Document Orientation

Indice generale

Document Orientation

La Document Orientation (DO) è il framework ORM di Instant Developer Cloud. Scopri come gestire i dati con classi e oggetti.

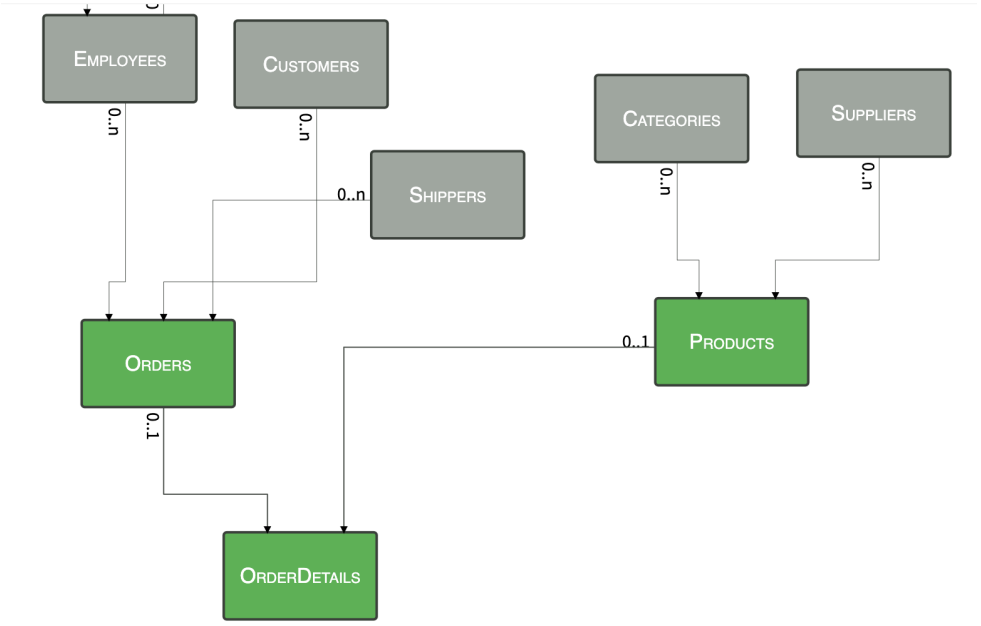
Introduzione

Nei capitoli precedenti abbiamo visto che l'accesso ai dati del database avviene tramite query SQL che restituiscono recordset, mentre la modifica degli stessi avviene tramite l'esecuzione di comandi SQL.

Questa modalità è efficace soprattutto quando si devono estrarre dal database situazioni riassuntive, che richiedono la costruzione di query complesse con molti *join* tra tabelle. In molti altri casi, invece, l'uso di SQL e recordset porta ad una minore efficienza del ciclo di sviluppo perché non permette di operare sui dati in modalità Object Oriented. Per ovviare a questo problema si utilizza la tecnica Object Relational Mapping.

La difficoltà accennata prima viene accentuata quando i dati su cui operare sono complessi, cioè derivano dalla composizione di dati semplici. Ad esempio, nella seguente struttura dati possiamo evidenziare tre tabelle:

1) *Orders*: contiene i dati di testata degli ordini di acquisto.
2) *Order Details*: contiene i dati degli articoli venduti in ogni ordine.
3) *Products*: contiene l'elenco dei prodotti che è possibile vendere.



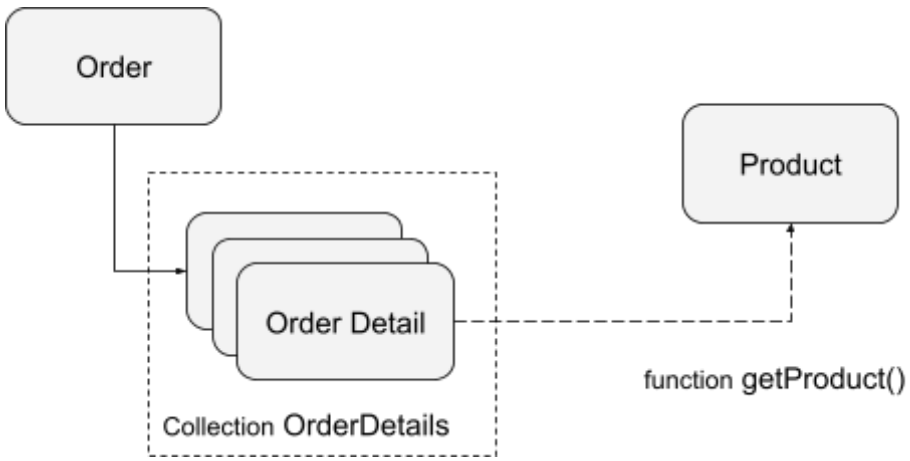
Fra queste tabelle esistono anche delle relazioni. In particolare fra *Orders* e *OrderDetails* esiste una relazione che collega ogni riga di un ordine alla sua testata; essa esprime un concetto di *possesso*: ogni testata ordine possiede le sue righe; se una testata venisse eliminata, anche le relative righe dovrebbero subire la medesima sorte.

Anche fra *Products* e *OrderDetails* esiste una relazione, che esprime un concetto diverso: ogni riga ha bisogno di identificare il prodotto venduto, quindi la relazione non è possessiva, ma identificativa. Se infatti una riga d'ordine venisse eliminata, il prodotto relativo rimarrebbe intatto. Se invece si volesse eliminare dal database un prodotto che è stato venduto, tale operazione dovrebbe essere impedita per preservare il legame con le righe degli ordini in cui esso è stato venduto.

Qual è il modo più immediato per lavorare su strutture dati di questo genere? Mappare tali strutture su una gerarchia di classi con cui lavorare in modalità *object oriented*. Nell'esempio precedente potremmo avere tre classi:

1) La classe *Order* che gestisce un intero ordine (testata e righe).
2) La classe *Order Detail* che gestisce una riga di un ordine.
3) La classe *Product* che gestisce un prodotto.

Potremmo gestire anche le relative relazioni: un'istanza di *Order* deve poter gestire anche le proprie righe, quindi la classe *Order* avrà una proprietà di tipo *Collection* per contenerle. Una istanza di *Order Detail* avrà bisogno di accedere ai dati del prodotto a cui essa si riferisce; in questo caso dovrà avere un metodo da richiamare per ottenerlo. Schematicamente potremmo avere la seguente gerarchia di classi:



Il framework Document Orientation (DO) di Instant Developer Cloud nasce proprio per rendere possibile lavorare sui dati presenti nel database relazionale in modalità object oriented.

Ma perché Document Orientation? Il nome deriva da un'accezione del termine documento che è propria di questo framework. La definizione del termine documento nell'ambito del framework DO è pertanto di fondamentale importanza per la comprensione del suo funzionamento.

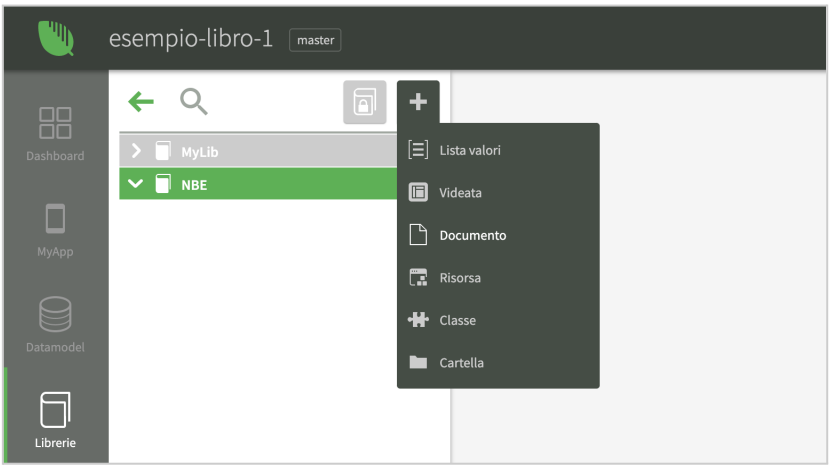
I documenti sono classi JavaScript in cui vengono mappate le tabelle di un database e che ne gestiscono i record. Un'istanza di queste classi è chiamata documento. I documenti possono mappare le relazioni possessive tra le tabelle del database tramite collection, e possono quindi gestire strutture multilivello. Tramite i documenti è quindi possibile rappresentare qualunque documento gestionale delle proprie applicazioni.

Il framework DO prende in carico la gestione del caricamento e salvataggio dei dati nel database non record per record, ma dell'intera struttura multilivello. Gestisce inoltre il ciclo di vita dei documenti, facilita il collegamento con il front-end e consente la sincronizzazione dati tra dispositivi e backend cloud.

Per queste ragioni, i documenti sono un componente fondamentale di ogni progetto Instant Developer Cloud.

Definire Documenti e Collection

Un documento è una classe del progetto Instant Developer Cloud che estende la classe base *Application.Document*. È possibile creare documenti sia a livello di applicazione che di libreria, ma siccome essi servono per gestire i dati del database, si consiglia di inserirli in una libreria in modo che siano utilizzabili da ogni applicazione presente nel progetto.



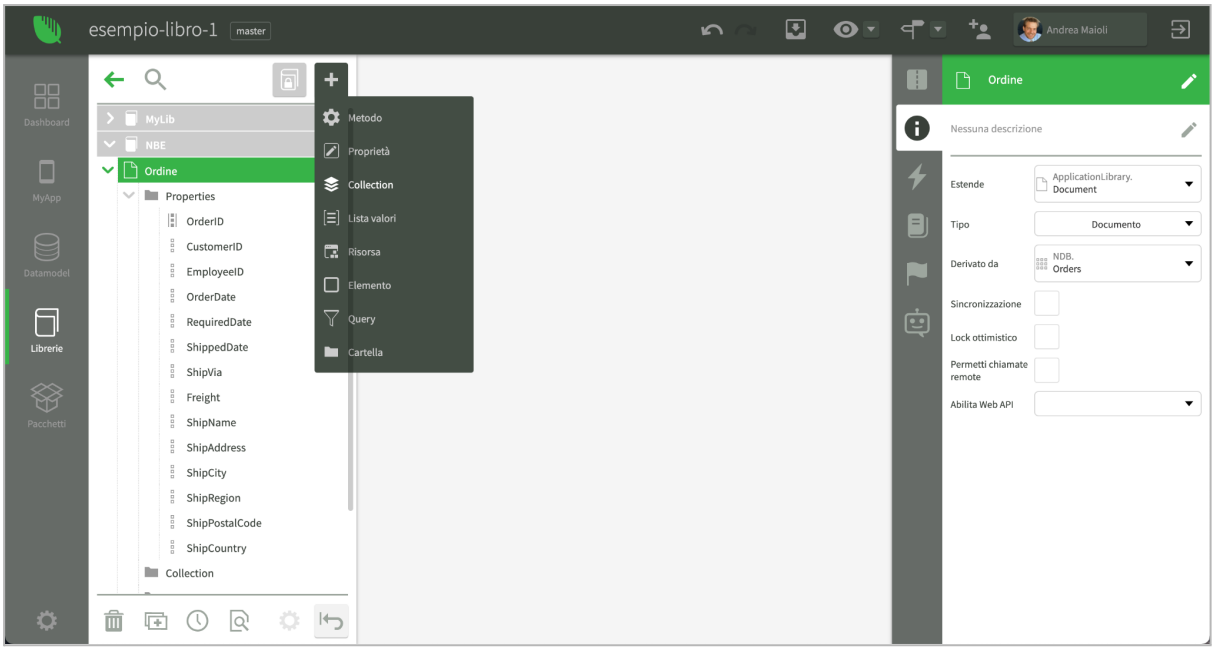
Creazione di un documento in una libreria

Dopo aver creato e nominato il documento è importante impostare la proprietà *Derivato da* scegliendo la tabella del database che tale documento deve gestire. A questo punto nel documento apparirà una cartella chiamata *Properties* in cui sono state inserite le proprietà di classe relative ad ogni campo della tabella.

Per aggiungere le collection, è sufficiente utilizzare il comando *+Collection* del menu relativo all'oggetto documento, come si vede nell'immagine alla pagina seguente. Una collection è una proprietà di classe di tipo *Collection*, e per poterla collegare al database, occorre definire le seguenti proprietà:

1) *Tipo contenuto*: la classe di documenti contenuti nella collection.
2) *Collegato a*: la relazione del database da utilizzare per caricare la collection.
3) *Ordina per*: proprietà opzionale che indica come comporre la clausola *order by* della query di caricamento.

Riprendendo l'esempio precedente, volendo inserire nel documento *Ordine* la collection delle righe, occorre prima definire il documento *RigaOrdine* relativo alla tabella *OrderDetails*, poi è possibile aggiungere al documento *Ordine* una collection di nome *righeOrdine*, che contiene documenti di tipo *RigaOrdine*, collegata alla relazione *OrdersFK*.



Il documento Ordine a cui sta per essere aggiunta una collection

Un metodo più veloce per definire documenti e collection è quello di derivarli direttamente dalla struttura del database, tramite il pulsante *Crea documenti* che appare nella parte centrale dell'IDE selezionando l'oggetto database nell'albero del progetto. Dopo averlo cliccato apparirà un elenco di tabelle per cui è possibile creare i documenti.

La generazione dei documenti avviene come segue:

1) Per ogni tabella selezionata viene generato un documento con lo stesso nome della tabella. Esso viene posizionato in una libreria che contiene già documenti derivati dallo stesso database; se una tale libreria non esiste, ne viene creata una nuova. Se esiste già un documento che deriva dalla stessa tabella, tale tabella non apparirà nell'elenco di quelle selezionabili per la generazione di documenti.
2) Per ogni documento vengono generate le proprietà di classe relative a tutti i campi della tabella. Inoltre se la tabella ha relazioni di tipo possessivo con altre tabelle, vengono generate anche le relative collection. Una relazione è definita come possessiva se la regola di cancellazione è *cascade*.

Al termine dell'operazione di generazione è possibile rinominare a piacere gli oggetti creati (librerie, documenti, proprietà e collection). Il modo consigliato è quello di dare ai documenti il nome della tabella in forma singolare per indicare che un'istanza del documento gestisce un singolo record di quella tabella; ad esempio il documento *Prodotto* gestirà i record della tabella *Prodotti*.

Proprietà unbound e proprietà derivate

Oltre alle proprietà derivate dai campi della tabella del database a cui il documento è collegato, come tutte le classi di codice, i documenti possono contenere altre proprietà di classe che non sono collegate al database.

Tali proprietà sono spesso chiamate *unbound*, proprio per indicare che non hanno un collegamento con la tabella del database.

Esiste un terzo tipo di proprietà del documento, chiamate proprietà *derivate*. Esse derivano da tabelle del database diverse da quella su cui è basato il documento. Un esempio di proprietà derivata può essere il nome del prodotto a cui una determinata riga ordine è collegata. Se non ci fosse la proprietà derivata, ogni volta che il documento riga ordine deve recuperare il nome del prodotto, dovrebbe fare una query.

Le proprietà derivate sono molto importanti perché i loro valori vengono recuperati automaticamente quando il documento viene caricato dal database nella stessa query che estrae il documento. Questo vale anche per il caricamento di collection di documenti: tutta la collection viene recuperata con un'unica query che estrae anche tutte le proprietà derivate di tutte le istanze in fase di caricamento. Si tratta di un'ottimizzazione importante che il framework DO mette a disposizione.

Per definire una proprietà derivata, occorre inserire una proprietà *unbound*, poi impostare la proprietà *Collegato a* selezionando la relazione della tabella da usare per il caricamento, infine impostare la proprietà *Derivato da* selezionando il campo della tabella di arrivo della relazione.

Esiste un modo più veloce per creare le proprietà derivate:

1) Nel documento da cui si desidera estrarre l'informazione, selezionare la proprietà e poi usare il comando *Copia (Ctrl-C)*.
2) Selezionare il documento in cui creare la proprietà derivata e usare il comando *Incolla (Ctrl-V).*

Viene in ogni caso creata una nuova proprietà, che sarà *derivata* se fra i due documenti esiste una relazione utilizzabile, altrimenti rimarrà *unbound*. Per distinguere i due casi è sufficiente osservare le proprietà *Collegato a* e *Derivato da* che rimarranno vuote nel caso *unbound*.

Si noti che una volta che è presente una proprietà derivata, sarà possibile derivare ulteriori proprietà anche usando le relazioni della tabella della prima proprietà derivata. Ad esempio se la classe *RigaOrdine* ha la proprietà derivata *NomeProdotto*, sarà possibile anche aggiungere la proprietà *NomeCategoria* che deriva dalla tabella delle categorie di prodotto.

Adattamento alla struttura del database

Le modifiche alla struttura del database si riflettono sui documenti collegati. In particolare:

1) Cancellando una tabella dal database i documenti collegati perdono il collegamento e non verranno più caricati dal database.
2) Cancellando un campo di una tabella, le proprietà collegate vengono cancellate.
3) Cancellando una relazione, le collection o le proprietà derivate perderanno il collegamento e non verranno più caricate automaticamente dal database.

Utilizzo dei documenti

Inizializzazione dei documenti

Di solito i documenti vengono caricati dal database, modificati e poi salvati. In questi casi le istanze di documento vengono restituite dai metodi di caricamento che verranno illustrati nel paragrafo seguente.

In altri casi, ad esempio quando si vuole creare un nuovo documento che dovrà essere inserito nel database, la fase di inizializzazione avviene a partire dal codice applicativo.

Per creare un'istanza di documento, si utilizza lo stesso costrutto di ogni altra classe di codice. Nell'esempio seguente viene istanziato un nuovo documento *Product*, classe contenuta nella libreria *NBE*.

```
var p = new App.NBE.Product(app);
```

Il costruttore dei documenti ammette, come secondo parametro, un oggetto che specifica le proprietà iniziali dell'oggetto. Si noti che impostare le proprietà in questo modo è più efficiente che farlo una alla volta sull'istanza. Ad esempio:

```
var p = new App.NBE.Product(app, {ProductName : "Pizza margherita"});
```

Ogni volta che viene istanziato un documento, sia dal codice applicativo che dal framework, viene chiamato l'evento *onInit* sulla classe del documento. Se si desidera scrivere codice specifico a livello di costruttore del documento, l'evento *onInit* è il posto giusto per farlo.

Stato rispetto al database

I documenti hanno quattro proprietà che ne specificano lo stato rispetto al database:

1) *loaded*: questa proprietà è vera se il documento è già stato caricato dal database. In tal caso, ulteriori istruzioni di caricamento verranno ignorate.
2) *updated*: questa proprietà è vera se almeno una proprietà del documento è stata modificata rispetto allo stato originale, che, solitamente, corrisponde allo stato in cui era il documento subito dopo il caricamento dal database.
3) *deleted*: questa proprietà è vera se il documento viene marcato per la cancellazione da parte del codice applicativo. La cancellazione dal database vera e propria avverrà al momento del salvataggio del documento.
4) *inserted*: questa proprietà è vera se il documento viene marcato per l'inserimento da parte del codice applicativo. L'inserimento vero e proprio nel database avverrà al momento del salvataggio del documento.

In un documento appena istanziato da codice, tutte queste proprietà sono *false*. Si noti che al variare delle proprietà *updated*, *deleted* ed *inserted* viene sempre chiamato sulla classe del documento il relativo evento. Di questi, l'evento più importante è l'evento *onInserting* che permette di specificare le proprietà di default di un documento marcato per l'inserimento.

Preparazione per l'inserimento

Vediamo infine un esempio di come preparare un documento per l'inserimento:

```
var p = new App.NBE.Product(app, {ProductName : "Pizza margherita"});
p.inserted = true;
yield p.save();
```

Dopo aver creato l'istanza ed eventualmente passato al costruttore i valori delle proprietà, il documento viene marcato per l'inserimento e poi salvato; quest'ultima operazione verrà dettagliata nei paragrafi seguenti.

Il cambiamento di valore della proprietà *inserted* da *false* a *true* causa le seguenti operazioni sul documento da parte del framework:

1) Se il documento contiene una proprietà che deriva da un campo primary key di tipo UUID, tale proprietà viene inizializzata usando un generatore di UUID valido per le lunghezze 20, 24 e 36. I campi di lunghezza 20 conterranno un UUID codificato in ASCII 85, quelli di lunghezza 24 useranno la codifica *base64* ed infine quelli 36 la codifica standard dei GUID. Si consiglia di utilizzare 24 per il massimo rapporto tra compatibilità ed efficienza.
2) Se per i campi del database da cui derivano le proprietà del documento era stato impostato un valore di default, tali valori vengono impostati sulle corrispondenti proprietà del documento se esse non sono state ancora definite.
3) Viene chiamato l'evento *onInserting* definito nella classe del documento.

Se, ad esempio, per un documento *Product* è necessario impostare delle proprietà di default ed esse non sono presenti a livello di database, si consiglia di implementare l'evento *onInserting* come si vede nell'esempio seguente:

```
App.NBE.Product.prototype.onInserting = function ()
{
  if (this.inserted) {
    if (this.SupplierID === undefined)
      this.SupplierID = app.SupplierDefault;
  }
};
```

Nell'esempio vediamo che il codice di inizializzazione viene condizionato al fatto che la proprietà *inserted* sia *true*. Questo è importante perché l'evento viene chiamato tutte le volte che la proprietà *inserted* varia, sia da *false* a *true*, ma anche da *true* a *false* e questo avviene subito dopo che il documento è stato salvato. Notiamo infine che il codice dell'evento *onInserting* può accedere ai dati di sessione tramite la variabile *app*; in questo modo è possibile generare valori di default che dipendono dallo stato della sessione.

Nota importante: le proprietà non valorizzate di un documento creato in memoria sono *undefined*. Le proprietà non valorizzate di un documento caricato dal database sono *null*.

Inserimento in collection

Una caso frequente di inserimento di documenti riguarda l'aggiunta di un sotto documento ad un documento principale. Nel caso dell'esempio precedente, si può trattare di aggiungere un documento *RigaOrdine* al relativo documento *Ordine*.

Questo può essere ottenuto operando sulla collection *righeOrdine* del documento *Ordine*, come mostrato con il codice che segue:

```
App.Session.prototype.inserimentoRiga = function (docOrdine)
{
  var r = new App.NBE.RigaOrdine(app, {ProductID : 1});
  r.inserted = true;
  //
  docOrdine.righeOrdine.add(r);
  //
  yield docOrdine.save();
};
```

Al metodo *inserimentoRiga* viene passato il documento *Ordine* nel parametro *docOrdine*. A questo punto il codice istanzia il documento *RigaOrdine*, lo marca come *inserted* e poi lo aggiunge tramite il metodo *add* alla collection *righeOrdine* dell'ordine passato come parametro.

A questo punto è possibile salvare l'intero ordine e questo causerà l'inserimento nel database dei dati relativi alla nuova riga.

Caricamento dal database

Il caricamento di documenti dal database è una delle operazioni più frequenti nell'ambito della programmazione di Instant Developer Cloud. Caricare i documenti dal database permette di visualizzarli, modificarli e salvare le modifiche.

Esistono due tipi principali di caricamento:

1) Il caricamento di un documento a partire da una chiave primaria o esterna.
2) Il caricamento di una collection di documenti a partire da filtri sui dati.

Caricamento di un documento per chiave

Per caricare un documento in base ad una chiave è necessario utilizzare il metodo statico *loadByKey* chiamato sulla classe specifica del documento. La seguente istruzione carica il documento prodotto con *ProductID* pari a 1.

```
let p = yield App.NBE.Product.loadByKey(app, {ProductID:1});
```

Il metodo *loadByKey* restituisce il documento corrispondente ai dati passati come primo parametro se la tabella da cui il documento è stato derivato contiene uno ed un solo record corrispondente a tali dati; in alternativa restituisce *null*.

Il valore *null* viene restituito anche nel caso in cui si tenti di chiamare il metodo *loadByKey* su un documento che non deriva da una tabella del database. In questo caso si vedrà anche il seguente messaggio di warning nella console di debug: *Unable to load data without query*.

Se il caricamento avviene per chiave primaria ed essa è costituita da un unico campo, è possibile passare direttamente il valore della chiave come primo parametro, come vediamo nell'esempio seguente:

```
let p = yield App.NBE.Product.loadByKey(app, 1);
```

È possibile passare anche altri criteri di filtro, che devono comunque identificare un record preciso per ottenere un risultato diverso da *null*. Ad esempio:

```
let k = {ProductName:"Chai",SupplierID: 1};
let p = yield App.NBE.Product.loadByKey(app, k);
```

Il metodo *loadByKey* permette anche di specificare una serie di opzioni di caricamento come secondo parametro. Vediamo le principali:

- *childLevel:* specifica se il caricamento del documento deve essere propagato anche alle collection e, nel caso, fino a quale livello. Il valore di default, 0, carica solo il documento ma non le collection. Passando 1 verranno caricate anche le collection, ma solo al primo livello e così via.
- *cache*: impostando a *true* questa opzione, il documento verrà caricato da una cache in memoria, senza accedere al database. Se il documento non è ancora in cache, verrà effettuata la query e poi il risultato verrà memorizzato anche nella cache. Vale solo se *childLevel* è 0.
- *remote*: impostando a *true* questa opzione, il caricamento avverrà tramite il sistema di sincronizzazione. Per maggiori informazioni si rimanda al capitolo relativo.
- *dataStore*: permette di specificare un'istanza di database tramite cui effettuare il caricamento. Se questa non viene specificata, verrà utilizzata la medesima connessione per tutti i documenti di una singola sessione. In alcuni casi particolari, legati alle operazioni di sincronizzazione, può essere utile utilizzare una connessione specifica.

Vediamo come esempio il caricamento di un documento *Ordine* e delle relative collection di primo livello:

```
let o = yield App.NBE.Ordine.loadByKey(app, 10248, {childLevel:1});
```

Eventi relativi al caricamento

I documenti possono rispondere al ciclo di caricamento implementando gli eventi *beforeLoad* e *afterLoad* sulla classe relativa al documento. Anche se il nome di questi due eventi è simile, il significato e l'utilizzo che essi hanno è molto diverso.

Il framework chiama l'evento *beforeLoad* su un'istanza fittizia del documento in fase di caricamento. Tale evento serve per consentire al documento di personalizzare il metodo di caricamento stesso. Ad esempio un documento potrebbe richiedere un algoritmo di

caricamento non esprimibile da una singola query sul database. In tal caso l'evento *beforeLoad* consente di scrivere un proprio algoritmo di caricamento al posto della query.

Di utilizzo più frequente è invece l'evento *afterLoad*, che viene notificato ad ogni istanza di documento subito dopo che essa è stata caricata dal database, oppure al primo caricamento di una collection dal database. Questo evento può essere usato per completare il caricamento del documento, ad esempio valorizzando proprietà unbound in funzione di quelle collegate ai campi della tabella. In questi casi si consiglia sempre di chiamare il metodo base *setOriginal()* che fotografa la versione attuale del documento come versione originale, cioè corrispondente alla situazione presente nel database.

Nell'esempio seguente l'evento *afterLoad* viene usato per impostare la proprietà unbound *NomeCognome* di un *Impiegato* a partire dai campi del database *FirstName* e *LastName*.

```
App.NBE.Impiegato.prototype.afterLoad = function (alreadyLoaded)
{
  this.NomeCognome = this.FirstName + " " + this.LastName;
  this.setOriginal();
};
```

Caricamento di una collection di documenti

Oltre al caricamento di un documento specifico per chiave, è necessario poter caricare dal database una collection di documenti che soddisfano determinati requisiti. A tal fine è possibile utilizzare il metodo *loadCollection* del documento di cui interessa caricare i dati. L'esempio di codice seguente carica dal database la collection dei prodotti di una determinata categoria e stampa nel log della console i loro nomi.

```
let c = yield App.NBE.Product.loadCollection(app, {CategoryID:1});
for (let i=0; i<c.length; i++) {
  console.log(c.rows[i].ProductName);
}
```

Il metodo *loadCollection* restituisce sempre un oggetto *Collection* che contiene il risultato della ricerca nel database. La collection potrebbe essere vuota se nel database non ci sono dati corrispondenti. In caso contrario, i documenti sono contenuti nell'array *rows* della collection.

Il metodo *loadCollection* ammette due parametri. Il primo è il *template*, un oggetto che contiene i criteri di caricamento. I valori delle proprietà del template diventano condizioni di filtro per la query di caricamento della collection: nell'esempio precedente, il filtro `{CategoryID:1}` verrà espresso come `where CategoryID=1`.

Invece che un valore singolo è possibile anche passare un array di valori e in questo caso verrà usato *in* come operatore di selezione. Ad esempio il criterio `{CategoryID:[1,2,3]}` verrà espresso come `where CategoryID in (1,2,3)`. Ovviamente al posto di valori costanti - usati solo negli esempi - in pratica si useranno quasi sempre variabili o parametri.

Il secondo parametro è rappresentato dalle opzioni relative al caricamento; oltre alle opzioni del caricamento del singolo documento troviamo le seguenti:

- *useQBE*: se impostato a *true*, attiva la modalità Query By Example.
- *maxRows*: numero massimo di documenti da restituire.
- *orderBy*: criterio di ordinamento; ad esempio `"CategoryName desc, ProductName"`.

Attivando la modalità Query By Example cambia il modo con cui vengono considerati i criteri di filtro: oltre ai casi precedenti i criteri potranno essere passati come stringhe per attivare ricerche più complesse. Vediamo alcuni esempi:

- `{ProductName: "Ch"}` viene espresso come `where ProductName ILIKE 'Ch%'`.
- `{ProductName: "*Ch*"}` diventa `where ProductName ILIKE '%Ch%'`.
- `{ProductName: "=Ch"}` viene espresso come `where ProductName = 'Ch'`.
- `{ProductName: "A:G"}` diventa `where ProductName between 'A' and 'G'`.
- `{ProductName: "."}` diventa `where ProductName is not null`.
- `{ProductName: "!"}` diventa `where ProductName is null`.
- `{ProductName: "A:G;!"}` diventa `where (ProductName between 'A' and 'G' or ProductName is null)`.

È possibile leggere la documentazione completa della modalità Query By Example nella documentazione in linea.

Caricamento di una collection contenuta in un documento

Esiste un altro caso di caricamento di collection, quello in cui essa è definita come parte di un documento padre. Negli esempi precedenti abbiamo immaginato un documento *Ordine* che possiede la collection *righeOrdine* allo scopo di gestire le relative righe.

Trattando del caricamento di documenti e collection abbiamo visto che per default viene caricato solo il documento stesso e non le sue collection, a meno di non specificare l'opzione *childLevel*. È questo il caso più frequente: infatti è opportuno ritardare il più possibile il caricamento di dati dal database.

Dobbiamo quindi vedere come effettuare il caricamento di una collection di un documento che è già stato caricato dal database. Questo può avvenire semplicemente chiamando il metodo *load()* della collection, come mostrato nell'esempio seguente:

```
let o = yield App.NBE.Ordine.loadByKey(app, 10248);
yield o.righeOrdine.load();
```

La prima istruzione effettua il caricamento dell'ordine, ma non delle sue righe. La seconda istruzione richiede il caricamento della collection delle righe. Si noti che in questo caso non è necessario passare criteri di filtro perché il legame tra l'ordine e le sue righe è già stato specificato in fase di definizione della collection, così come la clausola di ordinamento.

Al termine del caricamento della collection viene nuovamente notificato l'evento *afterLoad* al documento *Ordine* che può nuovamente completare il suo stato in funzione dei nuovi dati ottenuti.

Il caricamento della collection avviene solo se essa non è già stata caricata (vedi proprietà *loaded* della collection). È quindi possibile chiamare il metodo *load* più di una volta senza per questo causare perdita di performance. Se si desidera forzare un nuovo caricamento di una collection è possibile utilizzare il metodo *reload*.

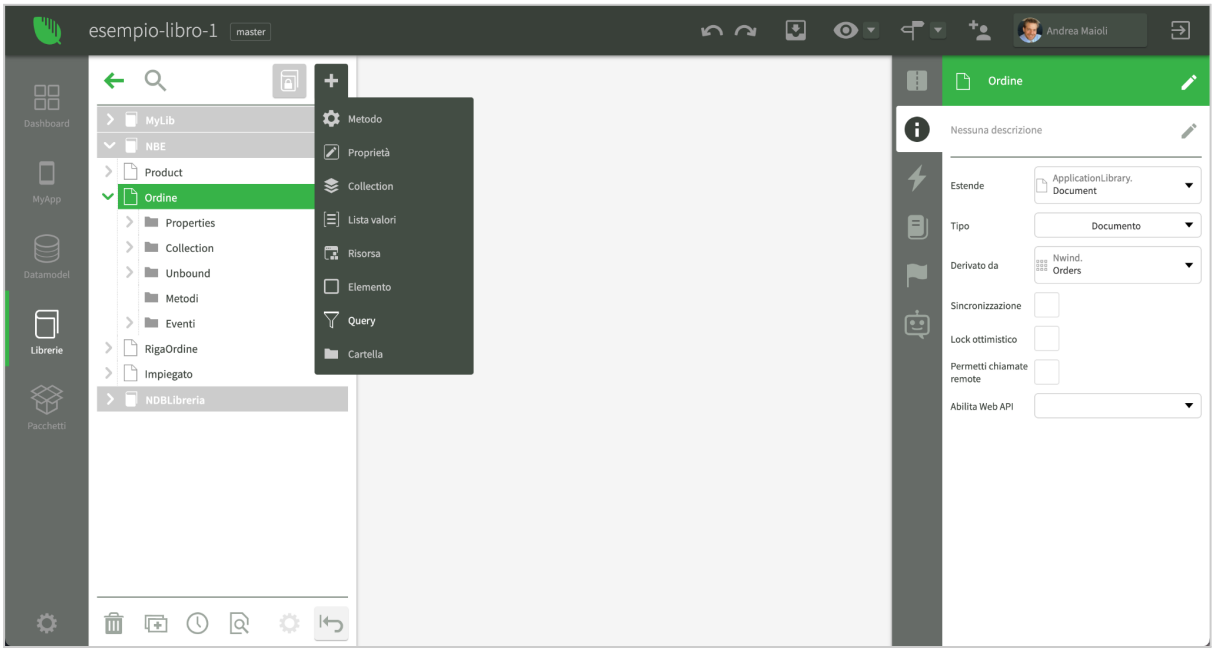
Si noti infine che la proprietà *loaded* ed i metodi *load* e *reload* sono disponibili anche su un'istanza di documento: è quindi possibile forzare un nuovo caricamento di un'istanza di documento dal database chiamando il metodo *reload* su di esso.

Query di caricamento di un documento

Nei casi visti finora, i metodi di caricamento dei documenti dal database erano in grado di calcolare automaticamente la query da effettuare, sia nel caso di singolo documento che di collection. Le query generate dal framework coinvolgono la tabella su cui il documento è basato, eventualmente in join con altre tabelle necessarie a reperire le proprietà derivate.

In alcuni casi, tuttavia, è necessario poter specificare una query di caricamento personalizzata per effettuare calcoli, per reperire informazioni più complesse anche usando subquery, o infine per impostare filtri di caricamento permanenti.

Instant Developer Cloud permette di aggiungere una query di caricamento al documento, tramite il menu *+query* relativo alla classe del documento.



Aggiungere la query di caricamento al documento Ordine

Utilizzando questo comando, al documento viene aggiunto un oggetto query che rappresenta la medesima operazione che il framework utilizzerebbe per caricare il documento. Ad esempio, la query generata per il documento *Product* potrebbe essere la seguente:

```
select
  *
from
  Products
```

Oppure per le righe dell'ordine, dove è presente una proprietà derivata:

```
select
  A.*,
  B.ProductName as NomeProdotto
from
  OrderDetails A
  left outer join Products B on B.ProductID = A.ProductID
```

Dopo avere aggiunto la query è possibile modificarla a piacimento tenendo conto delle seguenti regole:

- Ogni espressione aggiunta alla select list deve specificare la clausola *as* <*alias*>. L'alias dato all'espressione deve avere lo stesso nome di una proprietà unbound del documento in cui tale espressione verrà memorizzata.
- È possibile referenziare proprietà della sessione per inserire clausole di filtro che saranno sempre aggiunte alle query di caricamento.

Un esempio di query di caricamento di un documento *Product* può essere la seguente:

```
select
  *,
  (select sum(Quantity) from OrderDetails
   where ProductID = Products.ProductID) as soldQty
from
  Products
where CategoryID = app.sessionCategory or app.sessionCategory is null
```

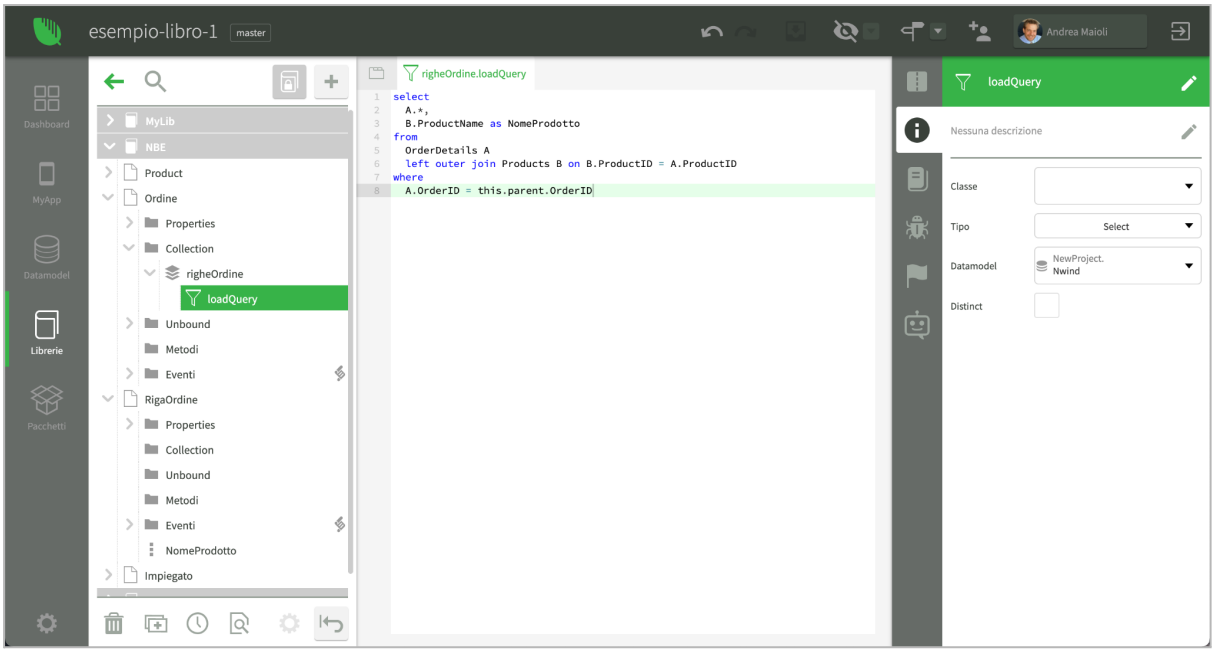
In particolare possiamo notare che il filtro `CategoryID = app.sessionCategory or app.sessionCategory is null` è un filtro permanente che permette di caricare solamente i prodotti appartenenti ad una determinata categoria, se la proprietà *app.sessionCategory* è valorizzata. Un metodo più flessibile per selezionare i documenti è quello di passare i criteri di filtro nel comando di caricamento dei documenti.

Inoltre possiamo notare la subquery che calcola il totale della quantità venduta per un determinato prodotto e la memorizza nella proprietà unbound *soldQty*.

Query di caricamento di una collection

In maniera analoga ai documenti, è possibile personalizzare la query di caricamento di una collection quando essa è parte di una struttura multilivello, come avviene, ad esempio, nel caso della collection *righeOrdine* del documento *Ordine*. Per personalizzare la query occorre utilizzare il comando *+query* relativo alla proprietà di tipo collection.

Dopo aver utilizzato questo comando, apparirà un oggetto query contenuto nella proprietà, come ad esempio il seguente:



Aggiungere la query di caricamento alla collection RigheOrdine

La composizione iniziale della query di caricamento della collection è la medesima della query che verrebbe calcolata dal framework al momento del caricamento. In sostanza si utilizza la query del documento *RigaOrdine* aggiungendo la clausola di collegamento con il documento che contiene la collection: **`where A.OrderID = this.parent.OrderID`**. Si noti che il riferimento a *this.parent* permette di raggiungere le proprietà del documento *Ordine*, in quanto esso è il *parent* della collection *righeOrdine*.

Caricamento di una collection tramite recordset

Nei paragrafi precedenti abbiamo visto che il metodo *loadCollection* è in grado di caricare dal database un insieme di documenti che soddisfano determinati parametri espressi come espressioni semplici oppure tramite Query By Example.

Tuttavia ci possono essere casi in cui le condizioni di estrazione sono più complesse di quanto gestibile da Query By Example, oppure perchè la select list deve essere modificata, o anche ridotta, rispetto al caso di default.

In tutti questi casi è possibile effettuare una normale query di caricamento dati, per poi chiamare il metodo *toCollection()* dell'oggetto recordset restituito dalla query. Questo metodo converte i dati del recordset in una serie di documenti della classe specificata e restituisce una collection che li contiene.

Nell'esempio che segue vediamo come caricare una collection di documenti tramite il metodo *toCollection*:

```
App.Session.prototype.productExample = function (id)
{
  var rs = yield App.Nwind.query(app," \
    select                    \
      *                       \
```

```
    from                    \
      Products              \
    where                   \
      ProductID > id        \
  ");
  var c = yield rs.toCollection(App.NBE.Product);
};
```

Il metodo *toCollection()* viene sincronizzato tramite *yield* perché durante la conversione dei dati viene chiamato l'evento *afterLoad* dei documenti in fase di conversione e tale evento potrebbe contenere codice asincrono di cui viene aspettata la conclusione.

Da notare infine che il metodo *toCollection()* valorizza nei documenti in fase di conversione tutte le proprietà presenti nel recordset cercandole per nome, sia che esse siano proprietà relative ai campi della tabella del documento, proprietà derivate da altre tabelle o, ancora, proprietà unbound o proprietà non definite a design time.

Caricamento di documenti correlati

Vediamo ora un ultimo tipo di metodo di caricamento che permette di navigare fra le relazioni identificative e ottenere al volo le informazioni aggiuntive necessarie.

Immaginiamo che al momento dell'impostazione della proprietà *ProductID* di un documento *RigaOrdine* si desideri impostare il prezzo unitario (*UnitPrice*) in base all'anagrafica del prodotto. L'operazione da eseguire sarebbe quella di un caricamento per chiave, come nell'esempio seguente:

```
let p = App.NBE.Product.loadByKey(this.ProductID);
```

Se il legame fra OrderDetails e Products (le tabelle correlate ai documenti RigaOrdine e Product) è unico, cioè esiste un'unica relazione fra le due tabelle, è possibile utilizzare anche il metodo *getRelated()* che recupera l'informazione utilizzando anche la cache:

```
let p = this.getRelated(App.NBE.Product);
```

getRelated() può essere usato anche nel caso in cui ci siano più relazioni fra le tabelle in questione, ma verrà utilizzata sempre quella definita per prima nel progetto.

Invece che una singola classe, è possibile passare come primo parametro un array di classi; in questo modo è possibile navigare nelle relazioni fra le tabelle in un unico passo. Se, ad esempio, a partire dalla *RigaOrdine* volessimo recuperare il documento categoria del prodotto specificato sulla riga, potremmo scrivere la seguente istruzione:

```
let c = this.getRelated([App.NBE.Product,App.NBE.Category]);
```

In questo modo possiamo scrivere meno righe di codice e ottenere un codice più ottimizzato in quanto i documenti verranno caricati solo la prima volta e successivamente sarà usata la cache.

Modifica e validazione di un documento

Dopo aver caricato un documento dal database, è possibile modificarlo nei seguenti modi:

1) Cambiando il valore di una proprietà.
2) Marcandolo per la cancellazione impostando a *true* la proprietà *deleted*. Questa modifica è valida anche se il documento è contemporaneamente marcato per l'inserimento.
3) Aggiungendo documenti marcati per l'inserimento ad una sua collection tramite il metodo *add()* della collection stessa.

Non sono invece ammesse le seguenti operazioni:

1) Marcare per l'inserimento un documento caricato dal database: al momento del salvataggio causerebbe un errore di chiave duplicata.
2) Rimuovere sotto-documenti da una collection del documento. Il documento rimosso non sarebbe più gestito dalla struttura in memoria, ma rimarrebbe nel database.
3) Aggiungere ad una collection del documento un altro documento appartenente ad una diversa struttura documentale. In tal caso, infatti, il documento verrebbe spostato, quindi rimosso dalla precedente struttura e si ricadrebbe nel caso precedente.
4) Aggiungere ad una collection del documento un documento non marcato per l'inserimento. In questo caso si otterrebbe un errore a livello di database perché il record non esiste ancora.

Il codice seguente carica il prodotto con *ProductID* uguale a 1 dal database, cambia il prezzo e poi lo salva.

```
var p = yield App.NBE.Product.loadByKey(app, 1);
p.UnitPrice = p.UnitPrice * 1.1;
yield p.save();
```

Gestione dello stato del documento

Lo stato di un documento viene memorizzato nelle sue proprietà booleane: *loaded*, *inserted*, *updated*, *deleted*. Per sapere se un documento o una delle sue collection sono state modificate da quando sono state caricate dal database, è possibile utilizzare il metodo *isModified()* del documento stesso. Questo metodo controlla lo stato delle proprietà *inserted*, *updated*, *deleted* sia del documento che del contenuto delle sue collection.

Esistono alcuni metodi che modificano globalmente lo stato di un documento:

- *setOriginal()*: imposta lo stato attuale come originale, perdendo traccia delle modifiche effettuate dal momento del caricamento, sia sul documento che sul contenuto delle sue collection.
- *restoreOriginal()*: annulla le modifiche effettuate ad un documento o alle sue collection, riportandolo allo stato originale. In particolare:
    - Tutte le proprietà vengono riportate al valore originale e *updated* viene impostato a *false*.
    - I documenti marcati per l'inserimento vengono rimossi.
    - I documenti marcati come cancellati vengono resuscitati: *deleted* = *false*.

Se un documento è stato modificato, è possibile leggere il valore originale delle sue proprietà tramite la funzione *getOriginalValue(<nome proprietà>)*. Il nome della proprietà può essere espresso come stringa, oppure riferendo la proprietà in modo statico come nell'esempio seguente.

```
var p = yield App.NBE.Product.loadByKey(app, 1);
p.UnitPrice = p.UnitPrice * 1.1;
var op = p.getOriginalValue(App.NBE.Product.UnitPrice);
console.log("Differenza = ", p.UnitPrice - op);
```

Si segnala che è disponibile anche il metodo *setOriginalValue()* per impostare ad un valore personalizzato il valore originale di una proprietà del documento.

L'evento onChange

In alcuni casi è importante che il documento possa gestire le modifiche che vengono apportate ad esso o alle sue collection. Immaginiamo, ad esempio, che il documento *Ordine* debba tenere aggiornata la proprietà unbound *importoTotale*: ogni volta che viene inserita, cancellata o aggiornata una delle sue righe si dovrebbe effettuare il ricalcolo.

Proprio per gestire questi casi è possibile implementare l'evento *onChange* sulla classe del documento; tale evento viene notificato ogni volta che il documento o il contenuto di una delle sue collection cambia, sia al momento del caricamento che a causa di cambiamenti subiti dopo il caricamento.

Si noti che l'evento *onChange* viene notificato una sola volta al termine di un ciclo di modifiche al documento e non per ognuna di esse. Ad esempio, aggiungendo 100 righe ad un ordine, l'evento *onChange* verrà notificato una sola volta al termine dello script che aggiunge le righe.

L'esempio seguente mostra il codice dell'evento *onChange* per il documento *Ordine*.

```
App.NBE.Ordine.prototype.onChange = function ()
{
  this.importoTotale = this.getTotaleOrdine();
};
```

Validazione di un documento

Prima di concludere la gestione delle modifiche salvandole sul database, è importante poter sapere se lo stato attuale del documento è corretto oppure contiene degli errori. A tal fine è possibile chiamare il metodo *validate()* del documento, che restituisce *true* se il documento è corretto.

Per decidere se lo stato di un documento è corretto, il framework chiama l'evento *onValidate* implementato nella classe del documento. Tale evento riceve come parametro un oggetto *options* che contiene le seguenti proprietà:

- *reason*: è una stringa che specifica perché viene richiesta la validazione. I valori predefiniti sono:
    - *save*: validazione richiesta dal framework prima del salvataggio.
    - sync: validazione richiesta dal framework durante la sincronizzazione.
    - *change*: validazione richiesta dal framework quando il documento è mostrato a video tramite un oggetto Datamap.
- *property*: nel caso *change*, indica quale proprietà del documento è cambiata.
- *skip*: se impostato a true permette di saltare i controlli sulle proprietà obbligatorie che vengono svolti dal framework.

Il codice dell'evento *onValidate*, in base ai parametri ricevuti, può effettuare i controlli richiesti e impostare errori sul documento tramite il metodo *setError()*. Se viene impostato almeno un errore, la validazione fallisce e il metodo *validate* ritorna *false*.

Si noti che i controlli sulle proprietà obbligatorie, cioè quelle collegate a campi not nullable del database, vengono svolti in autonomia da parte del framework. Se si desidera saltare tali controlli, è necessario impostare a *true* la proprietà *skip* del parametro *options*.

È inoltre importante considerare lo stato del documento durante l'esecuzione dei controlli. Se, ad esempio, il documento è marcato per la cancellazione, non è importante segnalare errori che dipendono dal valore delle sue proprietà. In questi casi è possibile condizionare questi controlli alla proprietà *deleted*, oppure al valore restituito dal metodo *isDeleted()* che considera anche se il documento è contenuto in un padre marcato per la cancellazione.

Nell'esempio che segue vediamo come segnalare un errore su una proprietà del documento. Si noti che il metodo *setError*, oltre al messaggio di errore, permette di specificare a quale proprietà esso si riferisce. In questo modo il framework potrà mostrare a video i messaggi di errore nel contesto giusto.

```
App.NBE.RigaOrdine.prototype.onValidate = function (options)
{
  // Solo se la riga non è cancellata
  if (!this.isDeleted()) {
    //
    // Controllo la quantità
    if (this.Quantity<=0) {
      this.setError("La quantità non può essere negativa o zero",
                    App.NBE.RigaOrdine.Quantity);
    }
  }
};
```

Vediamo adesso il caso in cui un documento *Ordine* non può essere salvato se non è stata specificata almeno una riga ordine. Nel codice seguente si noti l'utilizzo del metodo *load* della collection *righeOrdine*, che carica le righe solo se non sono già state caricate, e l'uso della proprietà *count* della collection in alternativa alla *length*. La proprietà *count*, infatti, restituisce il numero di documenti contenuti nella collection non marcati per la cancellazione, mentre *length* restituisce il numero di documenti totali.

```
App.NBE.Ordine.prototype.onValidate = function (options)
{
  if (options.reason === App.Document.validationReasons.save) {
    // Solo se non sono cancellato
    if (!this.isDeleted()) {
      // Carico le righe se non era già stato fatto prima
      yield this.righeOrdine.load();
      // Controllo se ce ne sono
      if (this.righeOrdine.count === 0) {
        this.setError("Prima di salvare occorre aggiungere
                       almeno una riga");
      }
    }
  }
};
```

Salvataggio di un documento

Il salvataggio di un documento avviene tramite il metodo *save()* del documento stesso, che restituisce *true* se l'operazione è andata a buon fine.

L'operazione di salvataggio coinvolge sempre un documento e il contenuto delle sue collection e si svolge come segue:

1. Vengono impostate le proprietà che gestiscono il collegamento fra il documento e il contenuto delle collection. Ad esempio, per tutti i documenti della collection *righeOrdine* verrà impostata la proprietà *OrderID* al valore della corrispondente proprietà del documento *Ordine* che li contiene.
2. Sia il documento che il contenuto delle collection viene validato con *reason* = *"save"*. Se tutti i documenti non rilevano errori, si passa alla fase successiva.
3. Si seleziona l'istanza del database su cui effettuare il salvataggio. Se su tale istanza non c'è una transazione attiva, si inizia una nuova transazione.
4. Viene notificato l'evento *onSave({phase:"beforeSave"})* a tutti i documenti coinvolti nel salvataggio.
5. Viene notificato l'evento *onSave({phase:"inserting"})* a tutti i documenti coinvolti nel salvataggio. Se un documento è marcato per l'inserimento, dopo aver notificato l'evento il documento viene inserito nel database con un comando SQL di *insert.*
6. Viene notificato l'evento *onSave({phase:"updating"})* a tutti i documenti coinvolti nel salvataggio. Se un documento è stato modificato, dopo aver notificato l'evento il documento viene aggiornato nel database con un comando SQL di *update* in cui compaiono solo le proprietà effettivamente modificate.
7. Viene notificato l'evento *onSave({phase:"deleting"})* a tutti i documenti coinvolti nel salvataggio. Se un documento è marcato per la cancellazione, dopo aver notificato l'evento il documento viene cancellato dal database con un comando SQL di *delete.*
8. Viene notificato l'evento *onSave({phase:"afterSave"})* a tutti i documenti coinvolti nel salvataggio.
9. Se nei database coinvolti nel salvataggio era stata aperta una transazione (vedi punto 2), essa viene confermata se il salvataggio ha avuto successo nel suo complesso, oppure viene annullata.

10. Se l'operazione ha avuto successo, tutti i documenti coinvolti nel salvataggio vengono marcati come originali in quanto lo stato attuale è ora sincronizzato con il contenuto del database.
11. Viene restituito *true* se tutto si è svolto correttamente o *false* se l'operazione è stata interrotta a causa di errori.

Il codice seguente esemplifica il ciclo di modifica di un documento che prima viene caricato dal database, poi modificato ed infine salvato. È molto importante controllare il valore restituito dal metodo *save* perché in caso di interruzione dell'operazione possiamo richiedere al documento gli errori avvenuti tramite il metodo *getErrors()*.

```
var p = yield App.NBE.Product.loadByKey(app, 1);
p.UnitPrice = p.UnitPrice * 1.1;
var b = yield p.save();
if (!b) {
  console.log(p.getErrors().join("\n"));
}
```

L'evento onSave

Tramite l'evento *onSave* il documento può gestire la propria fase di salvataggio sul database. Durante il ciclo di salvataggio, esso viene chiamato per cinque volte su tutti i documenti coinvolti, cioè sul documento per cui è stato richiesto il salvataggio e sui documenti contenuti nelle sue collection, a tutti i livelli.

L'evento *onSave* riceve il parametro *options* di tipo oggetto. Le proprietà di *options* sono le seguenti:

- *phase*: è la fase per cui viene chiamato l'evento (*beforeSave, inserting, updating, deleting, afterSave*).
- *cancel*: se viene impostato a *true*, l'intero salvataggio viene annullato.
- *skip*: se viene impostato a *true*, il comando SQL corrispondente non viene eseguito ma il ciclo di salvataggio continua.

Ogni fase del ciclo di salvataggio ha uno scopo specifico che viene illustrato di seguito.

Fase *beforeSave*

Questa fase avviene dopo la validazione e l'apertura della transazione sul database, ma prima di iniziare ad inviare i comandi SQL. Si consiglia di utilizzare questa fase per raggiungere una coerenza interna fra il documento e il contenuto delle sue collection.

Ad esempio, è in questa fase che si può calcolare il numero dell'ordine se si desidera avere numeri di ordine sequenziali senza interruzioni. Oppure, sempre in questa fase è possibile ricalcolare le proprietà del documento in funzione del contenuto delle collection, se non è stato possibile farlo in precedenza.

In altri casi si può utilizzare la fase *beforeSave* anche per aggiungere nuovi documenti alle collection contenute nel documento in fase di salvataggio. In questo caso, essi non verranno

nuovamente validati per questo salvataggio, inoltre non verranno impostate le proprietà di collegamento tra questi documenti e il contenuto delle loro collection.

Impostando la proprietà *cancel* del parametro *options* a *true* durante questa fase si interrompe immediatamente l'intero ciclo di salvataggio.

Fase *inserting*

La fase *inserting* inizia dopo quella *beforeSave* se nessun documento ha annullato il salvataggio impostando *options.cancel* a *true*. Durante questa fase è possibile personalizzare l'operazione di inserimento vera e propria; è possibile anche saltare quella generata dal framework impostando *options.skip* a *true*.

Dopo aver notificato l'evento *onSave* in fase *inserting,* i documenti marcati per l'inserimento vengono effettivamente inseriti nel database. Se il comando di inserimento genera un errore, l'intera procedura di salvataggio viene annullata e con essa la transazione nel database.

Fase *updating*

La fase *updating* inizia dopo quella *inserting* se nessun documento ha annullato il salvataggio impostando *options.cancel* a *true* e se non è avvenuto nessun errore. Durante questa fase è possibile personalizzare l'operazione di aggiornamento vera e propria; è possibile anche saltare quella generata dal framework impostando *options.skip* a *true*.

Dopo aver notificato l'evento *onSave* in fase *updating,* i documenti che risultano modificati vengono effettivamente aggiornati nel database tramite un comando SQL di *update* che modifica solo le proprietà diverse dallo stato originale. Se il comando di aggiornamento genera un errore, l'intera procedura di salvataggio viene annullata e con essa la transazione nel database.

Fase *deleting*

La fase *deleting* inizia dopo quella *updating* se nessun documento ha annullato il salvataggio impostando *options.cancel* a *true* e se non è avvenuto nessun errore. Durante questa fase è possibile personalizzare l'operazione di cancellazione vera e propria; è possibile anche saltare quella generata dal framework impostando *options.skip* a *true*.

Dopo aver notificato l'evento *onSave* in fase *deleting,* i documenti marcati per la cancellazione vengono effettivamente eliminati dal database tramite un comando SQL di *delete*. Se il comando di cancellazione genera un errore, l'intera procedura di salvataggio viene annullata e con essa la transazione nel database.

Nota bene: nella fase di cancellazione, i documenti vengono considerati in ordine inverso, cioè prima vengono cancellati i documenti figli e dopo i padri. Vedi anche il paragrafo seguente "Propagazione della cancellazione".

Fase *afterSave*

La fase *afterSave* inizia dopo quella *deleting* se nessun documento ha annullato il salvataggio impostando *options.cancel* a *true* e se non è avvenuto nessun errore.

Durante questa fase tutti i documenti sono stati aggiornati nel database ed è quindi possibile completare il salvataggio modificando eventuali altri documenti esterni alla struttura in fase di salvataggio. Le istanze in memoria dei documenti mantengono ancora traccia delle modifiche ed è ancora possibile annullare l'intero salvataggio impostando *options.cancel* a *true* oppure in caso di errore.

Un esempio di uso della fase *afterSave* è quello in cui si aggiorna la giacenza di un documento articolo in funzione del salvataggio di un documento che rappresenta un movimento di magazzino. Vediamo un esempio di codice da utilizzare:

```
App.NBE.RigaOrdine.prototype.onSave = function (options)
{
  if (options.phase === App.Document.savePhases.afterSave) {
    // Se la riga è modificata
    // Calcolo la differenza di quantità
    let newqty = this.deleted ? 0 : this.Quantity;
    let oldqty = this.getOriginalValue(App.NBE.RigaOrdine.Quantity) || 0;
    let deltaq = newqty - oldqty;
    //
    if (deltaq) {
      // Devo aggiornare la giacenza del prodotto.
      let p = yield this.getRelated(App.NBE.Product);
      if (p) {
        p.UnitsInStock -= deltaq;
        p.UnitsOnOrder += deltaq;
        if (!yield p.save())
          options.cancel = true;
      }
    }
  }
};
```

È interessante notare che il salvataggio del documento *Product* identificato nel documento riga ordine avviene nella medesima transazione utilizzata per salvare tale documento. In caso di problemi nella gestione del prodotto, tutto il salvataggio verrà annullato sia nel database che come istanza di documento in memoria.

Propagazione della cancellazione

Quando abbiamo parlato della definizione di documenti a più livelli, abbiamo visto che essa risulta conveniente quando nel database è presente una relazione di tipo possessivo fra le tabelle. Riprendendo l'esempio dei documenti *Ordine* e *RigaOrdine*, è corretto che *Ordine* abbia una collection di documenti *RigaOrdine*, in quanto la relazione fra le due tabelle esprime un concetto di possesso.

A livello fisico, una relazione di possesso dovrebbe essere espressa tramite la regola di cancellazione *delete cascade*; infatti, in questo caso, quando il record padre viene cancellato, anche tutti i figli vengono eliminati. Nel caso dell'esempio, se un documento *Ordine* viene marcato per la cancellazione e poi salvato, l'eliminazione delle relative righe

avviene automaticamente a livello di database. Non è infatti necessario che esse siano marcate per la cancellazione a livello di documento.

Quando non è possibile utilizzare una relazione di tipo *delete cascade* è necessario propagare la marcatura della cancellazione a livello di documento. A tal fine è possibile utilizzare l'evento *onSave* nella fase *beforeSave*, come mostrato dal codice seguente.

```
App.NBE.Ordine.prototype.onSave = function (options)
{
  if (options.phase === App.Document.savePhases.beforeSave) {
    if (this.deleted) {
      yield this.righeOrdine.load();
      for (let i = 0; i < this.righeOrdine.length; i++) {
        this.righeOrdine.rows[i].deleted = true;
      }
    }
  }
};
```

Salvataggio di una collection

Oltre all'operazione di salvataggio di un documento, il framework DO permette di salvare anche una serie di documenti contenuti in una collection che non sia parte della struttura di un documento.

Per salvare tutta una collection è disponibile il metodo *save(options)*. Per default tutti i documenti della collection verranno salvati nella medesima transazione per ragioni di performance. Se si desidera utilizzare una transazione per ogni documento, è necessario impostare a true la proprietà *autoCommit* del parametro *options*.

Estensione dei documenti

Tutti i documenti, considerati come classi di codice, devono derivare dalla classe del framework *App.Document*. Tuttavia questa derivazione può essere di tipo diretto o indiretto, cioè un documento può derivare da un documento base che a sua volta deriva da *App.Document*.

Ci sono due casi d'uso di derivazione indiretta di documenti:

1) Quando si desidera mettere a fattor comune una serie di comportamenti per un determinato insieme di documenti.
2) Quando la struttura del database contiene relazioni di estensione (*is-a*) fra le tabelle.

Nel primo caso il documento base, quello che contiene i metodi comuni, non deve essere collegato a nessuna tabella di database. In questo caso, istanziando i documenti estesi, si potranno indifferentemente utilizzare i metodi specifici e quelli del documento base.

Implementazione di eventi per i documenti estesi

Quando si implementa un evento in un documento esteso, se tale evento viene definito anche in quello base occorre ricordarsi di chiamare la versione base all'interno di quella estesa.

Per questa ragione, quando l'IDE genera il metodo per l'evento nella classe estesa viene inserita la chiamata a quello base, come si vede nell'esempio seguente:

```
App.NBE.Impiegato.prototype.onValidate = function (options)
{
  // Chiamo l'evento della classe base
  yield App.NBE.DocBase.prototype.onValidate.call(this, options);
};
```

La chiamata all'evento della classe base viene inserita solamente se l'evento è effettivamente stato implementato nel documento base.

Estensione con relazione di tipo is-a

Una relazione di tipo *is-a* a livello di database esprime il concetto di estensione di un tipo di record contenuto in una tabella rispetto ad un altro.

Un esempio di questo può essere una tabella di *soggetti* che contiene i dati anagrafici generali e poi una tabella di *clienti* che, oltre ai dati anagrafici, contiene i dati specifici dei soggetti che sono anche clienti. Per esprimere una relazione di tipo *is-a*, la tabella dei clienti deve avere una relazione verso quella dei soggetti di tipo 1-1, cioè che collega le primary key delle due tabelle.

La procedura di creazione documenti a partire dalla struttura del database identifica il caso indicato nell'esempio (relazioni 1-1 fra primary key di tabelle) e dichiara che il documento che possiede la relazione estende quello base.

Quando nella catena delle estensioni viene coinvolto più di un documento collegato con il database, le query generate dal framework diventano più complesse, ed in particolare:

1) La query di caricamento del documento mette in join tutte le tabelle coinvolte nella catena di estensione usando la relazione 1-1 fra le primary key delle tabelle. Vengono estratte contemporaneamente tutte le proprietà definite per l'intera catena di estensione.
2) Durante il salvataggio, vengono generate più query di inserimento, aggiornamento o cancellazione, una per ogni tabella coinvolta nella catena di estensione. L'ordine delle query è dal padre al figlio in caso di inserimento o dal figlio al padre in caso di cancellazione.

Si segnala che l'estensione di documenti di tipo *is-a* non è una pratica comune a causa del fatto che questo pattern genera strutture del database ritenute in genere meno leggibili. Più frequentemente si osserva l'utilizzo di normali relazioni identificative o possessive per esprimere concetti analoghi all'estensione.