EntityDAC, come TEntityTable e TEntityQuery.
	TEntityContext	Gestisce le entità. Utilizzato per creare, aggiornare ed eliminare entità, recuperare e archiviare entità da/al database, archiviare entità utilizzate nella cache per uso futuro, distruggere entità non utilizzate.
	TEntityDataSet	Mantiene i dati da una fonte arbitraria. Può contenere una singola entità o un elenco di entità. Può essere utilizzato solo in fase di esecuzione.
	TEntityDataSource	Fornisce un'interfaccia per la connessione di controlli in grado di riconoscere i dati in un modulo e componenti del set di dati EntityDAC.

# Introduzione: lista componenti












→ Componenti aggiuntivi edizione professionale

	EntityTable	Consente di recuperare e aggiornare le entità del singolo metatipo senza scrivere istruzioni LINQ.
	EntityQuery	Utilizza le istruzioni LINQ per recuperare le entità dalle tabelle del database e passarle a uno o più componenti data-aware tramite un oggetto TDataSource. Questo componente fornisce un meccanismo per l'aggiornamento dei dati.



# Introduzione: lista componenti

## → Provider di accesso ai dati

	<a href="#">TUniDACDataProvider</a>	Collega il fornitore di dati per Devart Universal Data Access Components a un'applicazione.	
	<a href="#">TODACDataProvider</a>	Collega il fornitore di dati per Devart Oracle Data Access Components a un'applicazione.	
	<a href="#">TSDACDataProvider</a>	Collega il provider di dati per Devart SQL Server Data Access Components a un'applicazione.	
	<a href="#">TMyDACDataProvider</a>	 <a href="#">TADODataProvider</a>	Collega il provider di dati per i componenti ADO a un'applicazione.
	<a href="#">TIBDACDataProvider</a>	 <a href="#">TDBXDataProvider</a>	Collega il fornitore di dati per i componenti dbExpress a un'applicazione.
	<a href="#">TPgDACDataProvider</a>	 <a href="#">Fornitore di dati TIBX</a>	Collega il fornitore di dati per i componenti di InterBase Express a un'applicazione.
	<a href="#">TLiteDACDataProvider</a>	 <a href="#">TFireDACDataProvider</a>	Collega il fornitore di dati per FireDAC a un'applicazione.

# Concetti base di EntityDAC

# Code-first Database-first Model-first

Esistono diversi approcci allo sviluppo di applicazioni di database.

## → Code-First

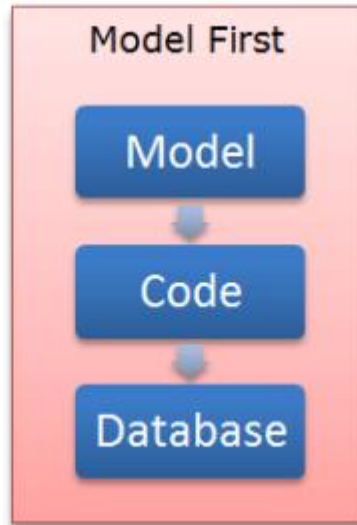
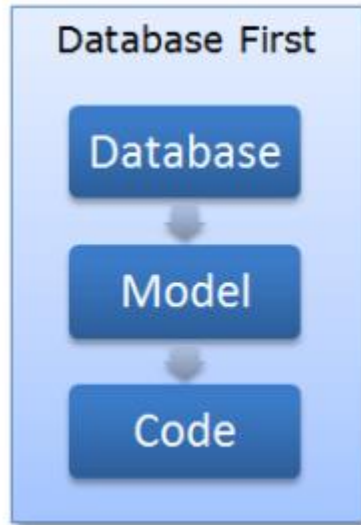
di dati. Il codice non corrisponde alle modifiche del modello.

## → Database-First

usa Entity Framework.

## → Model-First

del modello.



ano il modello  
sistente che  
nere traccia

se e quindi si

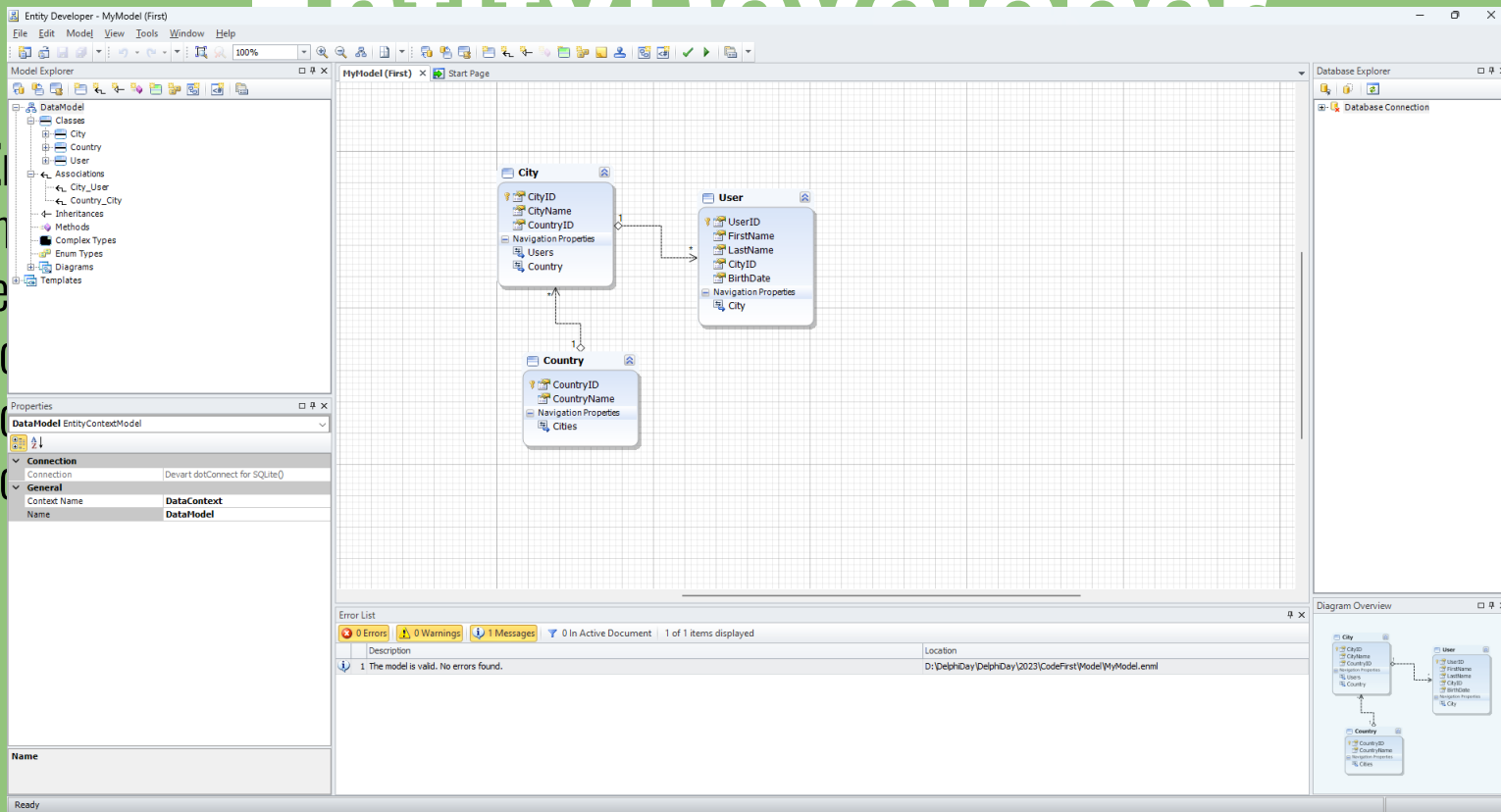
n la definizione  
ra

# Entity Developer



Entity Developer  
model  
le  
co  
co

creare  
entità,  
entità  
ORM,  
ente il



# Code Mapped Entities

→ Per ogni entità del database verrà generata una classe `TMappedEntity` completamente gestita da un contesto dati. I metadati del modello vengono generati in un'unità separata come un insieme di speciali classi di "metadati" collegate tramite codice alle classi di entità corrispondenti.

# Attribute-mapped entities

→ Per questo tipo di modello, verranno generate solo le classi di entità mentre le classi di metadati del modello non vengono generate. Le classi di entità sono contrassegnate con speciali attributi e i metadati verranno generati automaticamente in fase di esecuzione utilizzando questi attributi.

# XML-mapped entities

- Verranno generate solo le classi di entità. I metadati del modello devono essere specificati come file XML esterno che può essere generato utilizzando Entity Developer o creato manualmente.

# Attribute-mapped objects

→ Questo tipo di modello è simile al modello "Attribute-mapped entities", ma le classi generate non sono discendenti da TEntity, ma da TObject.



# Entity: create

→ Creazione di una entity:

```
var  
  Emp: TEmp;  
begin  
  // create new entity  
  Emp := TEmp.Create;  
end;
```

```
var  
  Emp: TEmp;  
begin  
  // create new entity with the specified primary key value  
  Emp := TEmp.Create([1]);  
end;
```

```
var  
  Context: TEntityContext;  
  Emp: TEmp;  
begin  
  // create and initialize the data context  
  // ...  
  // create new entity with the specified primary key value  
  Emp := Context.CreateEntity<TEmp>([1]);  
end;
```

# Entity: attach

→ Attach di una entity

```
var
  Context: TEntityTypeContext;
  Emp: TTemp;
begin
  // create and initialize the data context
  // ...
  // create new entity
  Emp := TTemp.Create;
  // set the entity primary key
  Emp.Empno.AsInteger := 1;
  // attach the entity
  Emp.Attach(Context);
end;
```

```
var
  Context: TEntityTypeContext;
  Emp: TTemp;
begin
  // create and initialize the data context
  // ...
  // create new entity
  Emp := TTemp.Create;
  // set the entity primary key
  Emp.Empno.AsInteger := 2;
  // attach the entity
  Context.Attach(Emp);
end;
```

# Entity: attach

→ Nel caso in cui la chiave primaria può essere determinata in fase di creazione o manualmente o direttamente dall'entità (campo identity o collegato ad una sequence) si può creare l'entity direttamente dal context (quindi già attached)

```
var
    Context: TEntityContext;
    Emp: TEmp;
begin
    // create and initialize the data context
    // ...
    // create attached entity with the specified primary key value
    Emp := Context.CreateAttachedEntity<TEmp>([1]);
end;
```

# Entity: get

→ Una entity può essere recuperata in vari modi:

```
var
  Context: TEntityContext;
  Emp: TTemp;
begin
  // create and initialize the data context
  // ...
  // get single entity by the condition
  Emp := Context.GetEntity<TTemp>('en');
end;
```

```
var
  Context: TEntityContext;
  Emp: TTemp;
begin
  // create and initialize the data context
  // ...
  // get single entity by the condition
  Emp := Context.GetEntity<TTemp>('en');
end;
```

```
var
  Context: TEntityContext;
  Query: ILinqQueryable;
  Emp: TTemp;
begin
  // create and initialize the data context
  // ...
  // create the query
  Query := Linq.From(Context['Emp'])
    .Where(Context['Emp']['Empno'] = 1)
    .Select;
  // get single entity by the query
  Emp := Context.GetEntity<TTemp>(Query);
end;
```

In tutti questi casi le entity recuperate sono sempre collegate al context quindi non ci dobbiamo preoccupare del loro rilascio

# Entities: get

→ Per recuperare una lista di entity, EntityDAC fornisce `IEntityEnumerable`. Esempi:

```
var
  Context: TEntityContext;
  List: IEntityEnumerable<TEmp>;
  Emp: TTemp;
  i: integer;
begin
  // create and initialize
  // ...
  // get a whole list of T
  List := Context.GetEntities<TEmp>();
end;
```

```
var
  Context: TEntityContext;
  List: IEntityEnumerable<TEmp>;
  Emp: TTemp;
  i: integer;
begin
  // create and initialize the data context
  // ...
  // get the list by the condition
  List := Context.GetEntities<TEmp>(i);
end;
```

```
var
  Context: TEntityContext;
  Query: IQueryable;
  Emp: TTemp;
begin
  // create and initialize the data context
  // ...
  // create the query
  Query := Linq.From(Context['Emp'])
    .Where(Context['Emp']['Empno'] > 1)
    .Select;
  // get the list by the query
  Emp := Context.GetEntities<TEmp>(Query);
end;
```

# Entities: accesso

→ L'interfaccia IEntityEnumerable discende da IEnumerable e quindi possiamo iterare le nostre liste in questo modo:

```
for Emp in List do begin
    // do something
    // ...
end;
```

# Entities: save

- La modifica delle proprietà delle entity non corrisponde ad una modifica verso il DB. Per persistere le modifiche al DB è necessario chiamare il metodo Save

```
var
    Context: TEntityContext;
    Emp: TEmp;
begin
    // creation and initialization of the context
    // ...
    Emp := TEmp.Create;
    // set the unique value to the entity primary key
    Emp.Empno.AsInteger := 1;
    // attach the entity
    Emp.Attach(Context);
    // save the entity in the database
    Emp.Save;
end;
```

```
var
    Context: TEntityContext;
    Emp: TEmp ;
begin
    // creation and initialization of the context
    // ...
    Emp := TEmp.Create;
    // set the unique value to the entity primary key
    Emp.Empno.AsInteger := 1;
    // attach the entity
    Context.Attach(Emp);
    // save the entity in the database
    Context.Save(Emp);
end;
```

# Entities: delete

- La cancellazione di un'entità è un processo in due fasi. Poiché tutte le entità sono archiviate nel context, l'entità deve prima essere eliminata da essa. Quindi, per applicare la cancellazione in base, l'entità deve essere salvata.

```
var
  Context: TEntityContext;
  Emp: TEmp;
begin
  // create and initialize the data context
  // ...
  // get single entity by the primary key
  Emp := Context.GetEntity<TEmp>([1]);
  // delete entity from the cache
  Emp.Delete;
  // apply deletion in the database
  Emp.Save;
end;
```

```
var
  Context: TEntityContext;
  Emp: TEmp ;
begin
  // create and initialize the data context
  // ...
  // get single entity by the primary key
  Emp := Context.GetEntity<TEmp>([1]);
  // delete entity from the cache
  Context.Delete(Emp);
  // apply deletion in the database
  Context.Save(Emp);
end;
```



# Entities: cancel changes

→ Come descritto in precedenza, tutte le operazioni di modifica con un'entità (modifica, eliminazione) devono essere confermate (salvate) per persisterle nel database.

```
var
  Context: TEntityContext;
  Emp: TEmp;
begin
  // create and initialize the data context
  // ...
  // get single entity by the primary key
  Emp := Context.GetEntity<TEmp>([1]);
  // change the entity property
  Emp.Ename.AsString := 'new name';
  // cancel changes
  Emp.Cancel;
end;
```

Se l'oggetto non è stato salvato è difficile

```
var
  Context: TEntityContext;
  Emp: TEmp ;
begin
  // create and initialize the data context
  // ...
  // get single entity by the primary key
  Emp := Context.GetEntity<TEmp>([1]);
  // change the entity property
  Emp.Ename.AsString := 'new name';
  // cancel changes
  Context.Cancel(Emp);
end;
```

# Entities: submit changes

- Nel mondo reale, la situazione più comune è quella in cui molti oggetti devono essere salvati contemporaneamente, ed l'esecuzione del salvataggio per ciascuno di essi non è adatto (per esempio, quando diversi oggetti correlati devono essere salvati contemporaneamente). In questo caso, il metodo speciale `SubmitChanges` viene utilizzato per salvare le modifiche massicce.

```
var
    Context: TEntityContext;
begin
    // create and initialize the data context
    // ...
    // making changes to entities
    // ...
    // submit all changes
    Context.SubmitChanges;
end;
```

# Entities: reject changes

- Contrariamente al metodo precedente, il metodo `RejectChanges` esegue l'azione opposta. Annulla tutte le modifiche apportate alle entità contenute nel `Context`.

```
var
    Context: TEntityContext;
begin
    // create and initialize the data context
    // ...
    // making changes to entities
    // ...
    // reject all changes
    Context.RejectChanges;
end;
```

# Entity Customization

- E' possibile customizzare il comportamento di una entity in due modi:
  - Class Helper
  - Ereditarietà

# LINQ

- Linq abbreviazione di «**L**anguage **IN**tegrated **Q**uery»
- Con Linq è possibile effettuare interrogazioni su classi «enumerabili», collection utilizzando una sintassi simile al linguaggio SQL

# LINQ

```
var
    D: IDeptExpression;
    Query: ILinqQueryable;
begin
    D := Context.Dept;

    Query := Linq.From(D)
               .Select;
end;
```

# Cache

→ Per aumentare le prestazioni dell'applicazione, EntityDAC consente di memorizzare nella cache i metadati, tutte le entità caricate dal database, le query LINQ e molto altro. Tale memorizzazione nella cache consente di evitare il caricamento multiplo degli stessi dati e di migliorare notevolmente le prestazioni rispetto all'utilizzo dei componenti di accesso ai dati standard.

# Cache: disabilitazione

→ La cache è abilitata per default. Per disabilitarla basta invocare:

```
begin  
  Result := TDataContext.Create(nil);  
  Result.Connection := EntityConnection;  
  Result.Options.Cache.Enabled := False;  
end;
```



# Transazioni

→ Poiché TEntityConnection fornisce metodi per accedere al database e implementare alcune funzionalità, sono disponibili alcune transazioni. I metodi StartTransaction e RollbackTransaction controllano le transazioni.

```
var
    Connection: TEntityConnection;
begin
    // create and initialize the connection
    // ...
    // begin the transaction
    Connection.StartTransaction;
    try
        // ...
        // commit the transaction
        Connection.CommitTransaction;
    except
        // rollback the transaction in case of an error
        Connection.RollbackTransaction;
    end;
end;
```

# SQL Executing

→ TEntityConnection espone anche il metodo ExecuteSQL che permette di eseguire codice SQL verso il database.

```
var
  Connection: TEntityConnection;
begin
  // create and initialize the connection
  // ...
  // execute the simple SQL statement
  Connection.ExecuteSQL('insert into EMP(ENAME) values(''Sample'')');
end;
```

# SQL Executing

```
// add necessary units to be able to use the TDBParams class
uses
    SQLDialect, EntityTypes;

var
    Connection: TEntityConnection;
    Params: TDBParams;

begin
    // create and initialize the connection
    // ...
    // create and fill query parameters
    Params := TDBParams.Create;
    try
        with Params.Add do begin
            Name := 'ret_param';
            ParamType := ptOutput;
            DataType := dbInteger;
        end;
        Connection.ExecuteSQL ('insert into EMP(ENAME) values(''Sample'') returning EMPNO into :ret_param', Params);
        ShowMessage('Return = ' + Params[0].Value.AsString);
    finally
        Params.Free;
    end;
end;
```

# Stored procedure

→ Se il DB  
delle stored

seguire

```
uses
    SQLDialect, EntityTypes;

var
    Connection: TEntityConnection;
    Params: TDBParams;

begin
    // create and initialize the connection
    // ...
    // create and fill procedure parameters
    Params := TDBParams.Create;
    try
        with Params.Add do begin
            Name := 'proc_param';
            ParamType := ptInput;
            DataType := dbInteger;
        end;
        Connection.ExecuteStoredProc('some_procedure', Params);
    finally
        Params.Free;
    end;
end;
```

# SQL script

→ Per eseguire una serie di istruzioni SQL

sequenzialmente.

ExecuteScript

come parametro

non può

re passato

gs. Lo script

```
var
    Connection: TEntityConnection;
    Script: TStrings;
begin
    // create and initialize the connection
    // ...
    // create and fill the script
    Script := TStringList.Create;
    try
        Script.Add('insert into EMP(ENAME) values(''Sample'');');
        Script.Add('insert into EMP(ENAME) values(''Sample 1'');');
        Connection.ExecuteScript(Script);
    finally
        Script.Free;
    end;
end;
```

# Campi blob

→ EntityDAC supporta tutti i tipi di campi, compreso i campi blob e clob.

```
LReportStream := DecodeBase64(AReport.Report.DocumentBase64);  
try  
  var LReportBlob: TBlobData;  
  SetLength(LDataReport, LReportStream.Size);  
  LReportStream.ReadBuffer(LDataReport[0], LReportStream.Size);  
  LReportBlob.AsBytes := LDataReport;  
  LReport.Report := LReportBlob;  
finally  
  LReportStream.Free;  
end;
```



**THANK YOU**