Datamap

Indice generale

Introduzione 3
- Cosa è una datamap? 3
- Elenco delle caratteristiche 3
- Struttura interna di una datamap 4

Creazione di liste di documenti 5
- Formattazione condizionale 6
- Caricamento dei dati 8
  - Documenti e collection 8
  - Query strutturata 8
  - Caricamento in memoria 8
  - Eventi coinvolti nel ciclo di caricamento dei dati 9
  - Filtratura dei dati caricati dalla datamap 9
- Gestione del contenuto della sorgente dati 10
  - Lettura del contenuto della sorgente dati 10
  - Ricerca e selezione dei dati 11
    - Ricerca tramite caricamento dal database 12
  - Ordinamento delle righe 12
  - Raggruppamento delle righe 13
  - Ottimizzazione delle performance 15
  - Massimo numero di record 15
  - Visualizzazione incrementale 15
    - Visualizzazione incrementale e raggruppamenti 16
  - Caricamento dati incrementale 17
  - Liste virtuali 18
  - Struttura del template 19
  - Eventi di composizione 21
- Gestione del template visuale della lista 21
  - Struttura del template 21
  - Collegamento tra datamap e template 22
  - Aggiornamento di elementi esterni al template 22
- Modifica e salvataggio dei dati 23
  - Salvataggio delle modifiche 24

Modifica dei documenti con videate di dettaglio 24
- Gestione delle modifiche pendenti 27
- Gestione dello stato dei pulsanti dell’intestazione 28
- Selezione di dati provenienti da altre tabelle 28
  - Uso di controlli di tipo autocomplete 30
  - Uso di datamap basate su documenti e controlli di tipo autocomplete 31
  - Selezione di dati provenienti da altre tabelle in layout lista 32

Datamap innestate e ricorsive **32**
Datamap ricorsive 34

Datamap

Scopri come unire backend e frontend con il controller Datamap.

Introduzione

Cosa è una datamap?

Una delle specifiche più comuni durante lo sviluppo di applicazioni consiste nel mostrare a video dati provenienti dal database o, più in generale, dal backend.

Quello che può sembrare un compito semplice, in realtà presenta aspetti di complicazione soprattutto nell’ambito delle applicazioni web basate su browser. Facciamo alcuni esempi:

1) Sincronizzazione degli elementi visuali al variare dei dati da mostrare.
2) Rendering efficace di una grande mole di record.
3) Gestione di filtri, raggruppamenti ed ordinamenti dei dati.
4) Rendering di strutture multi-livello.

Per risolvere queste problematiche, Instant Developer Cloud contiene un componente speciale chiamato *datamap*. Una datamap è un controller che, collegato ad una sorgente dati, si occupa di creare e mantenere sincronizzate le strutture di elementi visuali necessarie a mostrarli.

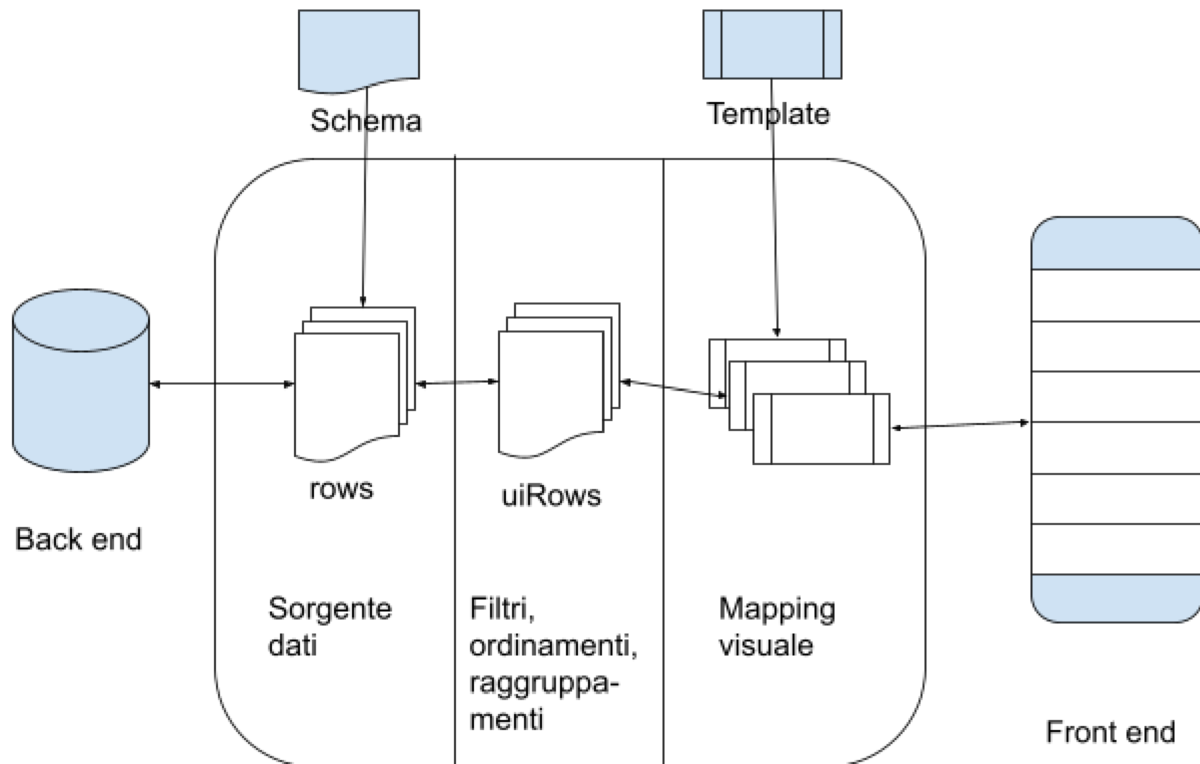
Elenco delle caratteristiche

Vediamo una lista delle principali caratteristiche gestite dalle datamap:

1) Sorgenti dati basate su documenti o collection, su query o su dati caricati in memoria.
2) Possibilità di estendere le strutture dati con proprietà unbound a livello di singola datamap.
3) Gestione del caricamento dei dati dal database, anche in modalità paginata.
4) Gestione di ordinamenti, raggruppamenti e filtri sui dati direttamente in memoria.
5) Creazione, aggiornamento e rimozione automatica degli elementi visuali corrispondenti ai dati in funzione di un template fornito alla datamap.
6) Stilizzazione degli elementi visuali in funzione dei dati.
7) Gestione delle sezioni visuali (intestazioni e piede) per i raggruppamenti.
8) Visualizzazione paginata automatica.
9) Visualizzazione con finestra dati virtuale automatica.
10) Gestione strutture dati innestate a n-livelli.
11) Gestione strutture dati ricorsive.
12) Gestione del collegamento con elementi visuali per la modifica dei dati.
13) Visualizzazione automatica degli errori di validazione.
14) Utilizzo di datamap come sorgenti di dati per elementi visuali come combo-box, auto complete, grafici, mappe, calendari, eccetera.
15) Ciclo di vita ad eventi per una gestione centralizzata del codice applicativo.

Struttura interna di una datamap

Nel diagramma seguente possiamo vedere il contenuto schematico di una datamap e il rapporto con gli elementi esterni ad essa.



Nel primo stadio, una datamap gestisce una sorgente dati, nella quale vengono memorizzati i dati da gestire. Tale sorgente dati può essere di tre tipi:

1) Dati in memoria: i dati sono contenuti nell'array *rows* della datamap e vengono caricati dal codice dell'applicazione.
2) Dati provenienti da database: la datamap può contenere una query strutturata che specifica come caricare i dati dal database. Il caricamento viene gestito dalle funzioni interne della datamap stessa. Anche in questo caso i dati sono contenuti nell'array *rows*.
3) Dati provenienti da una collection di documenti: la collection può essere gestita direttamente dalla datamap oppure esternamente. In entrambi i casi, l'array *rows* punta ai dati contenuti nella collection.

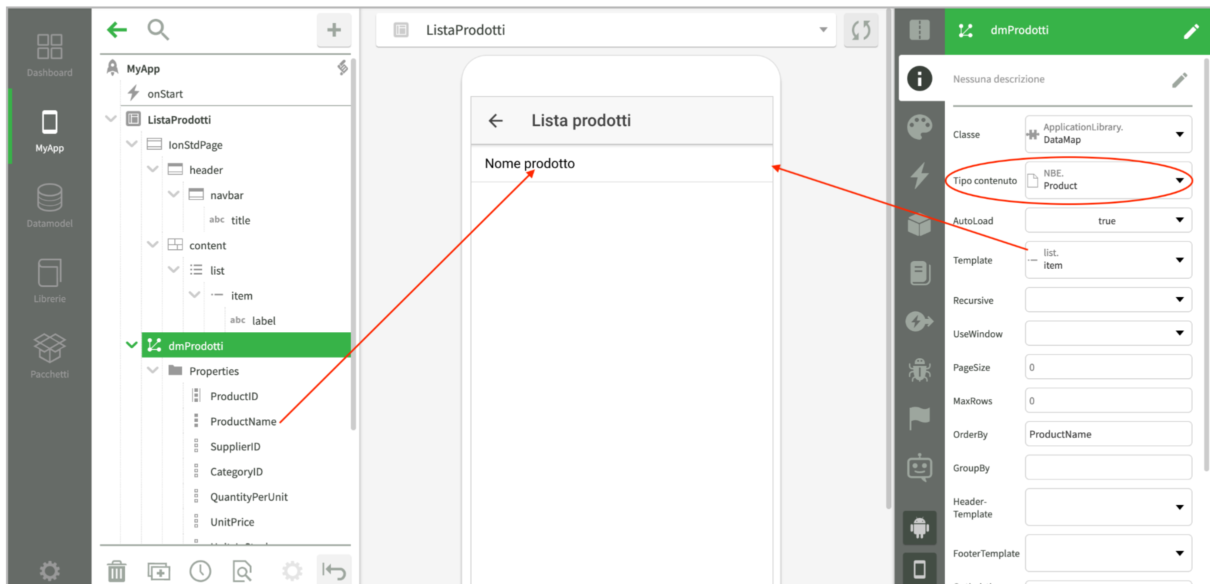
Nel secondo stadio, la datamap gestisce un set visuale di dati (array *uiRows*) che è diverso da quello contenuto nella sorgente dati. I dati possono venire nascosti a causa di un filtro, riordinati o raggruppati, o infine possono essere marcati per la cancellazione e quindi non essere visibili. In tutti questi casi il sistema interno alla datamap mantiene sincronizzato l'array *uiRows* che rappresenta i dati visibili con *rows*, la sorgente dati.

Nel terzo stadio la datamap si occupa di generare dei cloni del template fornito e di mappare i dati visibili sugli elementi clonati. La mappatura è bidirezionale in quanto se gli elementi sono modificabili, come ad esempio dei campi di input, allora i dati passeranno dal video alla sorgente.

Tutte le operazioni indicate sono sincronizzate in tutti gli stadi. Se, ad esempio, un documento della collection fornita come sorgente dati viene marcato per la cancellazione, la datamap lo rileva automaticamente, aggiorna l'insieme dei dati visibili rimuovendo i link con il documento in fase di cancellazione e distrugge il clone del template che lo mostrava a video.

Creazione di liste di documenti

Vediamo in pratica come utilizzare una datamap per visualizzare una lista di prodotti presenti nel database. Nell'immagine seguente vediamo tutto quello che serve per realizzarlo.



L'elemento selezionato nell'immagine è proprio una datamap che è stata aggiunta tramite il menù *+datamap* dell'oggetto videata. Tra le proprietà della datamap, possiamo notare il *Tipo contenuto* impostato a *Product* per indicare che la datamap conterrà dei documenti *Product*, il flag *AutoLoad* che vale *true* per consentire alla datamap di caricare i dati in autonomia ed infine il *Template* impostato all'elemento *item*, quello che contiene una riga della lista.

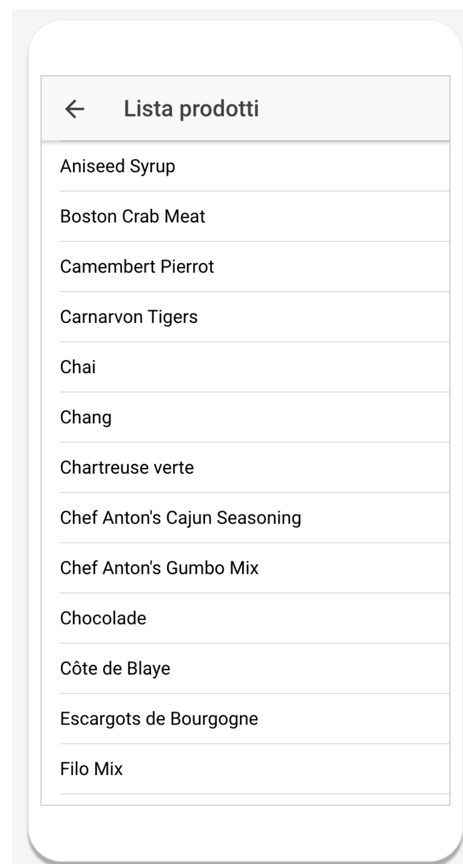
Selezionando nell'albero l'elemento *label*, possiamo notare che la proprietà *Sorgente dati* è stata impostata a *ProductName* per indicare che il nome del prodotto deve essere usato come *innerText* dell'elemento *label*.

Quando la videata viene aperta, vengono eseguite dalla datamap le seguenti operazioni:

1) Siccome è stato selezionato un tipo di documenti, ma non è stata passata alla datamap alcuna collection, la datamap crea una collection come propria sorgente dati.
2) L'elemento passato come template viene rimosso dalla videata e memorizzato internamente dalla datamap come modello per generare le righe.
3) Visto che il flag *AutoLoad* è attivo, la datamap carica la collection usata come sorgente dati usando come ordinamento *ProductName*.
4) Tutte le righe caricate vengono puntate tramite l'array *rows* e anche tramite l'array *uiRows*. In questo momento, infatti, tutte le righe della sorgente dati sono visibili e hanno lo stesso ordine di quelle da visualizzare.

5) Per ogni riga da visualizzare, la datamap genera un clone del template e lo aggiunge alla lista. Il clone creato per una determinata riga viene memorizzato tra i dati della riga stessa.
6) All'interno del clone vengono impostate le proprietà visuali collegate alle proprietà della sorgente dati. In questo esempio, la proprietà *innerText* dell'oggetto *label* viene impostata al valore di *ProductName*.

Il risultato di questa serie di operazioni è il seguente:



Formattazione condizionale

Abbiamo visto che è possibile collegare le proprietà della sorgente dati direttamente alle proprietà degli elementi visuali e questo permette di mostrare per ogni clone del template i dati corrispondenti.

Nei casi reali, sono tante le eccezioni a questo meccanismo. Immaginiamo ad esempio di voler applicare un particolare formato a campi numerici o date, oppure di mostrare oggetti diversi nel template in base ai dati, o ancora di usare stili dipendenti dai dati. Per tutti questi casi, la datamap notifica l'evento *onRowComposition* che permette di intervenire sulle proprietà della riga in fase di composizione.

Immaginiamo, ad esempio, di voler colorare di rosso i prodotti sotto scorta. Dopo aver aggiunto l'evento *onRowComposition* alla datamap tramite il pannello degli eventi, come per ogni altro elemento visuale, possiamo scrivere il codice seguente:

```
$dmProdotti.onRowComposition = function (row, template)
{
  let sottoScorta = row.UnitsInStock < row.ReorderLevel;
  $label.style.color = sottoScorta ? "red" : "";
};
```

È interessante notare che tramite il parametro *row* è possibile accedere alle proprietà della sorgente dati della riga per cui viene notificato l'evento. Siccome la sorgente dati è una collection di documenti *Product*, le proprietà di *row* puntano a quelle del documento *Product* corrispondente.

Tenendo conto che è possibile accedere direttamente al documento stesso tramite la proprietà *row.document*, possiamo concludere che: *row.ProductName === row.document.ProductName,* ma non è vero che *row === row.document*, in quanto *row* rappresenta la riga della datamap ed è un oggetto di tipo *DataRow*, mentre *row.document* è un oggetto di tipo *Product*.

Un altro aspetto interessante è la notazione *$label*. Solitamente essa rappresenta il riferimento ad un elemento visuale presente nella videata. Quando invece essa viene usata all'interno dell'evento *onRowComposition*, rappresenta l'elemento all'interno del clone del template utilizzato per la composizione della riga. Siccome il clone del template può essere letto tramite la proprietà *row.element*, la notazione *$label* usata all'interno dell'evento *onRowComposition* corrisponde a *row.element.label*.

Notiamo infine che i vari cloni dei template possono essere riutilizzati dalla datamap in varie occasioni. Per questa ragione l'evento *onRowComposition* deve essere in grado di modificare gli oggetti in entrambi i sensi, impostando o resettando le proprietà. Quindi non è corretto scrivere:

```
if (sottoScorta)
  $label.style.color = "red";
```

ma è necessario usare la forma completa:

```
if (sottoScorta)
  $label.style.color = "red";
else
  $label.style.color = "";
```

E quindi in forma contratta: `$label.style.color = sottoScorta ? "red" : "";`

L'evento *onRowComposition* può essere asincrono e in tal caso non ne viene aspettato il completamento. Si consiglia di concentrare la maggior parte delle modifiche ai dati prima delle operazioni asincrone: in questo caso infatti le modifiche visuali richieste appariranno insieme alla riga, mentre quelle successive alle operazioni asincrone potrebbero apparire dopo che la riga è già stata visualizzata.

Caricamento dei dati

I dati di una datamap possono essere forniti in tre modi diversi:

1) Tramite documenti e collection
2) Tramite una query strutturata
3) Caricandoli direttamente in memoria.

Documenti e collection

Per caricare in una datamap documenti o collection, è necessario che sia stata impostata la proprietà *Tipo contenuto* alla classe di documenti da caricare. A questo punto, la seguente istruzione:

```
$dmProdotti.load();
```

causa il caricamento della collection interna usando i filtri impostati nella datamap, se presenti. Tale istruzione non è necessaria se la proprietà *AutoLoad* è attiva, ma in tal caso non si ha un controllo preciso del momento in cui avviene il caricamento.

È possibile fornire alla datamap una collection o un documento caricandolo da codice, come vediamo nell'esempio seguente:

```
var c = yield App.NBE.Product.loadCollection(app);
$dmProdotti.collection = c;
```

Se deve essere caricato un solo documento, possiamo anche operare così:

```
var p = yield App.NBE.Product.loadByKey(app,1);
$dmProdotti.document = p;
```

Query strutturata

Per caricare in una datamap il risultato di una query, non deve essere stato impostato il tipo contenuto e deve essere stato aggiunto un oggetto query alla datamap tramite il comando *+query* dell'oggetto datamap. In questo caso le colonne della query diventano le proprietà della sorgente dati.

Anche in questo caso il caricamento può avvenire automaticamente se la proprietà *AutoLoad* è attiva oppure tramite codice con l'istruzione: `$dmProdotti.load();`

La query può referenziare sia proprietà di applicazione che proprietà della videata o di elementi visuali della videata. Se tali proprietà cambiano, la datamap non si aggiorna automaticamente, ma è necessario usare il comando: `$dmProdotti.reload();`

Caricamento in memoria

Il caricamento in memoria può avvenire se non è stato impostato il tipo di contenuto e non è stato aggiunto un oggetto query alla datamap. In questo caso è necessario definire le

proprietà della sorgente dati con il comando *+campo* della datamap e poi caricare i dati in memoria con il metodo *add*. Vediamo un esempio:

```
for (var i = 0; i < 100; i++) {
  $dmProdotti.add({ProductName : "Prodotto " + i});
}
```

Eventi coinvolti nel ciclo di caricamento dei dati

Quando la datamap esegue un caricamento di dati, notifica i seguenti eventi:

- *beforeLoad*: prima di effettuare il caricamento. Nel codice di gestione dell'evento è possibile modificare i filtri di caricamento.
- *afterLoad*: subito dopo il caricamento. Nel codice di gestione è possibile scorrere il contenuto della sorgente dati.
- *onDataChange*: al termine del ciclo di caricamento il contenuto della datamap cambia e quindi viene notificato l'evento *onDataChange* che permette di adattare la videata al contenuto attuale della datamap.

Filtratura dei dati caricati dalla datamap

Prima di inviare il comando di caricamento o durante l'evento *beforeLoad* è possibile aggiungere o togliere criteri di filtro, tramite i seguenti metodi della datamap:

- *addFilter*: aggiunge un filtro per la colonna specificata. I filtri sono espressi tramite la sintassi *query by example* indicata di seguito.
- *clearFilter*: rimuove un filtro da una colonna.
- *clearFilters*: rimuove tutti i filtri dalla datamap.

Tutti i filtri inseriti per default sono applicati in *and*. Se per uno o più filtri viene attivata l'opzione *or*, essi vengono posti in *or* fra di loro ed in *and* con tutti gli altri.

È possibile specificare il filtro nel seguente formato:

- *valore:* (ad esempio "abc"): se i dati sono di tipo carattere viene usato l'operatore *like*. Se sono tipo *data* verranno trovati quelli compresi nel giorno indicato. Se sono numerici, verranno trovati quelli uguali al valore indicato.
- *=valore* (ad esempio "=abc"): verranno trovati tutti i dati uguali al valore indicato, indipendentemente dal tipo.
- *>valore* (ad esempio ">abc"): verranno trovati tutti i dati maggiori al valore indicato.
- *<valore* (ad esempio "<abc"): verranno trovati tutti i dati minori al valore indicato.
- *valore1:valore2* (ad esempio "a:b"): verranno trovati tutti i dati compresi tra i due valori, estremi compresi.
- *valore:* (ad esempio "b:"): verranno trovati tutti i dati maggiori o uguali del valore indicato.
- *:valore* (ad esempio ":b"): verranno trovati tutti i dati minori o uguali del valore indicato.
- *#valore* (ad esempio "#abc"): verranno trovati tutti i dati diversi dal valore indicato. Se i dati sono di tipo carattere verranno trovato tutti quelli che non iniziano per il valore indicato.
- *!* (punto esclamativo): verranno trovati tutti i record per cui la colonna è vuota cioè vale *null*.

- *.* (punto): verranno trovati tutti i record per cui la colonna non è vuota cioè non vale *null*.
- *valore* (ad esempio *abc*): se la colonna è di tipo carattere è possibile utilizzare il carattere * oppure il carattere % per indicare il tipo di ricerca in sottostringa.
- *criteri multipli* (ad esempio "10:50;>100"): è possibile indicare più criteri di ricerca separati dal carattere ; (punto e virgola) che saranno considerati in *or*. Questo non è possibile se il primo carattere del filtro è = (uguale).

Se ad esempio si desidera selezionare solo i prodotti il cui nome inizia per *ch*, è possibile utilizzare le seguenti righe di codice:

```
$dmProdotti.clearFilters();
$dmProdotti.addFilter("ProductName","ch");
$dmProdotti.reload();
```

Gestione del contenuto della sorgente dati

Indipendentemente dal modo con cui la datamap è stata caricata, è possibile leggere e modificare la sorgente dati come segue.

Lettura del contenuto della sorgente dati

L'array *rows* della datamap contiene un oggetto *DataRow* per ogni riga della sorgente dati. In caso di query o caricamento in memoria, i dati sono memorizzati direttamente nell'array, mentre in caso di collection o documenti i dati vengono referenziati.

Per leggere tutte le righe della datamap, ad esempio per sommare le quantità dei prodotti a magazzino, è possibile usare il seguente codice:

```
let somma = 0;
for (let i = 0; i < $dmProdotti.length; i++) {
  somma += $dmProdotti.rows[i].UnitsInStock;
}
```

Si noti che la proprietà *length* della datamap restituisce la lunghezza dell'array *rows*. Se invece si desidera sommare solo le righe corrispondenti ai prodotti visibili a video, si può scorrere l'array *uiRows*:

```
let somma = 0;
for (let i = 0; i < $dmProdotti.count; i++) {
  somma += $dmProdotti.uiRows[i].UnitsInStock;
}
```

In questo caso si usa la proprietà *count*, che restituisce la lunghezza dell'array *uiRows*.

Per modificare il contenuto della sorgente dati senza eseguire query, sono disponibili i seguenti metodi della datamap:

- *add*: aggiunge una riga alla sorgente dati (e un documento alla collection se la datamap è basata su documenti).

- *insert*: aggiunge una riga alla sorgente dati e la marca per l'inserimento (e aggiunge un documento alla collection se la datamap è basata su documenti).
- *remove*: rimuove una riga dalla sorgente dati. Essa viene rimossa anche dal video, ma non viene cancellata dal database. Per effettuare una cancellazione, occorre marcare una riga per la cancellazione.
- *clear*: svuota la sorgente dati e rimuove tutti i cloni del template dal video.

Nota bene: gli array *rows* e *uiRows* devono essere considerati in sola lettura. Non deve essere mai modificato direttamente il loro contenuto, altrimenti si andrà incontro ad errori o eccezioni.

Ricerca e selezione dei dati

Immaginiamo di voler effettuare una ricerca direttamente in memoria, visualizzando solamente le righe della sorgente dati che rispettano determinati criteri. Per ottenere questo risultato è possibile utilizzare la proprietà *hidden* degli oggetti *DataRow* contenuti nell'array *rows* della datamap.

Se, ad esempio, aggiungiamo una search bar alla videata che mostra la lista dei prodotti, il codice per filtrare i dati in memoria ricercando per nome potrebbe essere il seguente:

```
$searchbar.onInput = function (event)
{
  for (let i = 0; i < $dmProdotti.length; i++) {
    let match = $dmProdotti.rows[i].ProductName.includes(this.value);
    $dmProdotti.rows[i].hidden = !match;
  }
};
```

Attivando la proprietà *hidden* per una determinata riga, essa viene rimossa dall'array *uiRows* e il corrispondente clone del template viene eliminato dal video. Disattivando la proprietà *hidden*, la riga viene nuovamente inserita nell'array *uiRows*, viene creato un nuovo clone del template di riga, e poi esso viene personalizzato con i dati della riga ed inserito a video nel punto giusto.

In casi come questo è possibile usare il metodo *find* della datamap, che consente di selezionare una serie di righe che rispettano determinati criteri e su di esse impostare il valore di una proprietà in funzione del fatto che esse rispettino o meno tali criteri. Il codice dell'esempio precedente potrebbe essere riscritto come:

```
$searchbar.onInput = function (event)
{
  $dmProdotti.find("ProductName like '%"+this.value+"%'", "visible");
};
```

Il funzionamento è il seguente: per tutte le righe della datamap viene attivata la proprietà *hidden* (che in questo caso è considerata in modo negato, tramite il nome *visible*) se la riga non contiene il testo inserito nella search bar.

Il metodo *find* si presta a ricerche di vario tipo nella sorgente dati. Ad esempio se volessimo trovare i prodotti della categoria 1, potremmo usare la seguente riga di codice:

```
var prodArray = $dmProdotti.find("CategoryID = 1");
```

Oppure, potremmo trovare il primo prodotto il cui nome inizia per A con la seguente istruzione:

```
var ap = $dmProdotti.find("ProductName like 'A%')[0];
```

Il metodo *find* esegue un ciclo sull'array *rows*, quindi ha complessità O(n). Se si utilizza *find* per effettuare ricerche puntuali per una chiave, è possibile ottimizzare le performance tramite il metodo *prepareRowMap* che, in questi casi, rende la complessità O(1).

Ricerca tramite caricamento dal database

Negli esempi precedenti la ricerca viene effettuata direttamente in memoria, avendo caricato nella sorgente dati tutte le righe da controllare. Questa modalità si può applicare sono in alcuni casi, ovvero quando il dataset è piccolo o non è contenuto in un database.

Se si deve implementare la selezione di dati su un grande numero di record, la scelta migliore è quella di caricare solo i dati che soddisfano i criteri di selezione. Nel caso precedente, il codice da utilizzare potrebbe essere il seguente:

```
$searchbar.onInput = function (event)
{
  $dmProdotti.clearFilter("ProductName");
  $dmProdotti.addFilters("ProductName",this.value);
  yield $dmProdotti.reload();
};
```

Durante l'operazione di *reload*, la datamap non svuota e ricarica l'array delle righe a video ma aggiorna le righe attuali con i nuovi dati. In questo modo le righe presenti a video non vengono distrutte e ricreate, ma solamente aggiornate in modo da annullare l'effetto di sfarfallamento visuale.

Ordinamento delle righe

È possibile specificare sia a design time che a runtime l'ordinamento dei dati di una datamap tramite la proprietà *orderBy*, sia specificando i nomi delle colonne che i loro indici. Dopo averla impostata, essa verrà considerata al caricamento dati successivo, oppure quando viene chiamato il metodo *sort*.

Se, ad esempio, volessimo ordinare la lista di prodotti in ordine decrescente di nome, potremmo utilizzare le seguenti righe di codice:

```
$dmProdotti.orderBy = "ProductName desc";
$dmProdotti.sort();
```

Il metodo *sort* ricalcola la posizione delle righe nell'array *uiRows* e poi sincronizza gli elementi visuali; l'array *rows* non viene modificato. Per tutte le righe visibili viene notificato l'evento *onRowComposition*.

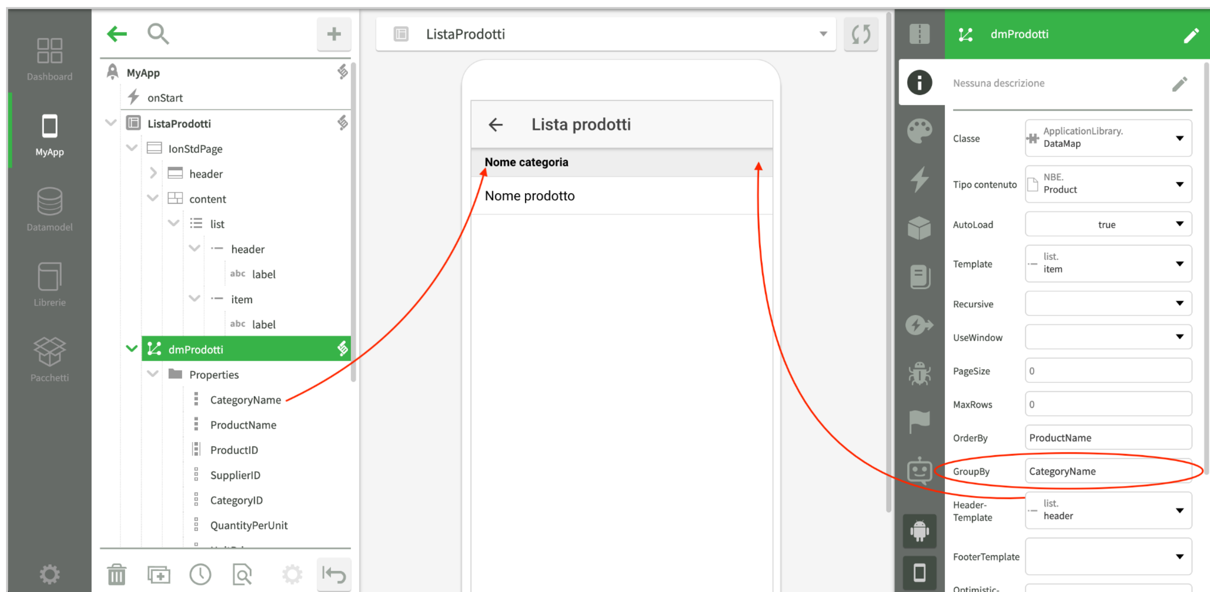
Raggruppamento delle righe

Le datamap contengono un complesso sistema di raggruppamento delle righe, basato sui seguenti presupposti:

1) Oltre al template per le righe, alla datamap vengono forniti a design time i template per l'intestazione e per il piede dei gruppi.
2) I raggruppamenti vengono impostati a design time o a runtime tramite la proprietà *groupBy*.
3) Al successivo caricamento o chiamata del metodo *sort*, la datamap genera una gerarchia di gruppi e per ognuno di essi crea un clone del template di intestazione, genera poi le righe del gruppo ed infine crea un clone del template del piede.
4) Il tutto viene ripetuto per tutti i livelli di raggruppamento.

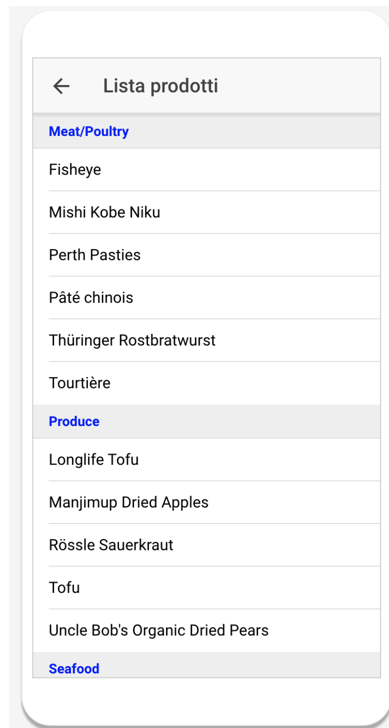
Se i dati sorgente di una riga variano in modo che essa dovrebbe cambiare gruppo, questo cambiamento avrà luogo al prossimo caricamento o chiamata al metodo *sort*.

Nella seguente immagine vediamo come creare una lista di prodotti raggruppata per categoria di prodotto.



Fra le proprietà della datamap possiamo notare l'impostazione di *GroupBy* e di *HeaderTemplate*. Inoltre la proprietà derivata *CategoryName* viene usata come sorgente dati della label contenuta nell'elemento *header*.

Il risultato di questa impostazione è il seguente:



Possiamo notare che il template per l'intestazione o il piede dei gruppi è uno solo, anche se i livelli di raggruppamentio sono più di uno. In questo caso la datamap genera i corrispondenti elementi di intestazione o piede clonando sempre il medesimo template e poi chiama l'evento *onHeaderComposition* per le intestazioni e *onFooterComposition* per i piedi.

Questi eventi sono analoghi a *onRowComposition*, ma invece che configurare il layout della riga, permettono di configurare il layout dell'intestazione o del piede. Ad essi infatti viene passato un oggetto di tipo *DataGroup* che contiene tutti i dettagli del gruppo in fase di configurazione.

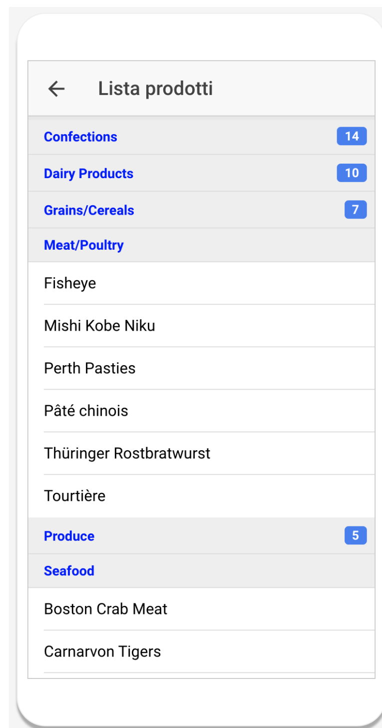
Come esempio di uso dell'evento *onHeaderComposition*, immaginiamo di voler visualizzare il numero di prodotti per categoria, ma solo se il gruppo è espanso, cioè mostra le righe al suo interno. A tal fine potremo scrivere il seguente codice nell'evento *onHeaderComposition*.

```
$dmProdotti.onHeaderComposition = function (group, template)
{
  $productCount.innerText = group.cardinality;
  $productCount.visible = !group.expanded;
};
```

E nell'evento *onClick* dell'elemento *header* (che funge da template dell'intestazione dei gruppi):

```
$header.onClick = function (event)
{
  this.group.expanded = !this.group.expanded;
};
```

Il risultato di questa impostazione sarà il seguente:



Cliccando sulle intestazioni i gruppi si aprono o chiudono ed in questo caso mostrano il numero dei prodotti che contengono.

Ottimizzazione delle performance

In questo paragrafo verranno illustrate tecniche e consigli per ottimizzare la visualizzazione delle datamap quando il numero di record da mostrare è alto, cioè oltre i 100.

Massimo numero di record

Se la datamap deve caricare i dati tramite una query su una tabella molto grande con criteri che non consentono di limitare in modo sicuro il numero di record, si consiglia di impostare la proprietà *maxRows* della datamap, che impone un limite superiore al numero di record estratti.

Se ad esempio dovessimo estrarre i prodotti da una tabella di un milione di referenze filtrandoli per nome, la ricerca per una sola lettera potrebbe estrarne più di centomila. Imponendo un limite superiore, ad esempio 1000, si evita di sovraccaricare il database, la memoria del server e la capacità di visualizzazione del browser.

Visualizzazione incrementale

Un’altra tecnica di ottimizzazione è quella di impostare la proprietà *pageSize* della datamap ad un valore maggiore di 0, solitamente compreso fra 30 e 50. In questo modo viene attivata una modalità di visualizzazione incrementale, in cui non vengono creati i cloni del template

per tutte le righe della lista *uiRows*, ma solo per il primo blocco di ampiezza pari alla dimensione della pagina, cioè al valore di *pageSize*.

Quando l'utente scorre la lista visualizzando le parti non ancora caricate, la datamap invierà ulteriori blocchi di righe, estendendo così la parte visibile della lista. Questo meccanismo è sufficientemente veloce per consentire una buona esperienza utente nella maggior parte dei casi.

Di seguito vengono indicate alcune limitazioni e considerazioni importanti che è necessario tenere presenti in relazione alla visualizzazione incrementale. In primo luogo viene ottimizzata solo la presentazione visuale dei dati, non il caricamento. Se, ad esempio, si esegue una query di caricamento di un milione di record, si otterranno tempi lunghi e sovraccarico di memoria sul server, e la maggior parte di questi dati record non verranno mai visti dall'utente.

Inoltre la visualizzazione incrementale è adatta solo quando l'utente vuole scorrere la lista dall'alto verso il basso senza saltare le parti intermedie. Se, ad esempio, si volesse passare immediatamente all'ultima parte della lista, il sistema dovrebbe visualizzare tutte le righe, perdendo così l'ottimizzazione ottenuta.

La visualizzazione incrementale, infine, richiede che la lista sia contenuta all'interno di un blocco scorrevole, in quanto il framework deve controllare gli eventi di scrolling del browser per caricare le parti mancanti della lista. Questo avviene per default quando si usa il template grafico *IonicUI* in modo standard, in quanto la lista viene sempre inserita all'interno di un elemento di tipo *IonContent* che per sua natura permette al suo contenuto di scorrere, ma potrebbe non essere vero in tutti i casi. Se le parti successive della lista non vengono caricate, occorre quindi correggere la struttura della pagina impostando lo stile *overflow:auto* o equivalente sul contenitore della lista.

Nonostante queste limitazioni e considerazioni, dopo aver limitato il numero di righe per evitare sovraccarichi della memoria del server, la visualizzazione incrementale è il metodo più usato per ottimizzare il rendering delle liste perché è semplice da utilizzare e solitamente efficace.

Visualizzazione incrementale e raggruppamenti

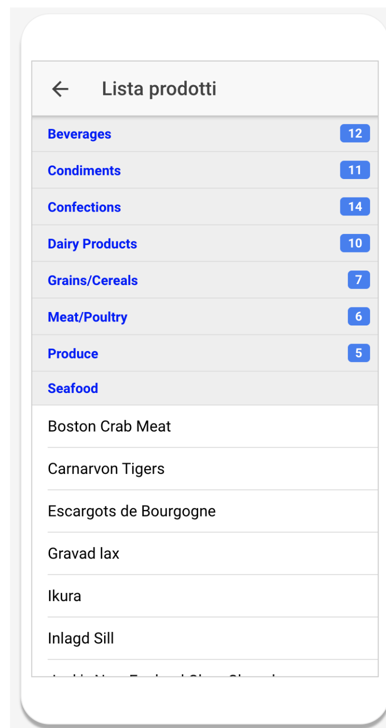
Una caratteristica quasi unica di Instant Developer Cloud è la possibilità di utilizzare la visualizzazione incrementale anche in caso di raggruppamenti con espansione dei gruppi.

Come abbiamo visto nel paragrafo precedente, la datamap gestisce la proprietà *expanded* dei gruppi di dati, permettendo cioè di mostrare o meno il contenuto di un gruppo. Tramite la proprietà *expandedGroupLevels* della datamap, impostabile a design time o a runtime, è possibile decidere quali gruppi devono essere espansi subito dopo il caricamento. Come valore di default vengono espansi tutti; inserendo -1 nessun gruppo verrà espanso; con 0 verrà espanso solo il primo livello.

Se per una datamap raggruppata si attiva la visualizzazione incrementale, essa non viene considerata semplicemente per le parti inferiori della lista, ma tiene conto anche

dell'espansione dei gruppi. Se, ad esempio, si imposta 10 come *pageSize* della lista dei prodotti raggruppata per categoria e -1 come proprietà *expandedGroupLevels,* verranno inizialmente mostrati tutti i gruppi perché il numero di righe totali mostrate a video è 0. Se a questo punto si espande un gruppo formato da più di 10 righe, il suo contenuto verrà caricato parzialmente, cioè verranno visualizzate solo le prime 10 righe, mentre le altre appariranno quando l'utente scorre verso il basso.

In questo modo è possibile permettere all'utente di accedere ad ogni parte della sorgente dati anche in caso di visualizzazione incrementale, come mostrato nell'immagine seguente:



Cliccando sul gruppo *Seafood* sono state visualizzate le righe relative all'ultimo gruppo, mentre tutti i gruppi precedenti sono ancora in stato non espanso e quindi essi non contengono ancora nessuna riga.

Caricamento dati incrementale

Un ulteriore strumento di ottimizzazione della datamap è il caricamento incrementale dei dati. Esso può essere attivato solo in caso di visualizzazione incrementale e consente di estrarre i dati dal database man mano che essi vengono richiesti per essere visualizzati.

L'attivazione del caricamento incrementale dei dati avviene tramite le proprietà *dataPageSize* e *dataPagingPreCount* e *dataPagingMode*. La prima stabilisce quanti dati devono essere estratti ogni volta. Si consiglia di utilizzare un multiplo del *pageSize* con un minimo di 100 o del doppio del *pageSize*.

dataPagingPreCount permette di conoscere subito il numero di dati totali, che verrà inserito nella proprietà *length* della datamap anche se l'array *rows* conterrà un numero di righe minori, al costo però di un'ulteriore query di conteggio dati.

dataPagingMode permette di scegliere fra due opzioni di caricamento incrementale, quella basata su *offset* e *limit* (consigliata) oppure quella basata sull'ordinamento delle chiavi (*keyset*).

Il meccanismo di funzionamento è il seguente: i dati vengono caricati dal database solo quando devono essere mostrati a video, inoltre essi non vengono caricati tutti, ma a blocchi utilizzando una delle due possibili tecniche indicate sopra. Al momento del caricamento della sorgente dati viene quindi estratto dal database solo il primo blocco di dati e, siccome è attiva anche la visualizzazione incrementale, solo una parte di questi dati sarà presentata a video. Man mano che l'utente scorre la lista, il framework richiede la visualizzazione di ulteriori dati, e se essi non sono ancora stati estratti, avverrà anche il caricamento del blocco dati successivo. Tutto questo avviene in automatico e con una velocità sufficiente a garantire una buona esperienza utente nella maggior parte dei casi.

Il caso d'uso più importante del caricamento incrementale è quando è difficile o impossibile limitare il numero di record totali da caricare, oppure quando l'accesso ai dati è più lento del solito, come avviene quando si accede ai dati via Cloud Connector.

Di seguito vengono indicate alcune limitazioni e considerazioni che è necessario tenere presenti in relazione al caricamento incrementale. In primo luogo, siccome i dati vengono estratti con query successive, se dopo aver caricato una pagina essi cambiano, si potrebbero ottenere delle anomalie. Se, ad esempio, dopo aver estratto la prima pagina di dati viene inserito un record che verrebbe posizionato proprio in tale pagina, il caricamento successivo conterrà anche l'ultimo della pagina precedente che verrà quindi duplicato a video.

Inoltre si deve tenere presente che la sorgente dati della datamap conterrà solo la parte caricata dei dati, quindi non sarà possibile scorrere l'array *rows* per tenere conto di tutti i dati estratti. Questo rende il caricamento incrementale **incompatibile** con la gestione dei raggruppamenti.

Liste virtuali

Le liste virtuali rappresentano una modalità di visualizzazione completamente diversa dalle precedenti. Finora, infatti, abbiamo visto che ogni riga visibile, presente nell'array *uiRows* ha una sua copia personalizzata del template di riga della datamap. Nel caso di visualizzazione incrementale questa copia viene creata solo quando l'utente scorre la lista, ma il meccanismo è sempre lo stesso.

Nel caso di lista virtuale, invece, la datamap gestisce un numero limitato di cloni del template di riga, li associa di volta in volta ad un determinato insieme di righe visibili e li posiziona a video nel punto corrispondente alle righe a cui sono state associate.

Quando l'utente si sposta nella lista, in funzione del punto in cui la lista è stata spostata, vengono ricalcolati sia l'associazione che il posizionamento delle righe, inoltre esse vengono nuovamente personalizzate in funzione dei nuovi dati a cui sono state associate.

L’attivazione della lista virtuale avviene impostando a *true* la proprietà *useWindow* della datamap.

Il caso d’uso di elezione delle liste virtuali è quando l’utente deve poter accedere istantaneamente a qualunque parte di una sorgente dati molto estesa. Immaginiamo ad esempio una lista di messaggi di log di almeno 10.000 elementi, in cui l’utente può scorrere avanti e indietro a suo piacimento. In questi casi la lista virtuale risolve il problema perché non sovraccarica mai il browser e contemporaneamente permette di accedere istantaneamente a qualunque parte della sorgente dati.

L’uso delle liste virtuali è tuttavia condizionato a diverse limitazioni:

1) Sono **incompatibili** con i gruppi e con il caricamento dati incrementale.
2) Il template deve avere dimensioni fisse rispetto ai dati che contiene.
3) La lista, non il suo contenitore, deve avere una dimensione fissata (solitamente al 100%). Ad essa il framework aggiunge gli stili *position:relative* ed *overflow:auto* in quanto deve essere scorrevole.

Il meccanismo di funzionamento delle liste virtuali necessita di un continuo lavoro di adattamento del contenuto del video man mano che l’utente scorre la lista. L’esperienza utente è ottimizzata per il caso offline, in cui il framework è in funzione nello stesso dispositivo in cui vengono mostrati i dati. Se invece si utilizza una lista virtuale in una applicazione online, quando lo scorrimento della lista è molto veloce possono apparire fenomeni di sfarfallamento perché il server non riesce ad inviare i nuovi dati abbastanza velocemente al browser.

Notiamo infine che l’evento *onRowComposition* viene notificato tutte le volte che un clone del template viene associato ad una diversa riga. Il codice di questo evento viene quindi mandato in esecuzione molte volte durante lo scorrimento della lista; si consiglia quindi di non effettuare operazioni asincrone per mantenere una buona esperienza utente.

Struttura del template

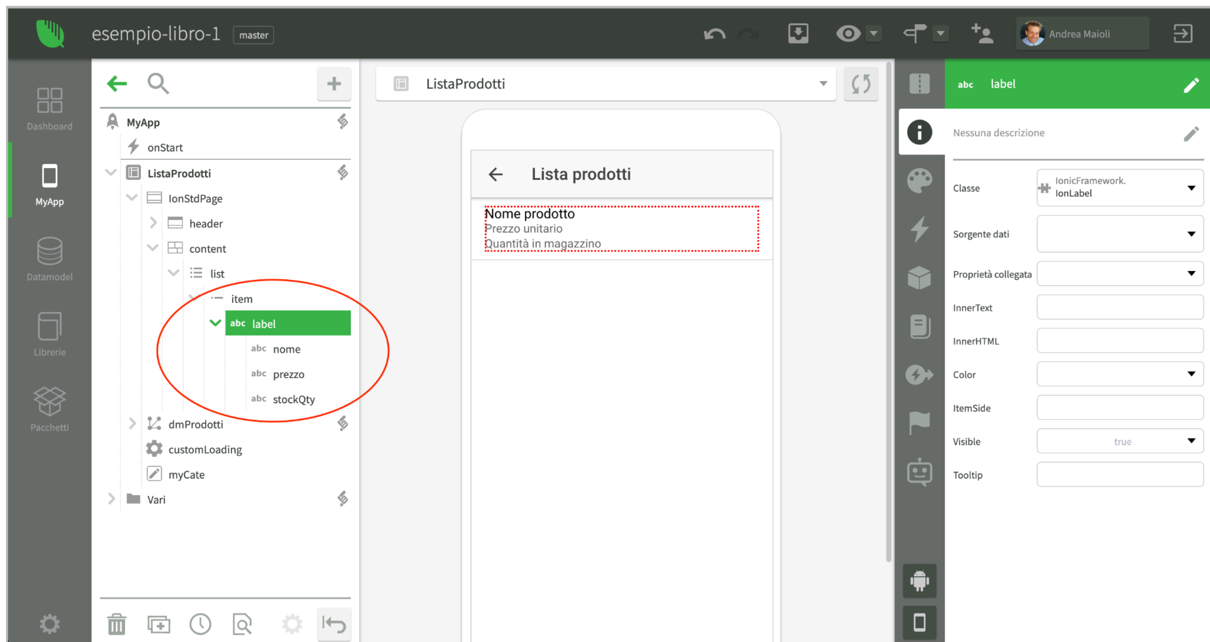
Una datamap ottimizza la fluidità dell’interfaccia piuttosto che la consistenza visuale perché l’utente perdona più facilmente il fatto che i dati appaiano in tempi leggermente diversi rispetto al fatto che l'applicazione non risponda immediatamente ai suoi comandi.

Per questa ragione le righe della datamap non vengono mostrate tutte contemporaneamente ma tramite la tecnica chiamata *renderAnimationFrame* che permette all’applicazione di essere sempre responsiva anche quando è in corso una costruzione visuale complessa.

Se però le righe appaiono con un effetto di scivolamento troppo pronunciato, è necessario ottimizzare le performance di visualizzazione semplificando la struttura del template di riga. Esso, infatti, viene clonato per ogni riga della datamap, quindi la sua composizione influisce sia sui tempi lato server che, soprattutto, sui tempi di visualizzazione nel browser.

L’ottimizzazione del template di riga consiste nella sua semplificazione: invece di creare una struttura di elementi innestati, è possibile utilizzare la proprietà *innerHTML* del contenitore della riga per renderizzarne immediatamente il contenuto. Includendo nella stringa HTML gli

ID opportuni, è possibile intercettare il clic dell'utente anche se avviene nei nodi interni al contenitore. Vediamo un esempio di questa ottimizzazione.



In questo esempio vediamo che il template di riga (*item*) contiene una *label* che a sua volta contiene tre elementi testuali, ognuno collegato ad una proprietà della datamap. Questo permette di comporre la riga in modo semplice perché ogni elemento è collegato alla corrispondente proprietà della datamap.

Se fosse necessario ottimizzare questo template sarebbe possibile rimuovere i tre elementi testuali contenuti nella *label* e, a questo punto, inserire la seguente riga di codice nell'evento *onRowComposition*.

```
let html = t("<span>@nome</span><p>@prezzo</p><p>@stock</p>", {
  nome: App.Utils.htmlEscape(row.nome),
  prezzo: App.Utils.htmlEscape(row.unitPrice),
  stock: App.Utils.htmlEscape(row.stockQuantity)
});
//
$label.innerHTML = html;
```

L'uso di due soli elementi visuali (*item* e *label*) come template di riga invece che di cinque migliora le performance di rendering della lista, ma richiede l'uso della proprietà *innerHTML*. In questo caso è necessario proteggere il sistema da attacchi di tipo XSS injection che possono avvenire se non si controlla accuratamente il contenuto dei dati che provengono dal database e che vengono inseriti direttamente come contenuto HTML della pagina. A tal fine è possibile utilizzare il metodo *htmlEscape* che provvede un primo livello di sanificazione del contenuto testuale passato come parametro.

Eventi di composizione

Un'ultima strategia di ottimizzazione delle performance della datamap consiste nella corretta implementazione dell'evento *onRowComposition*, che viene notificato almeno una volta per ogni riga della lista.

È molto importante che la gestione di questo evento non implichi l'esecuzione di query. Proprio per la natura asincrona di questo evento, infatti, tutte le query in esso contenute verranno eseguite in modalità autocommit causando così una serie di transazioni che possono sovraccaricare il database.

È questo un caso piuttosto frequente quando si devono mostrare dati provenienti da tabelle diverse: se, ad esempio, nella lista dei prodotti si vuole mostrare il nome della categoria a cui il prodotto appartiene, una possibile soluzione potrebbe essere quella di caricare il documento *Categoria* referenziato dal prodotto durante evento *onRowComposition,* e poi inserire il nome della categoria risultante in un elemento visuale del template. Questo può causare una query per ogni riga, anche se questo problema può essere mitigato dall'uso di *getRelated* o dell'opzione di caricamento con cache.

La soluzione migliore, tuttavia, è quella di utilizzare una proprietà di documento derivata in modo che il nome della categoria sia parte del documento *Product* fino dal momento del suo caricamento. In questo modo non è nemmeno necessario l'evento *onRowComposition*, in quanto l'informazione è già presente nel documento stesso.

La necessità di ottimizzare l'implementazione dell'evento *onRowComposition* si può rilevare dal fatto che le righe della lista appaiono subito, ma che alcune informazioni vengono mostrate dopo qualche istante. Inoltre è facile rilevare dalla console di debug dell'IDE che vengono effettuate troppe query: a volte viene addirittura interrotta la raccolta dei dati di debug per non sovraccaricare il sistema.

In questi casi è necessario evitare le query (e in generale le operazioni asincrone) nel codice dell'evento *onRowComposition*. Il recupero delle informazioni può avvenire direttamente a livello di documenti o di query della datamap, oppure nell'evento *afterLoad* in cui il contenuto della sorgente dati può essere completato tramite codice personalizzato.

Gestione del template visuale della lista

In questo paragrafo vengono illustrati gli aspetti principali del rapporto fra i template di riga e di gruppo e la datamap.

Struttura del template

In tutti gli esempi mostrati finora abbiamo utilizzato una datamap per mostrare liste di record. Tuttavia l'aspetto visuale della lista dipende solo dal tipo di contenitore e dalle caratteristiche del template. Se, ad esempio, il template di riga ha come proprietà di stile *display: inline-block* o se la lista è di tipo *flex* a scorrimento orizzontale, al posto di una lista verticale otterremo delle visualizzazioni in linea o a griglia. Se il template ha come proprietà di stile *position:absolute* potremo posizionare dei marker su una mappa.

Anche la struttura interna del template è completamente libera: è possibile inserire all'interno qualunque tipo di elemento visuale - fra i quali immagini, mappe, grafici, calendari - e con qualunque struttura interna. In questo modo la datamap consente una completa personalizzazione del risultato grafico.
Per quanto riguarda il rapporto fra i template di gruppo e di riga, vengono supportate due diverse tipologie:

1) Il template di riga è contenuto all'interno del template di gruppo.
2) Il template di riga è allo stesso livello del template di gruppo.

Collegamento tra datamap e template

Al momento dell'apertura della videata, la datamap acquisisce gli elementi definiti come template staccandoli dal resto della pagina per poi utilizzarli internamente come clone per generare la visualizzazione delle righe o delle sezioni di gruppo.

Quando i template vengono clonati, ad ogni elemento visuale del template viene attaccata la proprietà *row* nel caso di clone del template di riga o *group* se il template viene clonato per un gruppo. La proprietà *row* rappresenta la riga (*DataRow*) per la quale tale clone è stato generato; in modo analogo la proprietà *group* rappresenta il gruppo (*DataGroup*).

In questo modo è facile scrivere codice che dipende dai dati della riga. Se, ad esempio, nella lista dei prodotti volessimo inserire un evento di clic per mostrare il prezzo del prodotto tramite un messaggio di alert, potremmo semplicemente scrivere così:

```
$item.onClick = function (event)
{
  app.alert("Il prezzo del prodotto " + this.row.ProductName + " è " +
            this.row.UnitPrice);
};
```

L'espressione *this.row* rappresenta la riga (*DataRow*) collegata all'elemento cliccato *this*.

Aggiornamento di elementi esterni al template

In questo capitolo abbiamo visto che la datamap mantiene sincronizzato lo stato della sorgente dati con gli elementi visuali usati per rappresentarla. Tuttavia ci sono casi in cui è importante poter aggiornare anche elementi esterni al template di riga o di gruppo, ad esempio per mostrare il numero di righe contenute nella lista.

In tutti i casi in cui è necessario sincronizzare lo stato della sorgente dati con elementi esterni ai template, si consiglia l'utilizzo dell'evento *onDataChange* della datamap. Questo evento viene notificato tutte le volte che il contenuto della sorgente dati o della parte visibile della stessa cambia. Il cambiamento può avvenire non solo perché i dati vengono caricati dal database, ma anche perché vengono inseriti, modificati o cancellati.

Se, ad esempio, volessimo mostrare nel titolo della videata il numero di prodotti contenuti nella lista, potremmo scrivere il codice seguente:

```
$dmProdotti.onDataChange = function ()
{
  $title.innerText = $dmProdotti.count + " prodotti trovati";
};
```

In questo modo l'aggiornamento del risultato avviene automaticamente tutte le volte che nella datamap cambia il contenuto della sorgente dati.

Modifica e salvataggio dei dati

Il collegamento tra la sorgente dati e gli elementi visuali creati per visualizzare le righe non è in sola lettura come visto negli esempi precedenti, ma è bidirezionale. Ciò significa che tramite gli elementi visuali è possibile modificare la sorgente dati.

Sebbene sia possibile modificare qualunque tipo di sorgente dati, sia in memoria che basata su query o su documenti e collection, si consiglia di limitarsi a sorgenti dati basate su documenti e collection, perché ciò permette una gestione completa del ciclo di vita della transazione.

La modifica può avvenire in due modi. In primo luogo è possibile collegare un elemento visuale modificabile direttamente alla sorgente dati tramite la proprietà *datasource* dell'elemento; ad esempio si può usare un elemento *IonInput* collegato alla proprietà *UnitPrice* della sorgente dati per modificare al volo il prezzo di un prodotto della lista.

Quando l'utente modifica il contenuto dell'input, il nuovo valore viene propagato fino alla riga della sorgente dati corrispondente e, nel caso di documenti, viene anche chiamato il metodo *validate* con *reason:change*. Se l'evento di validazione genera errori collegati con le proprietà del documento, la datamap cerca di mostrarli a video impostando la proprietà *errorText* degli elementi visuali collegati alle proprietà del documento in stato di errore.

In ogni caso la datamap notifica l'evento *onValueChange* in modo che il codice dell'applicazione possa trattare la modifica anche in caso di sorgente dati basata su query o in memoria.

Il secondo modo per modificare la sorgente dati consiste nel manipolare direttamente il suo contenuto tramite la proprietà *this.row* oppure direttamente scrivendo i dati nell'array *rows* o *uiRows*. Se ad esempio si desidera inserire un pulsante per aumentare di una unità il prezzo del prodotto, si potrebbe scrivere il seguente codice:

```
$btnIncrease.onClick = function (event)
{
  this.row.UnitPrice++;
};
```

Si ricorda che la manipolazione diretta degli array *rows* e *uiRows* vale solo in caso di modifica, mentre per l'inserimento occorre utilizzare il metodo *insert* della datamap e per la cancellazione la proprietà *deleted* della *DataRow*.

Salvataggio delle modifiche

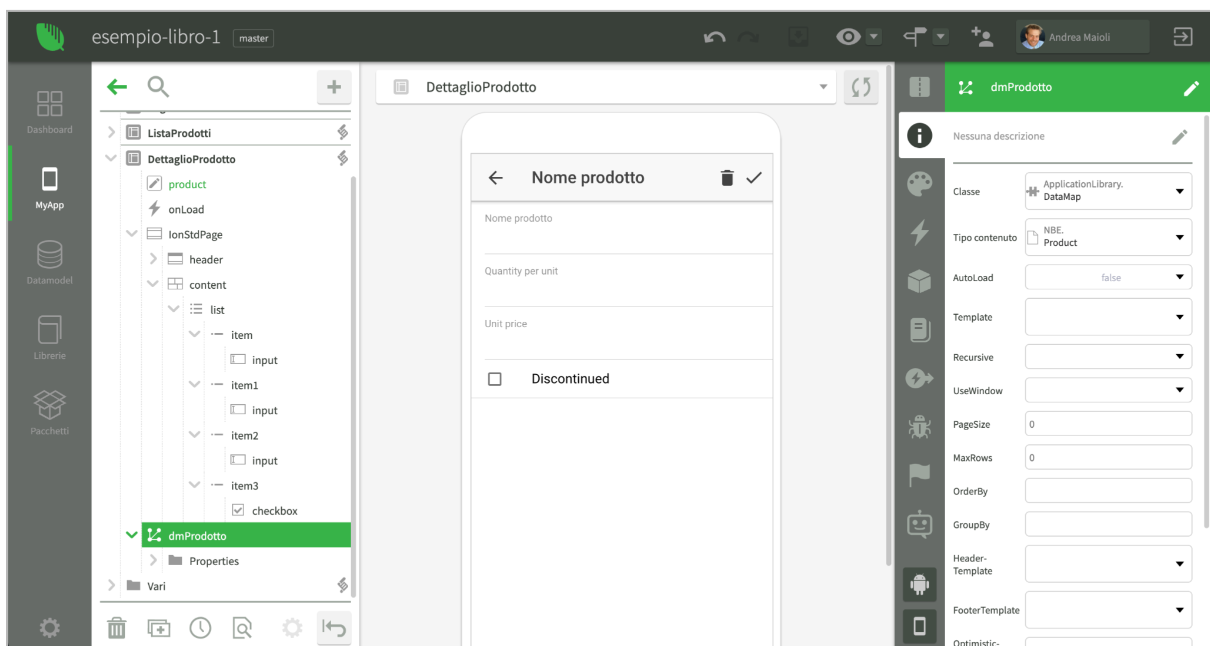
La conferma delle modifiche a livello di database avviene chiamando il metodo *save* della datamap. Se la datamap è basata su documenti o collection, il metodo *save* semplicemente esegue la chiamata corrispondente sul documento o sulla collection sottostante.

Nel caso di datamap basata su query, il metodo *save* esegue le operazioni direttamente sulla prima tabella della query (detta anche tabella master), inserendo, modificando o cancellando i dati e notificando gli eventi relativi. Si ricorda che il metodo consigliato per eseguire modifiche ai dati è l'utilizzo di documenti e collection.

Nel caso di sorgente dati in memoria, il metodo *save* notifica tutti gli eventi relativi al salvataggio, ma non esegue alcuna operazione automatica. Al termine del ciclo di salvataggio vengono resettati tutti i flag relativi allo stato di modifica delle righe e si procede alla rimozione delle righe cancellate.

Modifica dei documenti con videate di dettaglio

In questo paragrafo ripercorriamo i concetti illustrati nei paragrafi precedenti applicandoli ad un caso molto comune: la modifica di un documento tramite una videata di dettaglio. Nell'immagine seguente possiamo vedere una struttura di esempio.



La videata di dettaglio contiene i seguenti oggetti:

1) Una proprietà di videata che referenzia il documento da modificare (in questo caso è la proprietà *product*).
2) L'evento *onLoad* che permette di leggere il prodotto da modificare e di inizializzare la datamap.

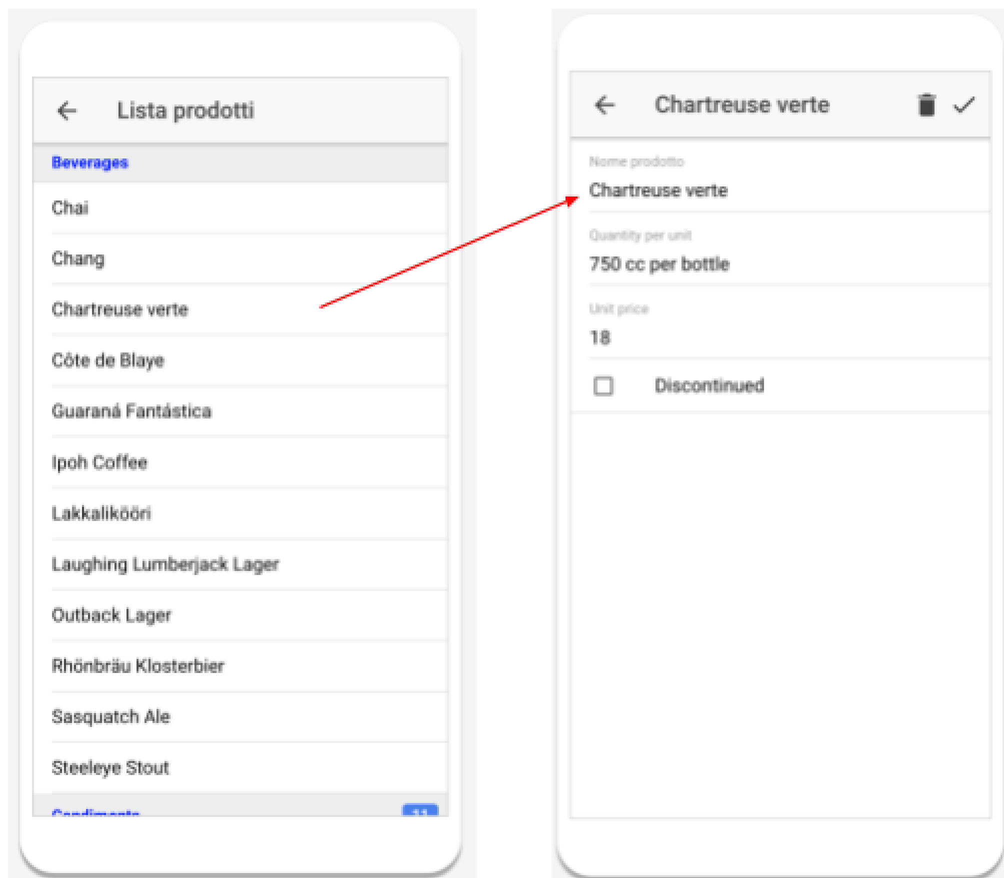
3) Gli elementi visuali che servono per modificare i dati (in questo caso tre *IonInput* e un *IonCheckbox*). Tali elementi sono collegati alle relative proprietà del documento *Product* tramite la datamap.
4) La datamap che permette di visualizzare e modificare il documento. In questo caso si chiama *dmProdotto*; si noti che nelle proprietà viene impostato il *Tipo contenuto* ma non viene indicato alcun *Template* di riga.
5) I pulsanti nell'intestazione per salvare le modifiche o cancellare il documento.

Per attivare il dettaglio, la videata *ListaProdotti* intercetta il clic su una riga della lista e apre la videata di dettaglio. Si noti che per navigare fra le videate viene usato *App.Pages,* un oggetto *NavigationController* che verrà illustrato nel capitolo relativo al framework *IonicUI*.

Il codice dell'evento *onClick* sull'item della lista è quindi il seguente:

```
$item.onClick = function (event)
{
  App.Pages.push(app, App.DettaglioProdotto, {doc : this.row.document});
};
```

Possiamo notare che fra le opzioni per la nuova videata viene passata anche la referenza al documento *Product* collegato alla riga della lista che viene cliccata (*this.row.document*).
L'effetto è il seguente:

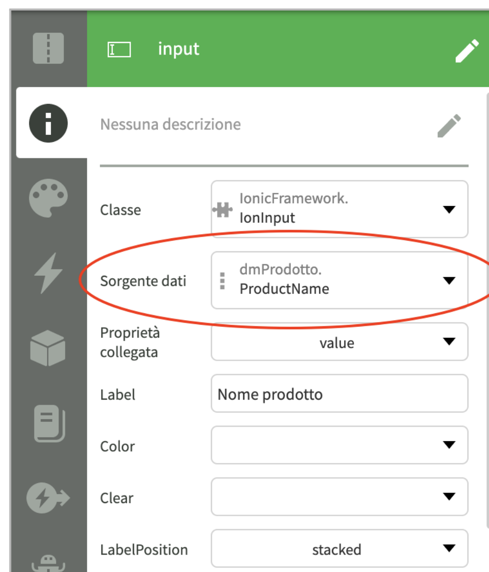


Il codice da scrivere per ottenere questo risultato è veramente limitato. Nell'evento *onLoad* della videata, viene semplicemente collegato il documento passato dalla lista alla datamap *dmProdotto*.

```
App.DettaglioProdotto.prototype.onLoad = function (options)
{
  view.product = options.doc;
  $dmProdotto.document = view.product;
  $title.innerText = view.product.ProductName;
};
```

È importante ricordare che per ottenere la visualizzazione dei dati del prodotto è necessario collegare a design time le proprietà della datamap agli elementi visuali corrispondenti tramite la *Sorgente dati*, come vediamo nell'immagine seguente relativa al nome del prodotto.

Inoltre notiamo che siccome la datamap non ha un template, essa non effettua alcuna operazione di clonazione per la generazione delle righe, ma semplicemente opera sugli elementi già esistenti nella videata. Ecco perché una datamap senza template è la struttura adatta ad una videata di dettaglio.



Vediamo adesso il codice dei pulsanti di conferma modifiche e di cancellazione:

```
$btnSave.onClick = function (event)
{
  let b = yield view.product.save();
  if (b) {
    App.Pages.pop(app);
  }
};
```

Si può notare che viene salvato direttamente il documento *product* invece che la datamap. Avremmo potuto salvare i dati anche scrivendo *$dmProdotto.save()*, ma abbiamo scelto una via più diretta. Se il salvataggio ha successo, la videata viene chiusa e si ritorna alla lista.

Vediamo ora il codice per cancellare il prodotto:

```
$btnDelete.onClick = function (event)
{
  view.product.deleted = true;
  yield $btnSave.onClick();
};
```

Esso viene marcato per la cancellazione e poi si procede come nel caso di salvataggio delle modifiche.

Si noti che il codice mostrato è solo esemplificativo. Nei casi reali occorre chiedere conferma e gestire eventuali errori di salvataggio o cancellazione.

Provando la videata a runtime è interessante notare che modificando il nome del prodotto nella videata di dettaglio, tale modifica viene riflessa automaticamente anche nella videata in lista, in quanto la stessa istanza del documento prodotto è parte della sorgente dati di entrambe le datamap, quella della lista (*dmProdotti*) e quella del dettaglio (*dmProdotto*). Se quindi una delle due datamap ne modifica le proprietà, anche l'altra se ne accorge e sincronizza la corrispondente parte visuale in autonomia.

Gestione delle modifiche pendenti

In queste videate può essere interessante controllare il pulsante *back* che permette di ritornare alla lista, presente nella videata al momento della creazione della pagina. Se infatti l'utente modifica i dati del prodotto e poi torna indietro senza prima confermare le modifiche, è opportuno avvertire che esse verranno perse, e poi effettivamente annullarle prima di tornare alla lista.

Il codice da utilizzare è il seguente:

```
$navbar.onBackButton = function (event)
{
  if (view.product.isModified()) {
    let b = yield app.confirm("Tornando alla lista perderai le modifiche.
                               Vuoi continuare?");
    if (!b)
      return;
    view.product.restoreOriginal();
  }
  App.Pages.pop(app);
};
```

L'evento *onBackButton* viene notificato quando l'utente preme il pulsante *back* nell'intestazione. Nell'implementazione mostrata, viene controllato se il documento è stato modificato e in tal caso si chiede conferma all'utente se vuole veramente uscire. Se l'utente risponde "No", l'evento termina senza chiudere la videata. Se invece l'utente conferma l'uscita, le modifiche al documento vengono annullate tramite il metodo *restoreOriginal* e poi la videata di dettaglio viene chiusa.

Gestione dello stato dei pulsanti dell’intestazione

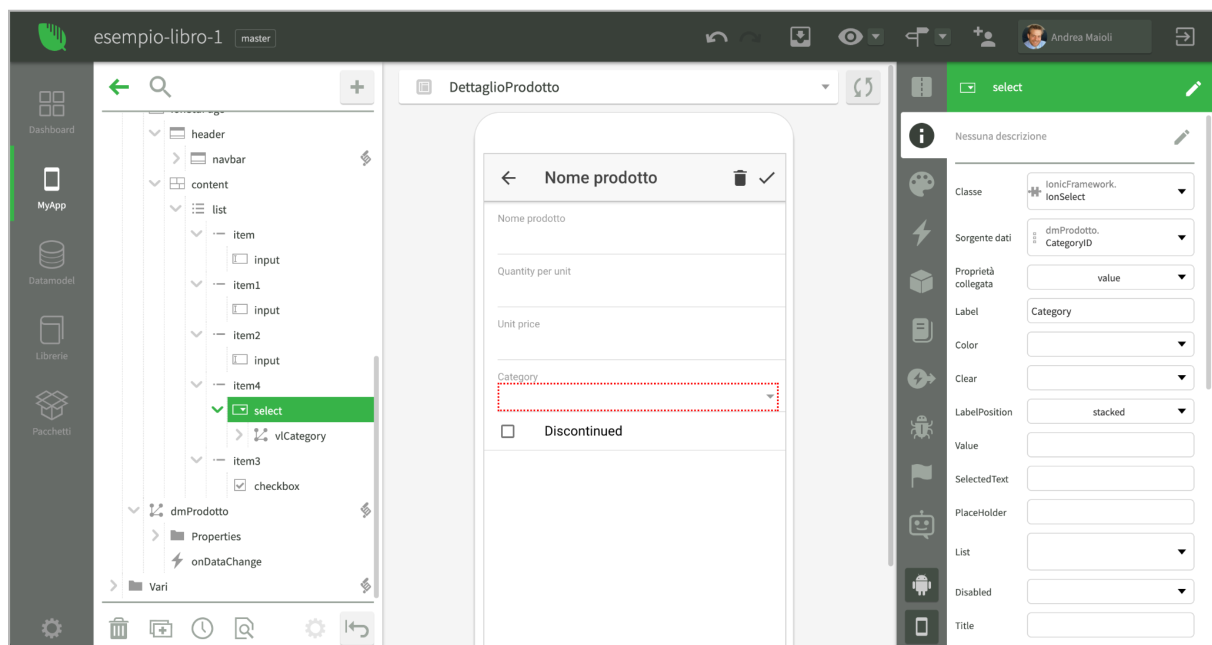
Un’ulteriore caratteristica che può migliorare l’esperienza utente è quella di visualizzare i pulsanti dell’intestazione in funzione dello stato del documento. In particolare, il pulsante di salvataggio può apparire solo se il documento è stato modificato, mentre quello di cancellazione solo se il documento non è stato modificato.

Per ottenere questo comportamento possiamo implementare l’evento *onDataChange*, che viene notificato ogni volta che cambia il contenuto della sorgente dati della datamap.

```
$dmProdotto.onDataChange = function ()
{
  let mod = view.product.isModified();
  $btnSave.visible = mod;
  $btnDelete.visible = !mod;
};
```

Selezione di dati provenienti da altre tabelle

Vediamo infine come implementare elementi visuali per la modifica dei dati quando essi sono basati su dati provenienti da altre tabelle. Ad esempio, possiamo implementare la selezione della categoria del prodotto tramite un elemento di tipo *IonSelect*.



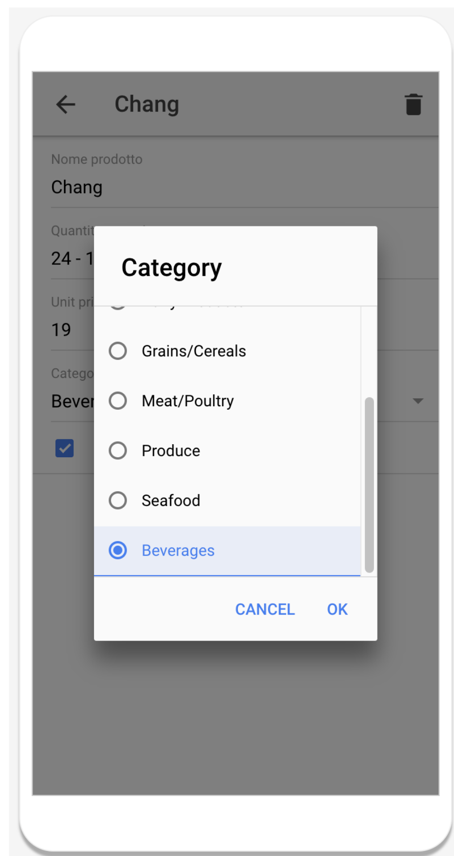
Nell’immagine possiamo notare che l’elemento *select,* di tipo *IonSelect*, ha come *Sorgente dati* la proprietà *CategoryID* del prodotto; inoltre esso contiene una datamap chiamata *vlCategory*, acronimo di lista valori delle categorie. Le uniche proprietà impostate di questa datamap sono *AutoLoad* a *true* e *Tipo contenuto* al documento *Category*.

Vediamo quindi un terzo tipo di utilizzo delle datamap: esse possono fungere da sorgente dati per un determinato elemento visuale. Possiamo riconoscere questa modalità di

funzionamento dal fatto che esse non compaiono al primo livello, subito sotto la videata, ma all'interno dell'elemento che devono servire.

Quando la videata si apre, la datamap *vlCategory* effettua il caricamento dei suoi dati e poi li invia come lista valori alla *IonSelect* che li utilizzerà per decodificare il valore di *CategoryID* del prodotto in fase di modifica, e anche per consentire all'utente di scegliere un'altra categoria.

Possiamo vedere il risultato nell'immagine seguente:



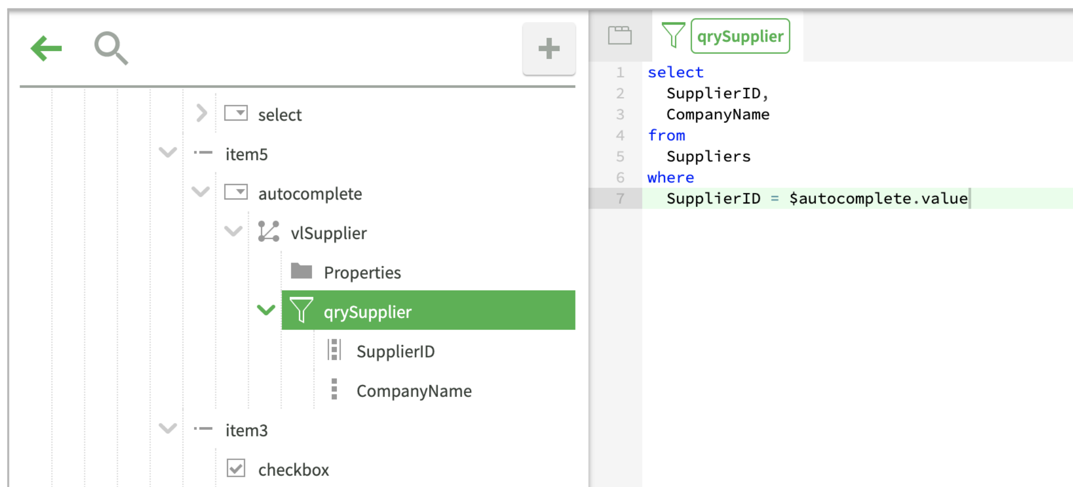
Notiamo che questo è un modo semplice di ottenere il risultato, ma è utilizzabile solo in alcuni casi. Infatti la datamap *vlCategory* carica i dati di tutte le categorie, che in questo caso sono solo 9 e quindi è accettabile. Inoltre non carica solo i campi *CategoryID* e *CategoryName*, ma tutte le colonne della tabella *Categories*. Un modo più performante può essere quello di utilizzare una query strutturata al posto di indicare il documento nella proprietà *Tipo contenuto* della datamap.

Possiamo notare che è possibile controllare le datamap usate come lista valori come tutte le altre datamap. Se, ad esempio, dovessimo implementare una select la cui lista valori dipende da quello che viene selezionato in un'altra select, non dovremmo fare altro che implementare l'evento *onChange* della seconda select e in esso ricaricare la datamap della prima select con i dati appropriati.

Uso di controlli di tipo autocomplete

Una soluzione più adeguata al caso di selezione dati da una tabella molto grande si può ottenere utilizzando un elemento *IonAutoComplete* che contiene una datamap basata su una query strutturata dipendente dal valore dell'elemento stesso, come mostrato nell'immagine seguente.

Le proprietà della datamap sono *AutoLoad* impostata a *true* e *maxRows* impostata a *30*. Al momento dell'apertura della videata la datamap esegue il caricamento e, siccome la query strutturata ha una dipendenza dal valore della proprietà *value* dell'elemento visuale *autocomplete*, essa installa un osservatore su tale proprietà: in questo modo tutte le volte che la proprietà cambia, la query verrà automaticamente eseguita.



La clausola *where* dipendente dal valore dell'autocomplete fa sì che la datamap esegua delle query mirate alla decodifica del valore della proprietà *SupplierID* caricando però solo i dati che servono. La query effettuata da *vlSupplier* all'apertura della videata è la seguente:

```
select SupplierID, CompanyName from Suppliers where SupplierID = 1 limit 30
```

È un caricamento con selezione per chiave primaria, accettabile anche se la tabella *Suppliers* avesse milioni di righe. Oltre alla visualizzazione della decodifica del valore, l'accoppiamento fra autocomplete e datamap attiva altre due modalità di funzionamento.

La prima modalità entra in gioco quando l'utente clicca sull'autocomplete se essa ha la proprietà *OpenOnFocus* attivata. In tal caso, infatti, l'utente si aspetta di vedere tutti i fornitori dell'archivio. Questo non è possibile se essi sono migliaia, ma comunque viene caricata la lista dei primi 30 in modo tale da soddisfare le aspettative dell'utente. Al momento del clic sull'autocomplete, la query eseguita è quindi la seguente:

```
select SupplierID, CompanyName from Suppliers limit 30
```

Quando l'utente comincia a scrivere nella parte di input dell'autocomplete, si attiva una seconda modalità di funzionamento nella quale la datamap cerca i record richiesti dall'utente eseguendo una serie di query sulla tabella *Suppliers* fino a trovare almeno un record. Scrivendo, ad esempio, la lettera E, la datamap esegue le seguenti query:

```
select SupplierID, CompanyName from Suppliers where (CompanyName = 'e')
limit 30

select SupplierID, CompanyName from Suppliers where (CompanyName ILIKE
'e%') limit 30
```

Dopo la seconda query non ne vengono eseguite altre perché vengono trovati dei dati. In caso contrario la datamap continuerebbe ad eseguire altre query utilizzando operatori più inclusivi, come ad esempio: `CompanyName ILIKE '%e%'`.

È possibile controllare da codice questo procedimento implementando l'evento *onSearch* della datamap oppure tramite l'evento *onFilter* dell'elemento *IonAutoComplete*.

Si noti infine che la query della datamap può contenere altre clausole where, sia collegate ad elementi visuali che ad altre proprietà della videata. In questo modo è possibile implementare elementi di ricerca basati su dati contenuti in altri elementi o comunque in un contesto specifico.

Uso di datamap basate su documenti e controlli di tipo autocomplete

Nel paragrafo precedente abbiamo visto l'utilizzo di una datamap basata su query come sorgente di valori per un elemento *IonAutoComplete*: se la query contiene una clausola *where* collegata all'elemento, si attiva una modalità efficiente di ricerca dei possibili valori.

Ci sono casi in cui è preferibile ottenere lo stesso comportamento usando una datamap basata su documenti, ad esempio quando si sta sviluppando un'applicazione locale basata sulla Document Orientation Remota, come illustrato nel libro relativo alla Sincronizzazione.

Per ottenere lo stesso comportamento del caso precedente usando i documenti, si inizia specificando il documento da selezionare nella proprietà *Tipo contenuto* della datamap. Siccome le datamap basate su documenti non hanno una query in cui inserire clausole *where*, al loro posto occorre scrivere la seguente riga nell'evento *onLoad* della videata:

```
$vlSupplier.addDOFilter("SupplierID", $autocomplete, "value");
```

Il metodo *addDOFilter* aggiunge un criterio di filtro alla datamap basata su documenti, specificando che il valore del filtro deve essere dinamicamente reperito dalla proprietà *value* dell'elemento *$autocomplete*. Così la datamap si comporta come nel caso precedente.

Una seconda modifica riguarda le proprietà del documento contenute nella cartella *Properties* nella datamap: siccome essa è basata su un documento, la cartella ne contiene tutte le proprietà. È importante eliminare tutte le proprietà a parte quella che fornisce il valore all'autocomplete, in questo caso *SupplierID*, e quella che deve essere usata come decodifica, in questo caso *CompanyName*. La datamap, infatti, utilizza questi riferimenti per scegliere quali dati comunicare all'autocomplete.

Anche in questo caso è possibile aggiungere alla datamap diversi filtri, sia verso altri elementi che verso proprietà della videata, chiamando più volte il metodo *addDOFilter*.

Selezione di dati provenienti da altre tabelle in layout lista

Le stesse tecniche viste finora per una videata di dettaglio possono essere utilizzate anche quando il controllo *IonAutoComplete* è contenuto in una lista. L'unica differenza è che se la datamap è basata su una query, la clausola *where* non può referenziare direttamente *$autocomplete* perché tutte le righe punterebbero sempre allo stesso elemento, cioè quello del template. Al posto di *$autocomplete.value* è possibile scrivere *this.parentElement.value*, dove *this* è la datamap e la proprietà *parentElement* si riferisce all'elemento che la contiene. In questo modo la datamap di ogni riga punta al proprio elemento.

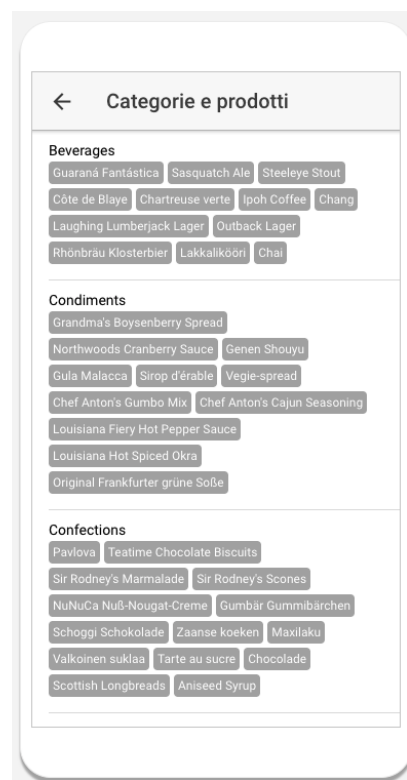
Se la datamap è basata su documenti, la modifica riguarda la riga di codice da inserire nell'evento *onLoad* che diventa:

```
$vlSupplier.addDOFilter("SupplierID", $vlSupplier.parentElement, "value");
```

Datamap innestate e ricorsive

In questo paragrafo affrontiamo un ulteriore utilizzo delle datamap per ottenere liste innestate o gestire strutture ricorsive come alberi, organigrammi ed altri elementi simili.

Come primo esempio vogliamo realizzare una lista di categorie che mostri, per ognuna di esse, anche l'elenco dei prodotti. Nell'immagine seguente viene mostrato un esempio dell'obiettivo da raggiungere.



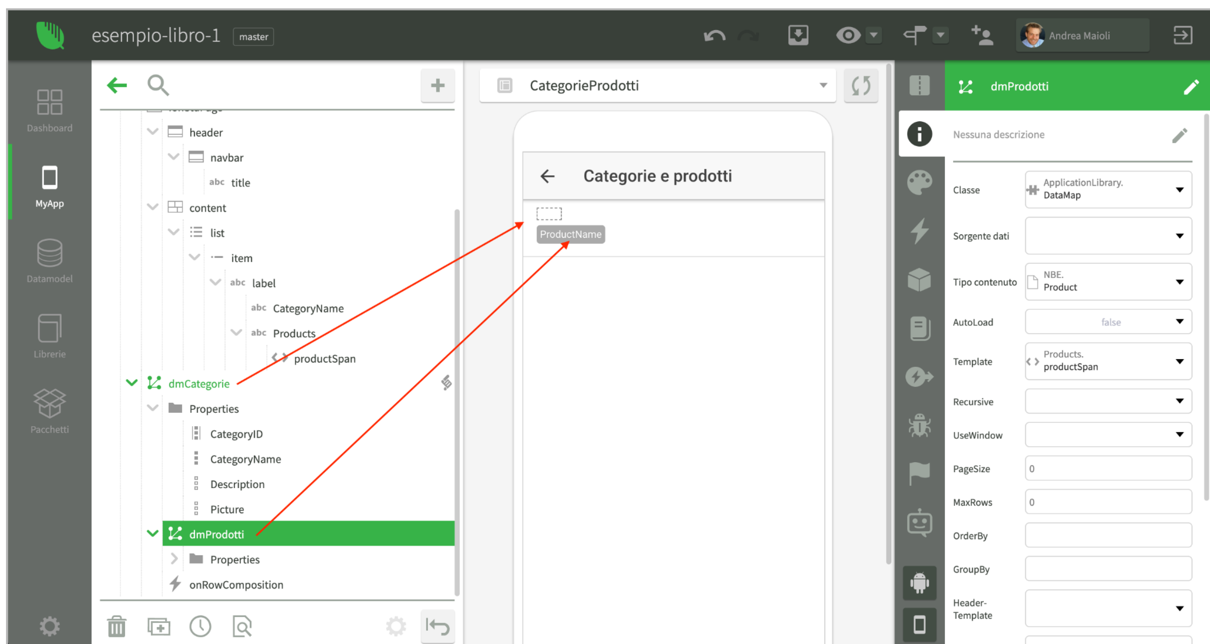
Per ottenere questa struttura, è possibile partire dalla lista delle categorie come indicato nel primo paragrafo del capitolo, poi è necessario innestare una seconda lista all'interno della prima. Questo avviene implementando tre strutture all'interno della videata.

Innanzitutto occorre aggiungere all'interno della datamap *dmCategorie* una datamap interna, che chiameremo *dmProdotti*. Siccome essa è contenuta in un'altra datamap, è chiamata datamap innestata.

In secondo luogo, per generare la lista dei prodotti di una categoria, *dmProdotti* ha bisogno di un suo template che deve essere all'interno del template della datamap *dmCategorie*.

Infine, l'evento *onRowComposition* della datamap esterna deve effettuare il caricamento dei dati della datamap interna. In particolare viene usato il seguente codice:

```
$dmCategorie.onRowComposition = function (row, template)
{
  let c = yield App.NBE.Product.loadCollection(app, {CategoryID :
                                  row.CategoryID});
  $dmProdotti.collection = c;
};
```



All'apertura della videata, la datamap *dmCategorie*, avendo il flag *AutoLoad* attivo, esegue la query e carica la lista delle categorie. Per ogni categoria, crea un clone del template visuale e associa i dati della categoria ad esso.

In questo caso però, è presente anche *dmProdotti*, una datamap innestata all'interno di *dmCategorie*. Quindi, subito dopo aver creato il clone del template per una riga di *dmCategorie*, viene clonata anche la datamap *dmProdotti*, questo clone viene associato alla riga generata per la categoria ed infine, come template di riga della datamap clone, viene

utilizzato l’elemento corrispondente al template di *dmProdotti* all’interno del clone del template di riga della datamap esterna.

A questo punto viene notificato l’evento *onRowComposition* per la datamap esterna, che si occupa di caricare la collection dei prodotti della categoria per cui l’evento è stato notificato e di usarla come sorgente dati per il clone della datamap interna. Questa operazione viene effettuata dall’istruzione: `$dmProdotti.collection = c` dove l’espressione *$dmProdotti* usata all’interno del codice dell’evento *onRowComposition* indica proprio il clone della datamap generato per la riga in fase di composizione.

A sua volta, la datamap *dmProdotti* clonata genera i propri elementi visuali in base ai dati ricevuti, clonando il suo template di riga, impostando le proprietà visuali e così via.

Per quanto riguarda l’accesso alle proprietà delle datamap da parte degli elementi visuali, occorre tenere presente della struttura visuale degli elementi stessi. Se ad esempio al clic sul nome di un prodotto volessimo visualizzare il messaggio “questo è il prodotto <nome> della categoria <nome>”, potremmo referenziare il nome del prodotto con l’espressione *this.row.ProductName*, come nei casi precedenti; mentre, per arrivare alla corrispondente *DataRow* della datamap esterna, possiamo risalire nella catena degli elementi visuali passando dal template della datamap interna a quello della datamap esterna. In sintesi l’espressione da utilizzare è la seguente:

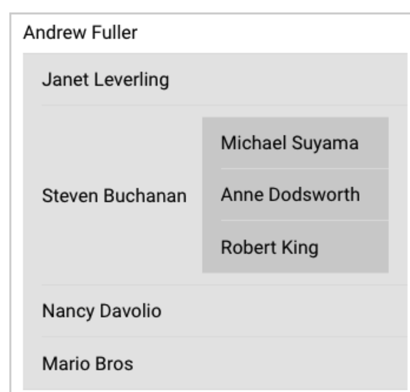
```
let nomeCategoria = this.row.element.parent.row.CategoryName;
```

Infatti *this.row.element* è il clone del template generato per il prodotto cliccato. Il *parent* di tale elemento appartiene invece al clone del template generato per la categoria a cui il prodotto cliccato appartiene. A tale elemento è associata la proprietà *row* che in questo caso contiene i dati della categoria. In questo modo abbiamo raggiunto il nostro obiettivo.

Datamap ricorsive

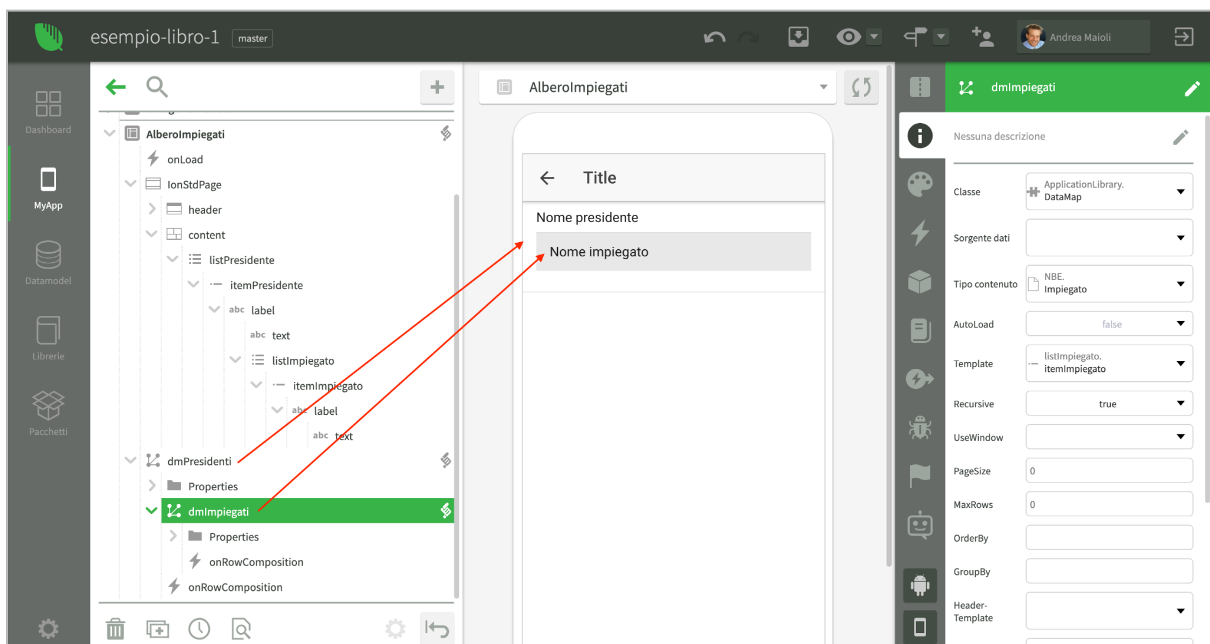
L’uso delle datamap innestate permette di costruire liste multi-livello nelle quali, tuttavia, il numero di livelli deve essere conosciuto a design time. Se si devono gestire strutture ricorsive, come ad esempio le strutture ad albero, una datamap innestata non è sufficiente.

Immaginiamo di voler sfruttare il legame ricorsivo dei record della tabella impiegati per visualizzare un organigramma come il seguente:



Per ottenere il risultato possiamo creare la struttura mostrata nell’immagine seguente, in cui:

1) È presente una datamap di primo livello che si occupa di caricare i dati di primo livello. In questo caso i presidenti, quelli che non dipendono da nessuno.
2) All’interno della datamap di primo livello è presente una datamap innestata che viene caricata nell’evento *onRowComposition* della precedente. Essa contiene gli impiegati che dipendono direttamente dai presidenti.
3) Questa seconda datamap ha il flag *Recursive* attivo, quindi quando essa genera una riga, all’interno di questa riga viene nuovamente clonata anche la datamap stessa, che in questo modo diventa di terzo livello.
4) La datamap di secondo livello carica gli impiegati che rispondono all’impiegato attuale e poi associa questa collection al clone di sé stessa che trova come proprietà della riga.



È interessante vedere il codice dell’evento *onRowComposition* della datamap di secondo livello:

```
$dmImpiegati.onRowComposition = function (row, template)
{
  row.dmImpiegati.collection = yield App.NBE.Impiegato.loadCollection(app,
                {ReportsTo : row.EmployeeID});
};
```

Il riferimento alla datamap di secondo livello clonata per diventare di terzo livello viene ottenuto dall’espressione *row.dmImpiegati*. La datamap di secondo livello diventa quindi il sistema per innescare la ricorsione all’n-esimo livello in quanto il suo codice, partendo dai dati del livello attuale, si occupa di caricare quelli del livello successivo.

A livello visuale, l’innesco della ricorsione avviene in maniera analoga. Quando la datamap di secondo livello viene clonata per generare il terzo, viene clonato anche il *listbox*,

l'elemento *parent* del template di riga, destinato a contenere tutti gli elementi delle righe della datamap. Il clone del *listbox* viene aggiunto agli elementi della riga della datamap di secondo livello e verrà usato dalla datamap di terzo livello per contenere le sue righe.

In questo modo è possibile costruire strutture ricorsive a *n* livelli sia come struttura di datamap innestate in memoria che come struttura di elementi visuali generati da queste datamap.