IonicUI

Indice generale

IonicUI

Ottieni il massimo dal template per applicazioni omnichannel di Instant Developer Cloud

Introduzione

Nell'ambito delle applicazioni di trasformazione digitale, una delle necessità più frequenti è quella di sviluppare applicazioni omnichannel, cioè per smartphone, tablet e desktop. Per ottenere questo risultato serve molto di più dei semplici framework responsive, che si limitano ad utilizzare qualche tecnica CSS per adattare la larghezza del contenuto al video.

Innanzitutto occorre un framework che generi un aspetto grafico aderente alle linee guida del tipo di device: per Apple e per Android esistono riferimenti specifici differenti fra loro.

Poi si deve tenere conto delle modalità di input specifiche dei dispositivi touch, come ad esempio, la gestione della tastiera a schermo, l'edge swipe per tornare alla pagina precedente, le dimensioni per i controlli adatte alle dita, il feedback aptico, eccetera.

Il tutto senza dimenticarsi dell'utilizzo desktop tramite mouse e tastiera, che richiede ancora una volta un adattamento del contenuto al tipo di esperienza utente atteso per le applicazioni desktop.

Per poter soddisfare queste necessità Instant Developer Cloud contiene un set di elementi grafici denominato *IonicUI* che permettono di realizzare applicazioni omnichannel in grado di funzionare correttamente su desktop, smartphone e tablet, sia di tipo Apple che Android.

IonicUI è un fork del noto framework Ionic per lo sviluppo di applicazioni mobile; la decisione di effettuare il fork è nata sia dalla necessità di integrazione con Instant Developer Cloud che dalla necessità di proteggere gli utenti di Instant Developer da breaking change introdotte dalla versione originale.

Elementi disponibili

Gli elementi visuali disponibili in IonicUI sono i seguenti:

- *Struttura dell'applicazione*: IonSplitPane, IonNavController, IonMenu.
- *Struttura della pagina*: IonPage, IonHeader, IonFooter, IonContent, IonToolbar, IonSearchBar, IonButtons, IonTabs, IonTab.
- *Griglie e Card*: IonGrid, IonRow, IonCol, IonCard, IonCardSection.
- *Liste*: IonList, IonItem.
- *Testi, Label e Immagini*: IonTitle, IonLabel, IonNote, IonBadge, IonText, IonIcon, IonAvatar, IonThumbnail.
- *Controlli*: IonButton, IonCheckbox, IonToggle, IonRadio, IonSegment, IonSegmentButton, IonInput, IonRange, IonDateTime, IonSelect, IonAutocomplete, IonSwipe, IonSpinner.

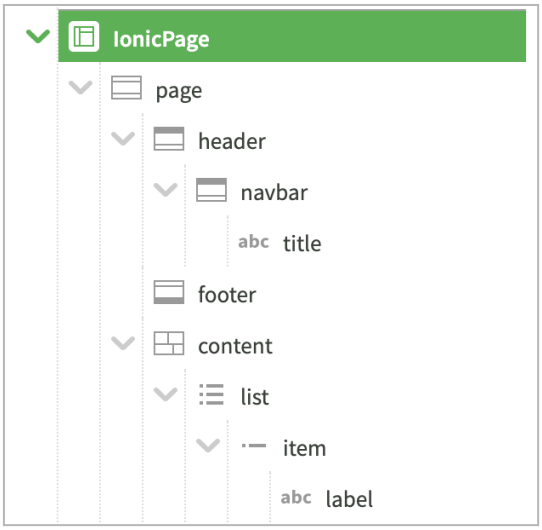
Oltre a questi elementi è possibile utilizzare ogni altro elemento visuale disponibile, sia di tipo base che componenti finiti come mappe e grafici, e anche elementi personalizzati. Non è invece possibile aggiungere nella stessa applicazione anche altri template grafici come ad esempio Bootstrap.

IonicUI contiene per default il set di icone IonIcons, sia in versione 4 che 5, date le caratteristiche diverse fra loro, e consente di aggiungere icon set personalizzati come verrà spiegato nei paragrafi successivi.

Per rendere disponibile IonicUI in un proprio progetto è necessario importare il package IonicUI tramite la videata apposita dell'IDE.

Le pagine IonicUI

Il template IonicUI permette di suddividere l'interfaccia utente dell'applicazione in pagine diverse, in cui ogni pagina corrisponde ad una videata. La struttura base di una pagina è costituita dagli elementi mostrati nell'immagine seguente.



Il primo elemento è di tipo *IonPage* e rappresenta il contenitore dell'intera pagina. All'interno troviamo i tre elementi principali, header (*IonHeader*), footer (*IonFooter*) e content (*IonContent*). Header e footer rappresentano le intestazioni fisse, mentre content è un contenuto scorrevole.

Di solito l'intestazione e il contenuto sono sempre presenti, mentre il piede può mancare. Se la videata è contenuta in una tabbed view, il piede non è mai presente.

Nei dispositivi con tastiera a video, se nella videata è presente un campo di input, quando la tastiera appare si sovrappone al piede della videata. Per default, la videata viene ristretta ed il piede si sposta apparendo subito sopra la tastiera. In alcuni casi, tuttavia, si preferisce lasciare che il piede rimanga nascosto dalla tastiera, ad esempio se è troppo ingombrante o se è meglio che l'utente chiuda la tastiera prima di accedere al contenuto del piede. Per

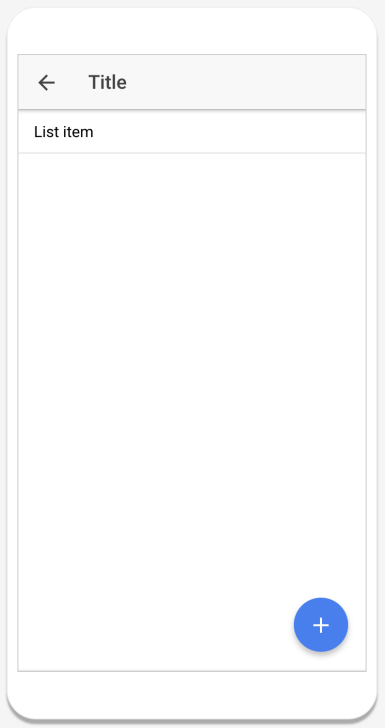
mantenere il piede fisso sotto la tastiera è necessario impostare *position:fixed* come stile in linea dell'oggetto IonFooter.

All'interno dell'elemento header troviamo sempre un elemento navbar (*IonNavBar*) che gestisce il pulsante di ritorno alla videata precedente o del menu; all'interno della navbar troviamo sempre un elemento title (*IonTitle*) che rappresenta il titolo della videata. Usando questi specifici elementi otterremo un'applicazione in grado di adattarsi alle linee guida dei vari tipi di dispositivi supportati.

Il contenuto del footer può variare. È possibile inserire direttamente nel footer un elemento *IonButton* per ottenere un floating action button in stile material design, altrimenti è preferibile inserire una toolbar (*IonToolbar*) in cui verrà composto il contenuto del piede.

Per quanto riguarda il content, è possibile includere elementi di qualunque tipo. Tuttavia il template IonicUI suggerisce di presentare il contenuto usando una struttura costituita da una *IonList* che a sua volta contiene elementi di tipo *IonItem*.

Nota importante: per ottenere velocemente la struttura di default della pagine è consigliabile inserire direttamente all'interno della videata vuota un template di elementi chiamato *IonStdPage* che è possibile trovare sempre nella barra degli elementi visuali dell'IDE. A questo punto sarà possibile eliminare le parti non necessarie o modificare le proprietà di quelle esistenti. Nell'immagine seguente è mostrata una videata in cui è stato inserito il template *IonStdPage*.

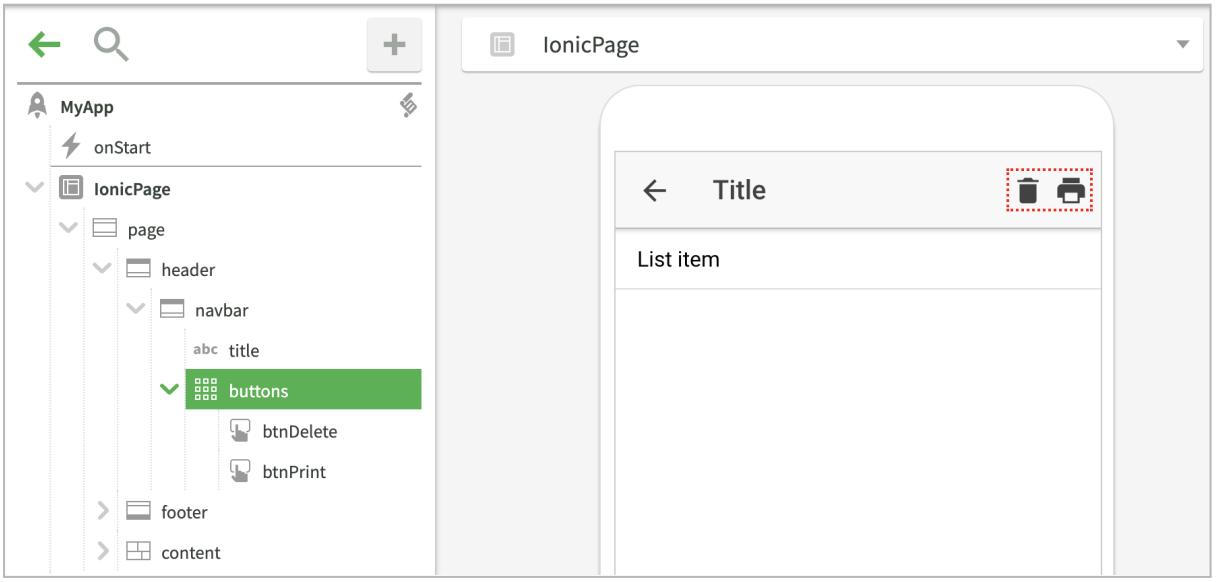


Il template IonStdPage

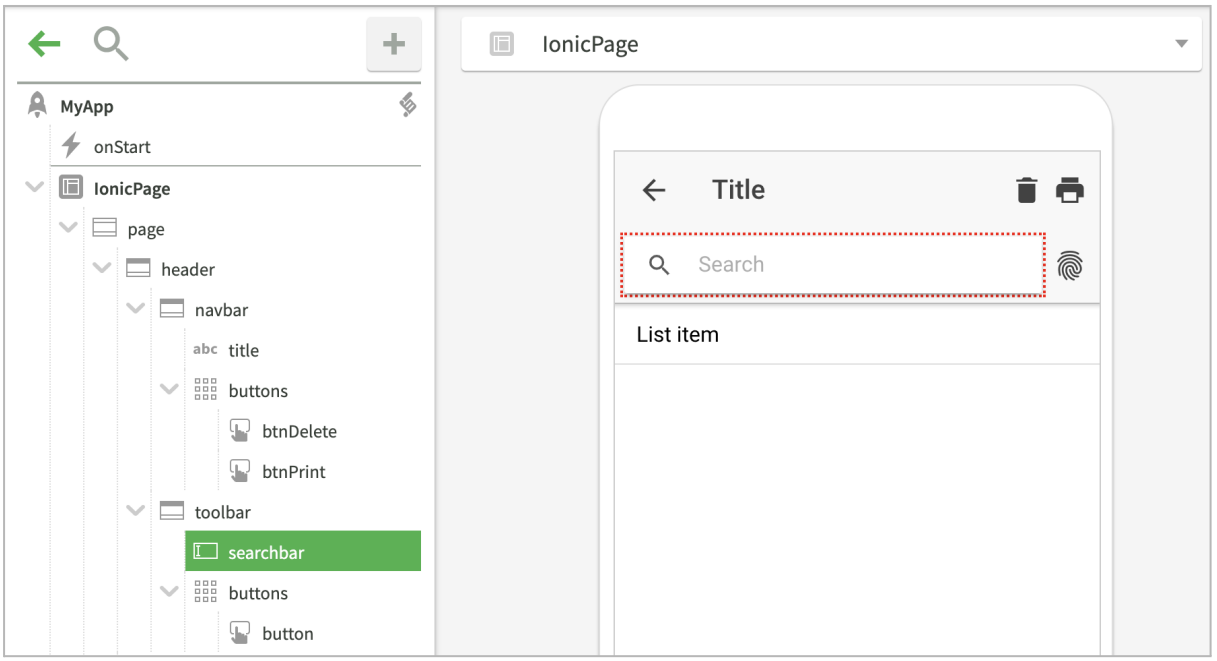
Toolbar e pulsanti

L'intestazione e il piede di pagina sono spesso utilizzate per contenere pulsanti di azione o ulteriori controlli come barre di ricerca o contenuti testuali di riepilogo.

Per inserire pulsanti nelle intestazioni o nei piè di pagina, a parte il caso "floating action button" visto in precedenza, il metodo migliore è quello di utilizzare un elemento contenitore di tipo *IonButtons*. Nell'immagine seguente possiamo vedere un esempio di questa struttura.



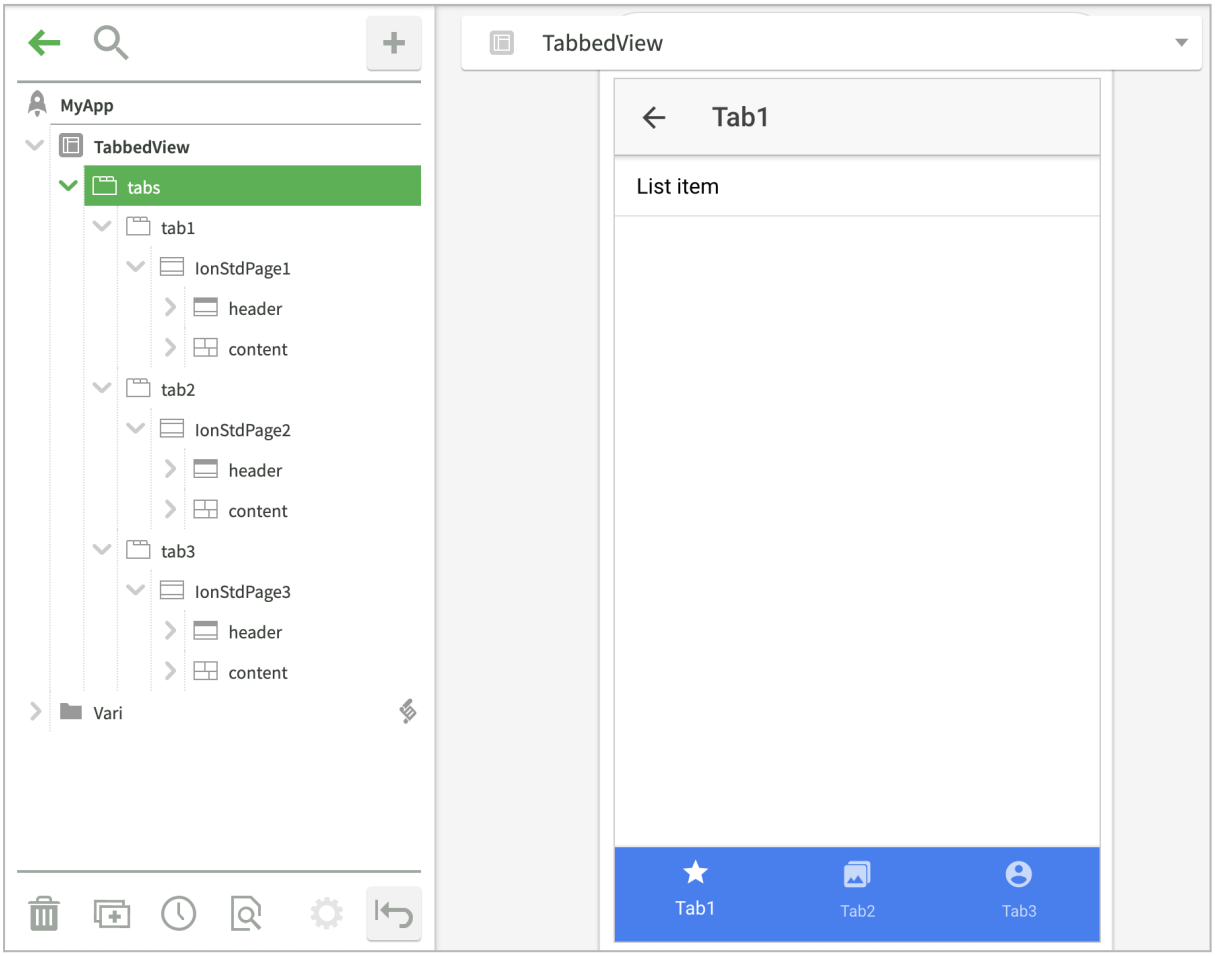
Per inserire ulteriori controlli nell'intestazione o nel piede, è possibile utilizzare elementi di tipo *IonToolbar*, che, a loro volta, potranno contenere barre di ricerca (*IonSearchBar*), contenitori di pulsanti (*IonButtons*), label (*IonLabel*), o altri elementi visuali. L'immagine seguente mostra un esempio di barra di ricerca nell'intestazione.



Le tabbed view

Una struttura di pagina molto utilizzata è la visualizzazione a schede, in cui nella pagina viene visualizzata una barra di selezione di contenuti alternativi. Questo è un buon pattern se si hanno al massimo cinque contenuti alternativi ed essi non sono correlati fra loro.

La struttura base di una tabbed view è mostrata nell'immagine seguente:



L'elemento base della pagina è di tipo *IonTabs* e a sua volta contiene un elemento *IonTab* per ogni contenuto alternativo. Nell'immagine ci sono tre *IonTab*, ognuno per i vari contenuti alternativi che si desiderano mostrare.

All'interno di ogni *IonTab* deve essere contenuta una pagina completa. Nell'esempio è stata inserita una *IonStdPage* in ogni tab eliminando il footer da ognuna di esse perché non interferisca con la barra sottostante.

Per rendere le varie tab ancora più indipendenti, al posto di creare la struttura visuale all'interno della medesima videata, è possibile creare videate separate per ogni contenuto, renderle utilizzabili come componenti di design time e poi inserire direttamente le videate come unico contenuto di ogni tab. Per maggiori informazioni sull'utilizzo di videate a design time, si legga il paragrafo Videate come elementi visuali successivo.

Si noti infine che l'elemento *IonTabs* ha la possibilità di essere posizionato nella parte alta dello schermo oppure in basso. Se non si specifica alcuna opzione, esso apparirà in basso su dispositivi iOS oppure in alto per Android o nel browser. Per una maggiore uniformità della user experience, si consiglia di impostare la proprietà *Placement* dell'oggetto *IonTabs* a *bottom*.

Definire il contenuto delle pagine

Dopo aver illustrato la struttura di una pagina, vediamo ora come definire la parte più importante: il contenuto, cioè gli elementi visuali posizionati all'interno dell'elemento *IonContent*.

Il contenuto può essere disposto secondo due tipi principali di layout: la lista e la griglia.

Il layout Lista

Nel template IonicUI l'elemento primario per la definizione del contenuto è la lista di elementi.

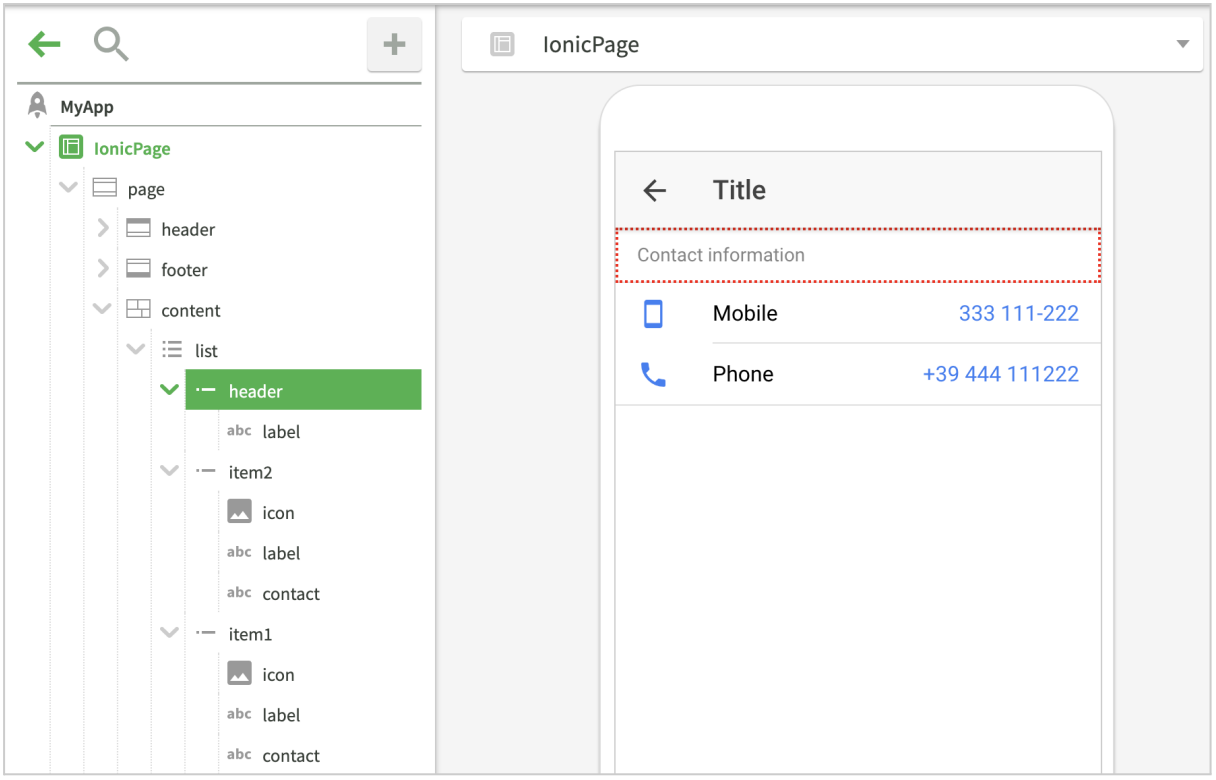
Una lista è costituita da un elemento di tipo *IonList* che funge da contenitore di righe, cioè elementi di tipo *IonItem*. All'interno di ogni riga troviamo uno o più elementi che definiscono il contenuto, come ad esempio immagini, label, elementi testuali e controlli per la modifica dei dati. Una lista può essere preceduta da elementi di testata, come ad esempio un'immagine a tutta larghezza, e seguita da altri elementi o anche da altre liste.

Dal punto di vista dei dati dell'applicazione, una lista può essere usata per rappresentare una collection di documenti, ed in tal caso ogni riga corrisponde ad un documento diverso, oppure per mostrare le informazioni di dettaglio relative ad uno specifico documento e in tal caso avremo una form di dettaglio, da usare anche per inserire o modificare i dati.

L'elemento *IonList* possiede diverse proprietà che ne controllano l'aspetto grafico e il funzionamento. In particolare si segnala la proprietà *refresher* che, se impostata a *true*, attiva il meccanismo *pull to refresh*. In questo caso, quando l'utente trascina la lista verso il basso, l'elemento notifica l'evento *onRefresh* all'applicazione. Nel codice che gestisce l'evento, oltre ad effettuare l'aggiornamento dei dati visualizzati, occorre chiamare il metodo *refreshCompleted* sull'elemento lista in modo da resettare la funzionalità.

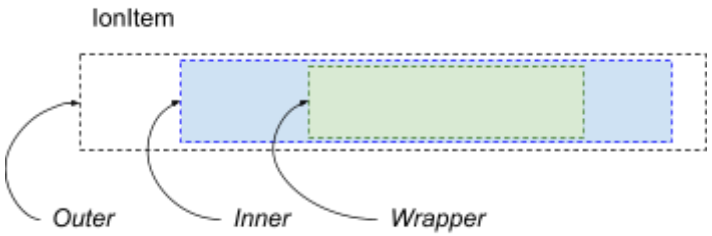
Gli elementi *IonItem* possono essere di tipo diverso per rappresentare le intestazioni di lista (*type=header*), le suddivisioni fra parti di lista (*type=divider*), e righe cliccabili (*type=button*). Si segnala la proprietà *wrapText* che, se attivata, permette ai testi dell'item di andare a capo, altrimenti per default saranno limitati ad una sola riga ognuno.

Un esempio di lista è mostrato nell'immagine seguente:



Costruzione di una riga della lista

L'elemento *IonItem* è costituito da tre parti principali, come si vede nello schema seguente:



Nella parte *Outer* vengono inclusi elementi di tipo *IonIcon, IonAvatar o IonThumbnail* posizionati all'inizio del contenuto della riga. Nella parte *Inner* compaiono elementi testuali come *IonLabel* e *IonNote*, mentre nella parte *Wrapper* troviamo i controlli di modifica dei dati come *IonInput*, *IonSelect* e così via.

Ogni elemento contenuto all'interno di un *IonItem* possiede una proprietà *ItemSide* che permette di personalizzare l'allocazione dell'elemento all'interno della riga. *ItemSide* è una proprietà di tipo stringa in cui è possibile inserire il lato in cui l'elemento deve apparire (*left* o *right*) e la zona (*outer*, *inner* o *wrapper*). Se ad esempio si desidera far apparire un elemento dal lato estremo destro, potremmo scrivere: $*elemento.itemSide = "right, outer"*.

itemSide può essere impostato anche a design time e, in questo caso, occorre cliccare sul pulsante di aggiornamento manuale dell'anteprima per vedere l'effetto.

Principali elementi per la costruzione della lista

Il framework IonicUI prevede i seguenti elementi per la costruzione di una riga di una lista:

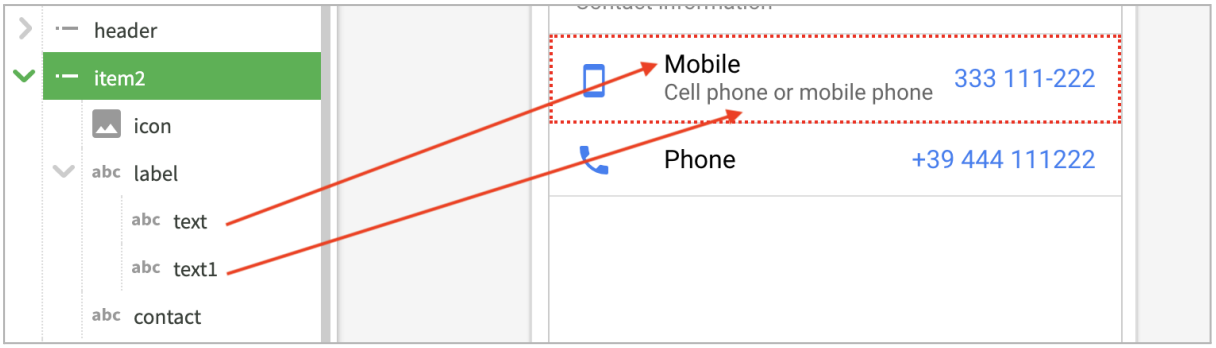
- *IonButtons*: contenitore di *IonButton* per inserire pulsanti sul lato destro di una riga della lista.
- *IonLabel*: elemento primario per inserire contenuti testuali in una riga di una lista. Può contenere elementi *IonText* nel caso di presenza di diversi elementi testuali separati.
- *IonNote*: elemento testuale posto sul lato destro della riga. Per default utilizza il colore di tema *light*. Si consiglia di impostare un colore più adeguato tramite la barra degli stili.
- *IonBadge*: elemento grafico posto sul lato sinistro o destro della riga.
- *IonIcon*: aggiunge un'icona sul lato sinistro o destro della riga.
- *IonAvatar*: aggiunge un'immagine inclusa in un cerchio sul lato sinistro o destro della riga.
- *IonThumbnail*: rappresenta un'immagine quadrata di dimensioni maggiori disposta nel lato sinistro o destro della riga.
- *IonCheckbox, IonToggle, IonRadio, IonInput, IonRange, IonDateTime, IonSelect, IonAutocomplete:* elementi per la modifica dei dati.
- *IonSwipe*: aggiunge alla riga un menu di tipo swipe.

Inserendo questi elementi in una riga essi si adattano naturalmente per comporne il contenuto. Tuttavia una riga può contenere qualunque elemento o gestire strutture visuali complesse tramite i contenitori a layout orizzontale o verticale. In questi casi potrebbe essere necessario modificare lo stile degli elementi inseriti in modo da ottenere i risultati desiderati. Si segnala inoltre che alcuni elementi potrebbero presentare comportamenti particolari o anomali quando inseriti nelle liste, in quanto l'elemento *IonItem* crea i propri elementi figli prima di essere inserito nel DOM, mentre alcuni elementi potrebbero richiedere la presenza nel DOM al momento della creazione. In questi casi l'elemento dovrà essere aggiunto ad *IonItem* solo dopo che la videata è già stata aperta.

Gli elementi *IonAvatar* e *IonThumbnail* sono pensati per mostrare immagini quadrate. Se si desidera usarli per immagini con qualunque fattore di forma, si consiglia di definire una classe CSS come la seguente e poi applicarla all'elemento avatar o thumbnail.

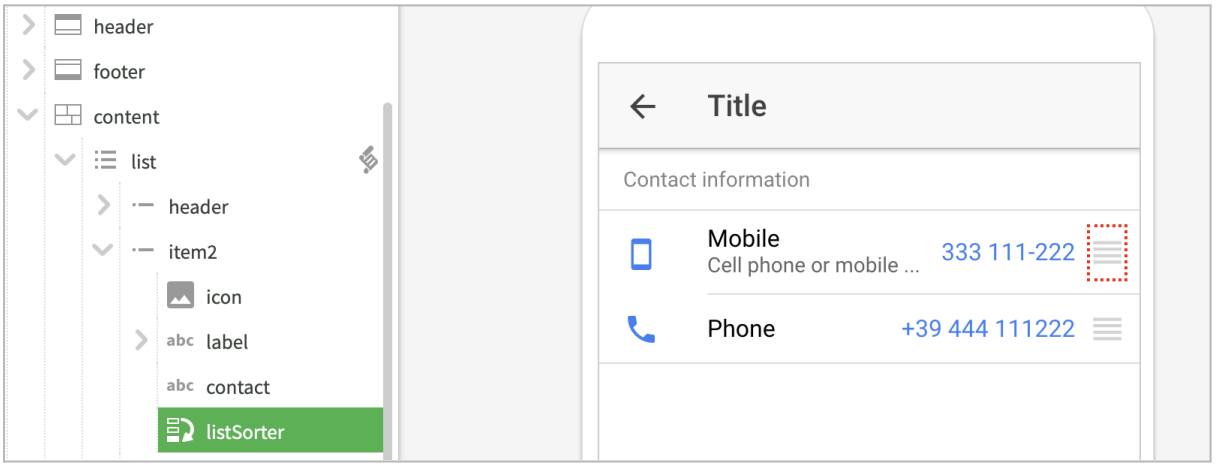
```
.anyimage img {
  height: 4rem;
  object-fit: cover;
}
```

Se, infine, si desidera inserire più elementi testuali separati, si consiglia di usare un elemento *IonLabel* come contenitore di elementi *IonText*; in base alla proprietà *type* di questi ultimi potremo ottenere label vere e proprie (*type=span*) o testi secondari (*type=p*), come mostrato nell'immagine seguente. Si ricorda che in questo caso l'elemento *label* non deve contenere testo, cioè la proprietà *innerText* deve essere vuota.



Riordinamento delle righe

Alcune liste devono includere un controllo per il riordinamento delle righe. Per ottenere questo risultato si può aggiungere un elemento di tipo *ListSorter* come ultimo della riga. L'effetto è il seguente:



Trascinando l'elemento evidenziato, si potrà riordinare le righe della lista. Al termine del riordinamento, l'elemento *ListSorter* notifica l'evento *onReorder* all'applicazione che può così applicare il nuovo ordine ai dati sottostanti.

Normalmente questa funzionalità è usata quando nella lista viene mostrata una collection di documenti. Per maggiori informazioni relative alla gestione del riordinamento delle righe di una collection, si consiglia di aprire il progetto di esempio Mobile Design Patterns.

Costruzione di un menu swipe

Per costruire un menu swipe di riga è sufficiente aggiungere in fondo alla riga un elemento di tipo *IonSwipe*.

Per definire la lista dei comandi del menu occorre impostare la proprietà *commands* di *IonSwipe*. A design time, tale proprietà deve essere impostata ad una delle liste valori del progetto e ognuna delle sue costanti diventa un comando del menu. In questo caso, la proprietà immagine della costante può essere impostata al nome di un'icona del set *IonIcons*, ad esempio *trash*, e la proprietà stile può contenere il nome di un colore del tema che rappresenta il colore del comando, ad esempio *danger*, e la parola *default* per indicare

che il comando relativo alla costante è quello di default, cioè quello che viene inviato continuando a trascinare la riga fino in fondo.

Quando l'utente usa il menu swipe, l'elemento notifica l'evento *onSwipeSelected* passando come parametro il valore della costante relativo al comando utilizzato.

È possibile definire la lista dei comandi anche a runtime, impostando la proprietà *commands* ad un array di oggetti che rappresenta l'equivalente della lista valori. Ad esempio:

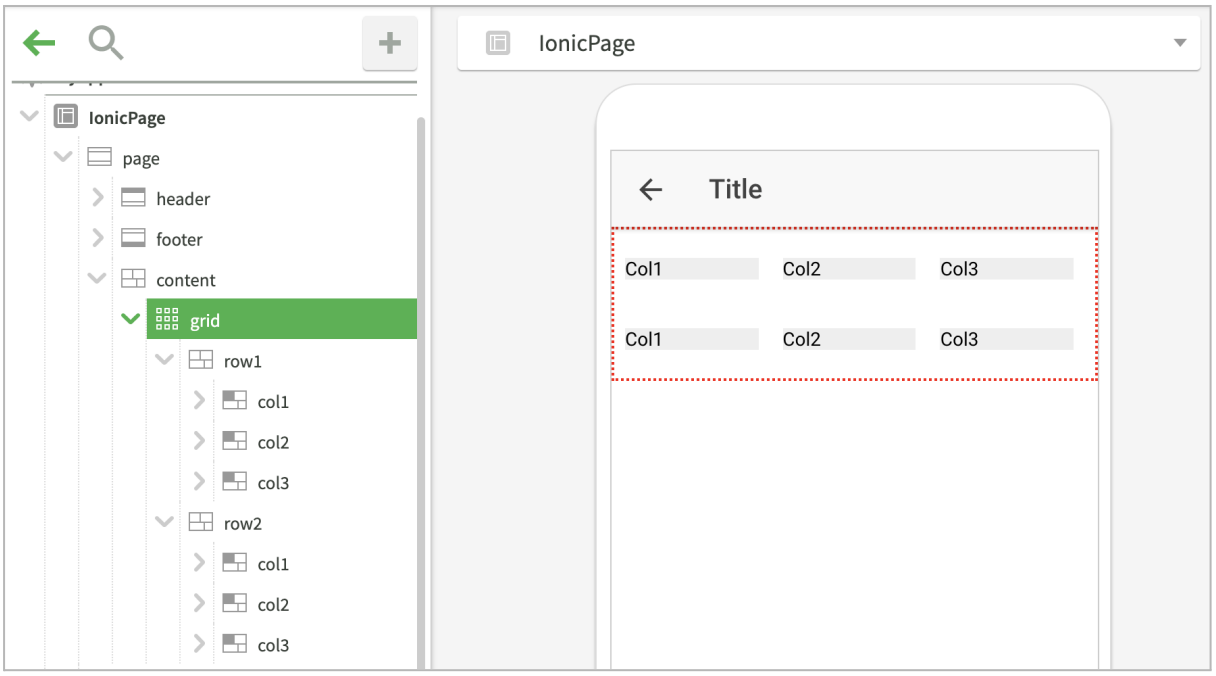
```
$swipe.commands = [{n:"Cancella", v:"DEL", icon:"trash",
                    style:"danger, default"}, { … } ];
```

Come sempre, la comunicazione di una lista valori agli elementi visuali può essere eseguita a runtime tramite l'array delle costanti, in cui ogni costante ha le seguenti proprietà:

- *n (name, label)*: nome della costante.
- *v (code, cmd)*: valore della costante.
- *src (img, icon)*: immagine o icona.
- *s (style, class, cls, className)*: stile visuale.

Il layout Griglia

Il layout griglia permette di gestire una molteplicità di contenuti in un formato righe e colonne con caratteristiche responsive. Per definire una griglia occorre utilizzare gli elementi *IonGrid*, *IonRow* e *IonCol* come mostrato dall'immagine seguente.



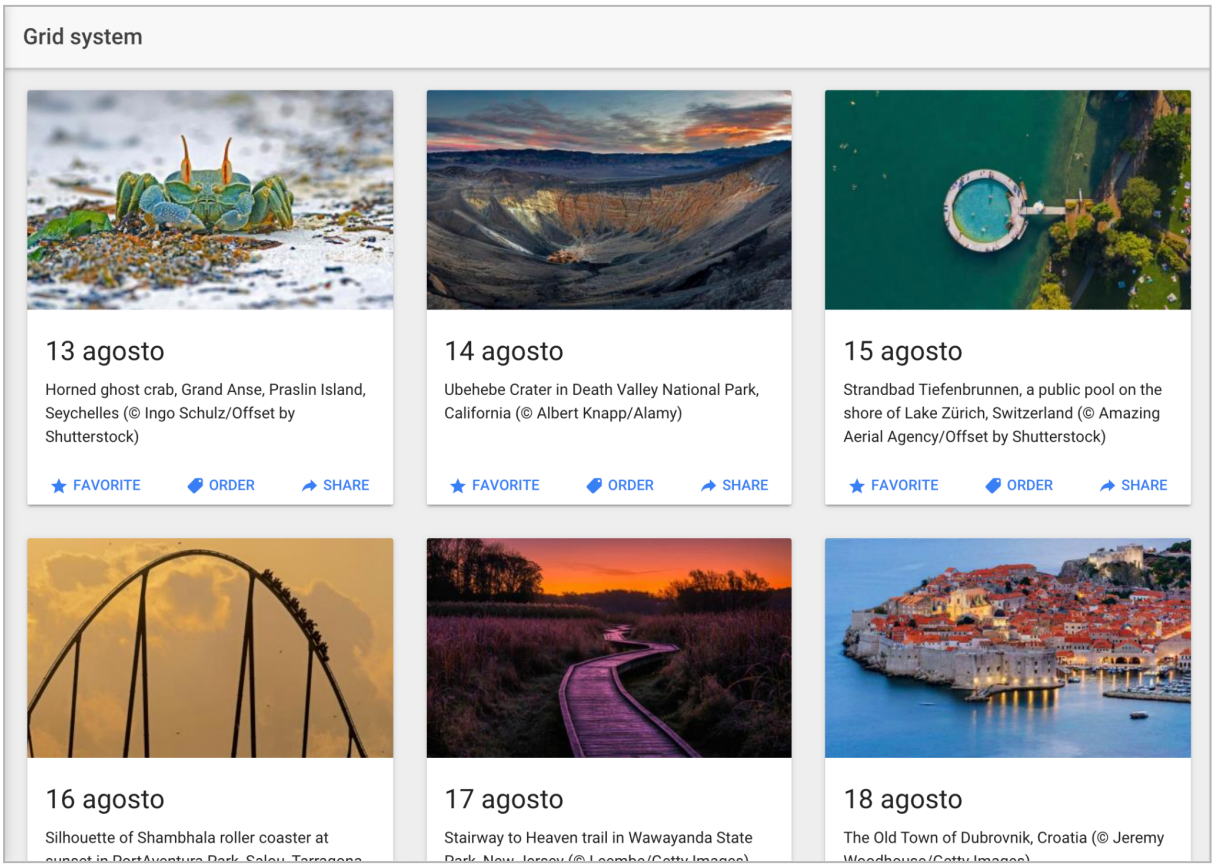
Se non si specifica la larghezza delle colonne, esse saranno tutte sulla stessa riga.

Impostando invece le proprietà *xs, sm, md, lg* e *xl* degli *IonCol* sarà possibile definire la larghezza relativa di ogni colonna, tenendo conto di un layout suddiviso su 12 colonne. La

proprietà *xs* è relativa ad uno smartphone verticale, *sm* a quello orizzontale, *md* è un tablet verticale, mentre *lg* è un tablet orizzontale. *xl* è un browser in un desktop.

Se, ad esempio, si imposta per un *IonCol* la proprietà *xs* a 12 e *sm* a 6, tale colonna sarà larga quanto l'intero schermo su uno smartphone verticale, mentre solo metà in caso orizzontale.

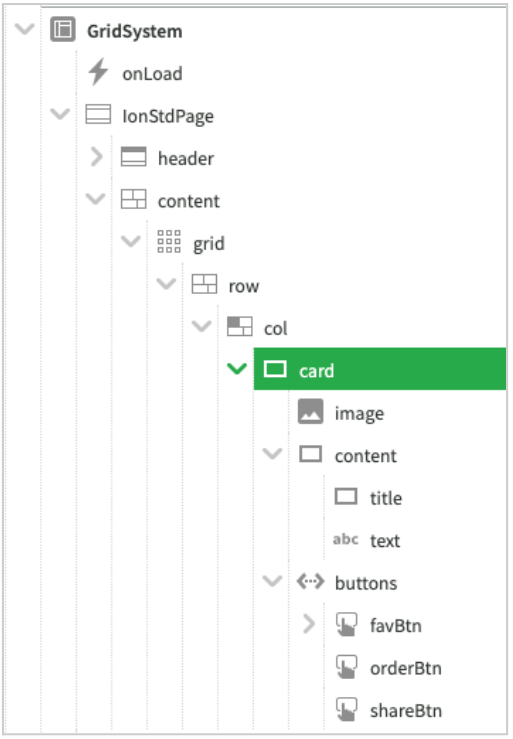
Un oggetto *IonCol* può contenere da una semplice label ad una intera *IonList*. Un caso particolare frequentemente utilizzato con il layout griglia è l'oggetto *IonCard*, con cui si può realizzare una galleria di immagini responsive come la seguente:



L'oggetto *IonCard* può essere inserito all'interno di un oggetto *IonCol* che, a sua volta, funge da template di riga per la datamap che contiene i dati delle foto.

La struttura della card comprende un elemento *Image*, gli elementi *IonCardSection* per rappresentare il titolo e il contenuto e un container di pulsanti.

L'immagine seguente rappresenta la struttura della card. Per maggiori informazioni si consiglia di vedere il progetto di esempio Mobile Design Patterns.



Struttura di una card per una galleria di foto

I font e le icone

Spesso si desidera includere nuovi font e set di icone nel proprio progetto; per ottenere questo scopo è possibile aggiungere una risorsa di tipo font all'applicazione o alla libreria.

Abbiamo già visto nel capitolo 01-Struttura di un'applicazione come utilizzare una risorsa di tipo font. Vediamo ora come utilizzare la stessa tecnica per caricare un font di icone che possono essere utilizzate in aggiunta al set IonIcons.

Per caricare un font di icone, ad esempio IcoFont, occorre seguire questi passaggi:

1) Creare una libreria di tipo "Personalizzata".
2) Creare una classe nella libreria. Se la classe CSS da applicare alle icone è preceduta da un prefisso, il nome della classe appena creata deve essere uguale al prefisso. Ad esempio, se la classe CSS è *fa fa-book*, il nome della classe deve essere *fa*.
3) Creare una nuova risorsa di tipo font nella classe appena creata. Assegnare il nome del font alla risorsa, ad esempio *IcoFont*. Impostare la proprietà *Tipo di contenuto* a *Icona.*
4) Inserire l'URL del CSS che importa il font, ad esempio: https://icofont.com/icofont/icofont.min.css
5) Dal menu personalizzato dell'oggetto risorsa (quello con la ruota dentata sotto l'albero degli oggetti), selezionare il comando *Crea copia locale*. In questo modo i file relativi al font verranno inclusi nelle risorse del progetto e non referenziati da un sito esterno, e potranno essere usati anche nelle applicazioni mobile offline.
6) Se il font non appare quando l'applicazione viene aperta in anteprima, significa che il nome della classe non è uguale al prefisso del nome delle icone. Per correggerlo, si

può usare il picker delle icone per assegnare un'icona ad un elemento e poi usare il prefisso del valore restituito come nome della classe. Nel caso di *IcoFont*, ad esempio, il nome della classe sarà *Woff2*.

A questo punto sarà possibile selezionare le icone presenti nel font in tutti i punti in cui è possibile impostare la proprietà icona di un elemento e anche tramite codice. Il nome dell'icona deve essere preceduto dal nome della classe, ad esempio:

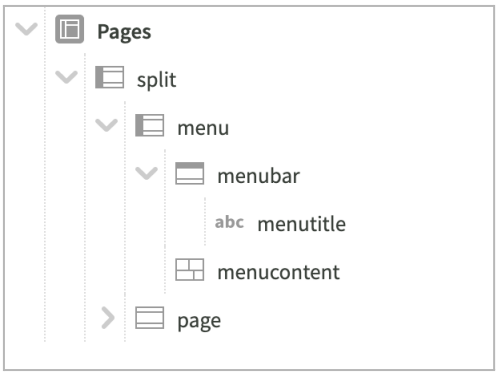
```
$fab.icon = "woff2 icofont-dart";
```

Il page controller

Oltre agli elementi visuali per la costruzione delle pagine, il framework IonicUI mette a disposizione un componente fondamentale per creare applicazioni omnichannel di successo: il *page controller*.

Lo scopo di questo componente è quello di organizzare la navigazione tra le varie pagine dell'applicazione in maniera coerente con il tipo di dispositivo utilizzato. A differenza degli elementi visuali, il cui codice sorgente è puro JavaScript, il page controller è nella videata *MainPage*, creata con Instant Developer Cloud e contenuta nella libreria personalizzata *Ionic* che viene importata con il package IonicUI.

Per utilizzare il page controller nella propria applicazione è sufficiente creare una videata, ad esempio di nome *Pages*, e cambiare l'estensione da *Application.View* ad *Ionic.MainPage*. Subito dopo aver compiuto questa operazione, all'interno della videata che prima era vuota appariranno le strutture personalizzabili del page controller, come mostrato nell'immagine seguente.



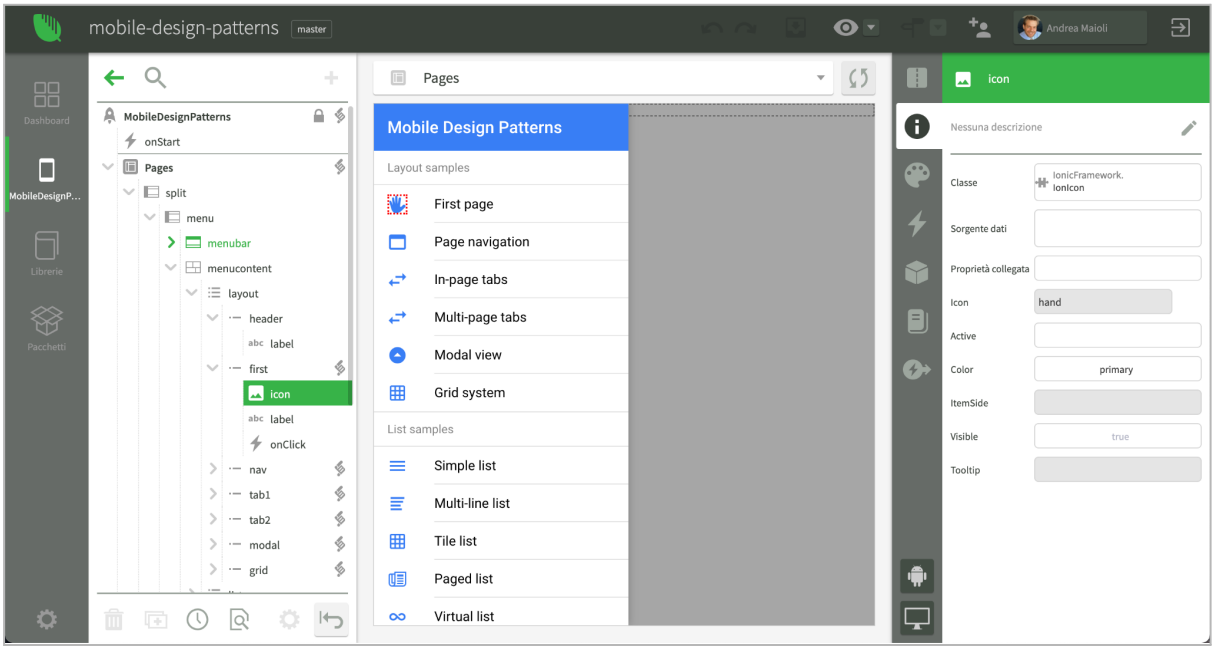
Il page controller è quindi costituito da una sezione visuale in cui inserire il menu dell'applicazione, da una seconda sezione visuale utilizzata per contenere le pagine che man mano appariranno, ed infine da una serie di metodi ereditati dalla classe base *MainPage*.

In questo paragrafo verranno illustrati i metodi principali; se si desiderano maggiori informazioni sul funzionamento interno del page controller è possibile leggere il codice sorgente contenuto nella videata *MainPage* della libreria *Ionic*.

Definizione e controllo del menu principale

La definizione del menu principale avviene direttamente nella videata *Pages* della propria applicazione. L'intestazione del menu può essere gestita tramite gli elementi *menubar* e *menutitle*, mentre gli item del menu andranno inseriti all'interno di *menucontent*.

Per creare gli elementi visuali delle righe di menu si consiglia di creare una lista tramite *IonList*, *IonItem* e *IonLabel*. L'immagine seguente mostra il menu del progetto di esempio Mobile Design Patterns che può essere utilizzato come referenza.



Se il menu viene definito dinamicamente invece che a design time, sarà possibile inserire una datamap, eventualmente con raggruppamento, per generare la lista degli item.

Per ogni riga del menu dovrà essere gestito l'evento *onClick*, in modo da aprire la videata corrispondente. Continuando l'analisi dell'esempio precedente vediamo il seguente codice:

```
$first.onClick = function (event)
{
  view.push(App.FirstPage, {root : true, remove : true});
  view.hilightMenu(this);
  view.hideMenu();
};
```

La prima riga chiama il metodo *push* del page controller per aprire la videata corrispondente alla voce di menu, chiudendo tutte quelle precedentemente aperte. Il metodo *push* sarà analizzato in seguito.

La seconda riga chiama il metodo *hilightMenu* del page controller che aggiunge la classe CSS *menu-hilight* alla voce di menu. Se questa classe viene definita nel foglio CSS dell'applicazione, sarà possibile evidenziare la voce di menu attiva con un colore di sfondo. Un esempio di definizione della classe può essere la seguente:

```
.menu-hilight {
  background-color : #0fb340;
  color : white;
  border-radius: 0;
}
.menu-hilight ion-icon {
  color: white;
}
```

Infine l'ultima riga chiama il metodo *hideMenu* del page controller che nasconde il menu se il device è uno smartphone o un tablet verticale.

La gestione del menu richiede inoltre che, in ogni videata che può essere aperta tramite menu, la proprietà *menuButton* della *navbar* abbia valore *true* e che venga implementato l'evento *onMenuButton* della *navbar* con il seguente codice:

```
$navbar.onMenuButton = function (event)
{
  App.Pages.showMenu(app);
};
```

In questo modo, se il device è di tipo smartphone o tablet verticale, apparirà un pulsante in alto a sinistra per aprire il menu e, quando l'utente lo cliccherà, verrà chiamato il metodo *showMenu* che mostra il menu con un'animazione. In caso di tablet orizzontale o desktop il framework fa in modo che il menu sia sempre visibile e che i pulsanti di apertura del menu nelle navbar non appaiano. È possibile modificare il comportamento standard tramite la proprietà *exposed* dell'elemento *IonSplitPane* contenuto nella propria videata *Pages*.

L'ultimo metodo del page controller da utilizzare per il controllo del menu è *enableMenu*. Esso deve essere chiamato quando la sessione di lavoro entra in uno stato in cui il menu non può essere usato, come ad esempio durante la fase di login.

In questi casi è necessario chiamare il metodo *enableMenu* del page controller passando il parametro *false* per impedire l'uso del menu dell'applicazione. Questa chiamata può essere effettuata, ad esempio, nell'evento *onLoad* della videata di login, come mostrato di seguito.

```
App.Login.prototype.onLoad = function (options)
{
  App.Pages.enableMenu(app, false);
  ...
};
```

Se il login ha successo, è necessario attivare nuovamente il menu chiamando *enableMenu*,questa volta passando *true* come parametro.

Aprire una videata

L'apertura di una videata avviene tramite il metodo *push* del page controller, solitamente chiamato in modo statico tramite l'istanza di default: *App.Pages.push(app, App.Login).*

Il metodo *push* ammette due parametri. Il primo è la classe che rappresenta la videata da aprire, il secondo è un oggetto che rappresenta le opzioni di apertura che verranno passate all'evento *onLoad* della videata.

L'apertura potrà avvenire in diversi modi in funzione delle opzioni passate come parametro. In particolare il page controller gestisce le seguenti:

- *root:true* - La videata rappresenta un nuovo percorso di navigazione; si consiglia di utilizzare questa opzione quando si apre una videata tramite il menu dell'applicazione e, in questi casi, di applicare sempre anche l'opzione *remove:true*. Specificando questa opzione non viene usata l'animazione di apertura, ma la videata appare immediatamente.
- *remove:true* - Dopo aver aperto la nuova videata, tutte le videate precedenti vengono chiuse.
- *remove:<n>* - Dopo aver aperto la nuova videata, le *n* videate precedenti vengono chiuse. *n* deve essere un numero intero maggiore di zero.
- *animate:false* - Disabilita l'operazione di apertura.
- *popup:true* - Apre la videata come popup. In questo caso è possibile utilizzare anche le altre opzioni ammesse dal metodo *show* di una videata come, ad esempio, *modal:true* per ottenere un popup modale.
- *autoclose:false* - In caso di popup impedisce di chiudere la videata cliccando all'esterno di essa.
- *wait:false* - La videata appare prima possibile. Se non specificato, il page controller aspetta 100 ms prima di effettuare l'animazione di apertura per dare tempo alla videata di preparare il proprio aspetto visuale.

Si ricorda che nelle opzioni di apertura si potranno specificare le opzioni personalizzate che devono essere passate all'istanza di videata che viene aperta. Ad esempio questa riga di codice passa il documento da una lista ad una videata di dettaglio.

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

Chiudere una videata

La chiusura di una videata avviene tramite il metodo *pop* del page controller. Tale metodo ammette due parametri. Il primo rappresenta il numero di videate da chiudere (per default una), il secondo è un oggetto che rappresenta le informazioni che verranno passate alla videata che diventa attiva.

L'operazione di chiusura avviene come segue:

1) viene notificato l'evento *onBack* sulla videata che diventa attiva, passando le eventuali informazioni inserite come secondo parametro.

2) viene eseguita l'animazione di “ritorno indietro” dipendente dal dispositivo utilizzato.
3) Quando l’animazione è terminata ed è diventata già attiva una nuova videata, vengono chiuse le videate indicate. A tali videate viene notificato l’evento *onClose*.

Come si può notare, il metodo *pop* non è interrompibile, cioè le videate vengono rimosse comunque dal video. Per tale ragione si consiglia di non restituire mai il valore *false* nell’evento *onClose* perché in tal caso la videata sparisce dal video, ma la memoria allocata non viene liberata.

Di solito il metodo *pop* viene chiamato dal codice applicativo quando si verificano le condizioni per tornare alla videata precedente, come vediamo nell’esempio seguente:

```
// Chiude la videata attuale e ritorna alla precedente
App.Pages.pop(app);
```

Esiste un caso particolare in cui la chiamata al metodo *pop* viene generata dal framework. Questo avviene quando l’utente preme il pulsante *back* della *navbar* di una videata o esegue il gesto di *edge swipe* dal lato sinistro dello schermo. In questo caso, se per la videata non viene implementato l’evento *onBackButton* della *navbar*, il framework esegue automaticamente il metodo *pop* chiudendo la videata attuale.

Se si desidera impedire la chiusura della videata o comunque controllare il processo di chiusura, è necessario implementare l’evento *onBackButton* della *navbar* e nel codice che gestisce l’evento scegliere se chiamare o meno il metodo *pop*. Se l’evento *onBackButton* viene implementato, il framework non genera la chiamata automatica a *pop*.

Inviare messaggi alle videate

Il framework di Instant Developer Cloud comprende alcuni sottosistemi che sono in grado di generare eventi in maniera indipendente dall’interfaccia utente.

Ad esempio, se un utente effettua l’upload di una fotografia catturata con l’applicazione, al termine del caricamento viene notificato alla sessione l’evento *onTransfer* da parte del plugin fotocamera. Un altro caso riguarda il sottosistema di sincronizzazione, che notifica eventi relativi ad aggiornamenti dei documenti o al processo di sincronizzazione stesso.

In questi casi è utile informare le videate aperte delle novità avvenute. A tal fine il page controller contiene il metodo *postMessage* con cui è possibile avvisare dell’accaduto la videata attiva o tutte le videate aperte. *postMessage* ammette un solo parametro di tipo oggetto che contiene tutte le informazioni da notificare. Se la proprietà *bc* di questo oggetto è *true*, allora il messaggio verrà notificato a tutte le videate aperte, altrimenti solo a quella attiva.

Le videate ricevono le informazioni implementando l’evento *onMessage*. Vediamo un esempio di codice relativo all’upload di foto tramite fotocamera.

Nel plugin *camera* contenuto nell’oggetto *device* definito a livello di applicazione possiamo implementare l’evento *onTransfer* scrivendo il seguente codice:

```
app.device.camera.onTransfer = function (event)
{
  // Post the photo to the current view (edit item page)
  App.Pages.postMessage(app, {type : "camera-transfer", event : event});
};
```

A questo punto nella videata che deve visualizzare la foto, potremo utilizzare le informazioni arrivate dal plugin camera:

```
App.EditItem.prototype.onMessage = function (message)
{
  if (message.type === "camera-transfer") {
      view.itemToEdit.Photo = app.sync.serverUrl + "/" + app.sync.appName +
                              "/files/uploaded/" + view.fileName;
      view.itemToEdit.FileName = view.fileName;
  }
};
```

È possibile visualizzare il codice completo nel progetto di esempio ToBuy.

Personalizzare il funzionamento del page controller

Come abbiamo visto nei paragrafi precedenti, il page controller è implementato tramite codice Instant Developer Cloud. È quindi possibile leggere il codice completo del componente aprendo la libreria importata *Ionic* e sfogliando i metodi della videata *MainPage*.

Siccome il metodo predefinito per implementare un page controller è quello di creare nella propria applicazione una videata che estende *Ionic.MainPage*, sarà possibile personalizzare il funzionamento del page controller sovrascrivendo i metodi base.

Se, ad esempio, si sovrascrive il metodo *canClose* di *MainPage* inserendo un metodo sovrapposto nella propria videata *Pages*, si potrà definire se l'applicazione si deve chiudere al momento in cui l'utente clicca ancora il pulsante *back* e non ci sono videate aperte nel page controller.

Se, ad esempio, si vuole chiedere all'utente cosa preferisce fare, si potrebbe utilizzare il seguente codice:

```
App.Pages.prototype.canClose = function ()
{
  return yield app.confirm("Vuoi chiudere l'applicazione?");
};
```

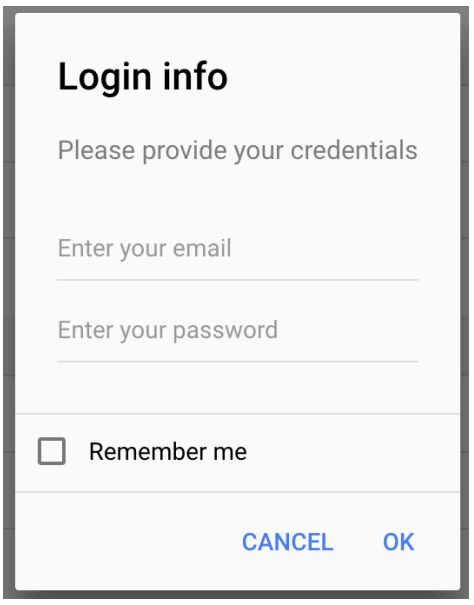
Il metodo app.popup

Oltre alle videate costituite da elementi visuali, nelle applicazioni web e mobile vengono utilizzati ulteriori componenti grafici con compiti specifici, come ad esempio i messaggi di attenzione o a scomparsa, le richieste di conferma, l'inserimento di un dato, gli spinner per segnalare il progresso di un'operazione in corso.

Per ottenere questi comportamenti, il framework IonicUI contiene un metodo di sessione chiamato *app.popup* che consente di visualizzare il componente specificato dalle opzioni e poi restituisce il risultato che l'utente ha indicato. Se ad esempio, si desidera richiedere dei dati all'utente, è possibile utilizzare *app.popup* con il seguente codice:

```
var ris = yield app.popup({
    type : "alert",
    title : "Login info",
    message : "Please provide your credentials",
    inputs : [{
      id : "username",
      type : "email",
      placeholder : "Enter your email",
      focus : true
    }, {
      id : "pwd",
      type : "password",
      placeholder : "Enter your password"
    }, {
      id : "rm",
      type : "checkbox",
      label : "Remember me"}],
    buttons : ["Cancel", "Ok"]
    });
```

Il risultato è la videata mostrata nell'immagine seguente:



Quando l'utente clicca su un pulsante, il metodo *popup* restituisce un oggetto che riporta i dati inseriti dall'utente e quale pulsante è stato cliccato.

Il tipo di popup viene selezionato tramite la proprietà *type* dell'oggetto passato al metodo. Sono disponibili i seguenti tipi:

1) *alert*: una videata costituita da titolo, messaggio e pulsanti. Può contenere anche controlli visuali come input, radio button o checkbox.
2) *actionsheet*: un menu di opzioni (massimo 6) che appare dal fondo del dispositivo. Si consiglia l'uso di questo popup solo in caso di applicazioni mobile, soprattutto se su smartphone.
3) *loading*: un messaggio con un elemento spinner che indica che c'è una operazione in corso. L'interfaccia utente è bloccata.
4) *toast*: un messaggio che appare dal basso o dall'alto dello schermo e scompare automaticamente dopo qualche secondo.
5) *menu*: un popup che contiene un menu di scelta. Può essere fatto apparire vicino ad un pulsante nell'interfaccia utente.

Per conoscere tutte le opzioni aggiuntive, si consiglia di leggere la documentazione in linea. del metodo *app.popup*. Per ulteriori esempi è disponibile il progetto: Mobile Design Patterns.

Si segnala infine che il framework IonicUI ridefinisce l'output grafico dei metodi *app.alert*, *app.confirm* e *app.prompt*: essi vengono visualizzati tramite *app.popup*.

Videate come elementi visuali

Nel paragrafo relativo alla visualizzazione a schede (tabbed view), abbiamo visto che gli elementi visuali della pagine relative ad ognuna delle schede devono essere definiti all'interno delle schede stesse.

Questo può rendere più difficile la realizzazione di videate con molte schede, in quanto nella stessa videata deve essere definito il contenuto di ogni scheda e tutto il codice relativo ad esso.

Instant Developer Cloud comprende un meccanismo di composizione che rende decisamente più facile approcciare questa classe di problemi. Infatti, attivando la proprietà *design time* di una videata essa diventa disponibile anche come elemento visuale da includere all'interno di altre videate.

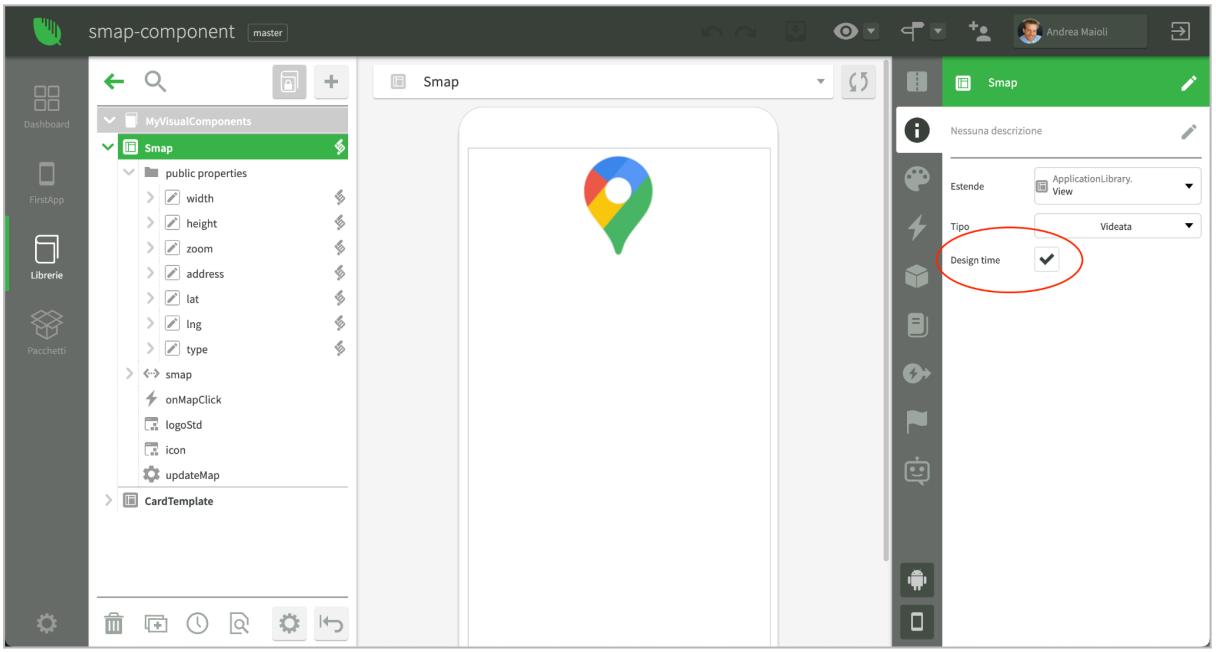
In questo modo si risolve il problema delle tabbed view, in quanto il contenuto di ogni scheda può essere definito in una videata separata. Poi ognuna di queste videate sarà utilizzata come elemento visuale da includere in una scheda della tabbed view.

Ma c'è di più: le videate attivate come elementi visuali possono essere utilizzate più volte in contesti diversi, addirittura all'interno di un template di una datamap per visualizzarne un'istanza per ogni documento di una collection. In questo modo è quindi possibile creare una libreria di componenti visuali da utilizzare in ogni proprio progetto.

Interfaccia dell'elemento videata

Quando una videata viene attivata per l'utilizzo come elemento visuale, essa deve diventare un componente vero e proprio, pertanto deve poter esprimere un'interfaccia completa in termini di proprietà, metodi ed eventi, come avviene per gli altri elementi visuali definiti nelle librerie.

Nell'immagine seguente vediamo un esempio di videata utilizzata come elemento visuale. La videata *Smap*, infatti, serve per visualizzare una mappa di Google statica. Nelle proprietà della videata possiamo notare il flag *design time* attivato.



In queste condizioni, le proprietà della videata, come ad esempio *width*, possono essere definite come parte dell'interfaccia del componente attivando anche per esse il flag *design time*. In questo caso, si potrà aggiungere alla proprietà l'evento *onChange*, che viene notificato quando tale proprietà cambia a runtime. Ad esempio per la proprietà *width* il codice dell'evento *onChange* è il seguente:

```
App.MyVisualComponents.Smap.prototype.width_onChange = function (newValue)
{
  setImmediate(function () {
    view.updateMap();
  });
};
```

Fra gli oggetti inclusi nella videata vediamo anche il metodo *onMapClick*, che tramite le sue proprietà è stato definito come evento di design time. Tale metodo, infatti, è pensato per essere sovrascritto nell'istanza di videata usata come elemento visuale. Quando all'interno della videata viene chiamato il metodo *view.onMapClick*, in realtà si notifica tale evento a chi ha usato la videata come elemento.

All'interno della videata il metodo non ha una sua implementazione, infatti viene definito come segue:

```
App.MyVisualComponents.Smap.prototype.onMapClick = function ()
{
};
```

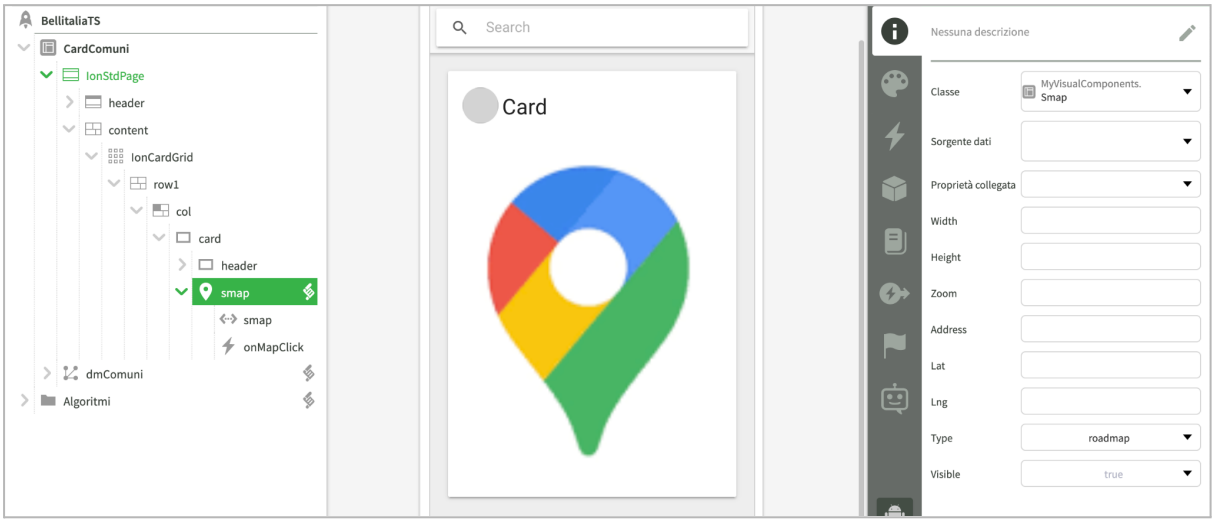
Questo metodo viene chiamato quando l'utente clicca sull'immagine statica della mappa, tramite il seguente codice:

```
$imageMap.onClick = function (event)
{
  view.onMapClick();
};
```

Quindi, quando l'utente clicca sulla mappa, che è un elemento interno al componente *Smap*, il codice della videata richiama il metodo *onMapClick* che normalmente non fa nulla, a meno che non sia stato ridefinito da chi ha utilizzato la videata *Smap* come elemento. In tal caso la chiamata assume il significato di notifica dell'evento di clic.

Per quanto riguarda i metodi della videata, come ad esempio *updateMap*, essi sono sempre utilizzabili come metodi di istanza, quindi, quando la videata viene usata come elemento, essi saranno richiamabili sull'istanza di videata che lo rappresenta, a meno che non siano stati definiti come privati o protetti.

Vediamo adesso un esempio di utilizzo della videata *Smap* all'interno di un'altra videata che la utilizza come elemento.



Nell'immagine precedente possiamo notare che la videata *Smap* è stata utilizzata come elemento interno di una card che costituisce il template di una datamap. Fra le proprietà dell'elemento *Smap* troviamo sulla destra quelle definite come design time, come *width*, *height*, eccetera. Inoltre per l'elemento *Smap* è stato gestito l'evento *onMapClick*.

Visualizzazione a design time di videate usate come elementi

Occorre tenere presente che il codice applicativo di una videata usata come elemento sarà eseguito solo quando l'applicazione è in esecuzione, sia in anteprima che in produzione. Questo significa che quando si visualizza l'anteprima di una videata nell'editor delle videate, in quel momento il codice delle videate usate come elementi non viene eseguito, quindi il valore delle proprietà di design time non influenza ciò che si vede nell'anteprima.

Ecco perché nell'immagine precedente viene visualizzato un logo statico a forma di pin: esso rappresenta il valore di design time della proprietà *src* dell'elemento *img* usato da *Smap* per visualizzare la mappa statica.

Anche impostando i valori delle proprietà a design time sull'elemento *Smap,* non si avrebbe alcun effetto sull'anteprima nell'IDE, perchè tali valori vengono considerati dal codice della videata che viene eseguito soltanto quando l'applicazione viene lanciata in anteprima dall'IDE o utilizzata in ambiente di produzione.

Personalizzazione di IonicUI

In quest'ultimo paragrafo vedremo quali tecniche sono previste per personalizzare la resa grafica delle applicazioni sviluppate con il framework IonicUI.

Il tema

Il tema è un oggetto JavaScript condiviso tra la sessione di lavoro e il codice del browser che gestisce l'interfaccia dell'applicazione. Le proprietà del tema permettono di personalizzare l'aspetto e i comportamenti del framework IonicUI.

Le proprietà specifiche riconosciute dal tema IonicUI sono presenti nella classe *IonTheme* contenuta nella libreria *IonicFramework* e sono le seguenti:

1) *ionPlatform*: rappresenta la grafica di base dell'applicazione e può valere "md" per Material Design oppure "ios" per dispositivi Apple. Il valore predefinito viene calcolato automaticamente in base al dispositivo utilizzato, ma può essere modificato nell'evento *onStart* dell'applicazione, prima dell'apertura della prima videata.
2) *primary*: colore primario dell'applicazione.
3) *secondary*: colore secondario dell'applicazione.
4) *danger*: colore utilizzato per messaggio di errore o azioni distruttive.
5) *light*: sfumatura di colore utilizzata per visualizzare elementi chiari.
6) *dark*: sfumatura di colore utilizzata per visualizzare elementi scuri.
7) *bright*: colore utilizzato per visualizzare elementi ad alto impatto visivo.
8) *vibrant*: colore alternativo utilizzato per visualizzare elementi ad alto impatto visivo.
9) *focus*: colore della sottolineatura di elementi che hanno il fuoco (usato se *ionPlatform="md"*).
10) *ionIcons*: versione del set di icone da utilizzare. I possibili valori sono 4 e 5. Si consiglia di usare il valore 4.
11) *iconSet*: determina quale set di icone deve essere attivato. I possibili valori sono "md" e "ios" e il default è uguale ad *ionPlatform*.

12) *tippy*: parametri da passare al sistema di gestione dei tooltip tippy.js in formato oggetto JavaScript.
13) *darkMode*: attiva o meno il *darkMode* per la sessione. I possibili valori sono “true”, “false” e “auto”. Il default è “auto”.
14) *darkPercent*: percentuale di diminuzione della luminosità dei colori del tema quando si attiva il tema scuro. Default 0.30.
15) *gmapKey*: comunica al browser il valore della chiave *gmap* da utilizzare per visualizzare le mappe. Deve essere impostato prima di caricare una mappa per la prima volta e deve iniziare con “key=”. Ad esempio: “key=aIsalz3fg4…”.

È possibile modificare i parametri del tema sia a design time che da codice. Per modificare i parametri di tema a design time si deve procedere come segue:

1) Creare all’interno di una libreria una classe di tipo *Tema* che estende *IonicFramework.IonTheme*.
2) Inserire nella classe le proprietà con il nome corrispondente ai parametri del tema da modificare. Impostare quindi il valore iniziale di ogni proprietà al valore da assegnare alla corrispondente proprietà di tema.
3) Selezionando l’oggetto applicazione, impostare come proprietà *Tema* la classe tema appena creata.

A questo punto le proprietà del tema verranno applicate anche alle videate in anteprima nell’editor delle videate. Dopo aver modificato il valore di una proprietà del tema, può essere richiesto l’aggiornamento manuale dell’anteprima dell’editor delle videate.

La modifica di una proprietà di tema da codice è molto semplice. Se tale modifica avviene nell’evento *onStart* dell’applicazione è necessario solamente modificare l’oggetto *app.theme*. Ad esempio per modificare un colore del tema è possibile scrivere:

```
app.theme.secondary = "#51d677";
```

In questo caso la modifica non sarà visibile nell’anteprima dell’editor delle videate ma solo quando l’applicazione viene lanciata dall’IDE o in produzione.

Se la modifica al tema avviene dopo l’apertura della prima videata, sarà necessario chiamare il metodo *app.updateTheme()* per renderla effettiva. È possibile passare le proprietà da aggiornare direttamente a tale metodo, ad esempio:

```
app.updateTheme({secondary: "#51d677", ...});
```

Editing del CSS

La modalità più completa per ottenere qualunque effetto grafico di interesse è quella di personalizzare il CSS del framework IonicUI. Questa modalità è estremamente potente, ma richiede la conoscenza degli inspector web dei browser e del linguaggio di configurazione dello stile degli elementi (CSS).

Per ottenere questo risultato è sufficiente aggiungere un foglio CSS alla propria applicazione e poi inserire le regole aggiornate.

Per testare quali selettori specificare e quali regole aggiornare, si consiglia di ispezionare il DOM della pagina browser in cui sono presenti gli elementi da modificare e poi testare al volo le modifiche necessarie. Dopo aver trovato la configurazione richiesta, si può procedere a definire il contenuto del proprio foglio CSS.

Il metodo setAttribute

Alcuni elementi visuali del framework IonicUI vengono visualizzati creando diversi oggetti nel DOM del browser. La modifica di uno stile inline di un elemento visuale a design time o tramite codice agisce solo sull'oggetto DOM principale, solitamente quello che contiene tutti gli altri.

In alcuni casi, per modificare lo stile degli oggetti interni potrebbe quindi essere necessaria la creazione di una classe CSS apposita che possa arrivare agli oggetti interni attivando i selettori opportuni.

Una soluzione alternativa può essere quella di utilizzare il metodo *setAttribute* dell'elemento che, oltre a poter impostare gli attributi dell'elemento, è in grado di operare anche sugli oggetti interni.

Se ad esempio si volesse modificare il colore di sfondo di una *navbar* senza usare i colori del tema, potremmo scrivere il codice seguente:

```
$navbar.setAttribute("toolbar-background style",
                     "background-color:yellow");
```

Questa riga di codice fa sì che l'attributo *style* non venga modificato sull'oggetto DOM principale della *navbar*, ma su un suo sotto-oggetto che ha come classe *toolbar-background*, e che è quello che visualizza lo sfondo.

Per maggiori informazioni sulle varie configurazioni ammesse dal metodo *setAttribute*, si consiglia di leggere la documentazione in linea.

Configurazione dei ruoli e degli accessi

Molte applicazioni sono pensate per essere utilizzate da utenti con diversi ruoli applicativi. In questi casi è necessario poter riconfigurare l'interfaccia dell'applicazione in modo da adattarsi ai ruoli attivi in una determinata sessione, ad esempio nascondendo le funzioni non disponibili.

L'implementazione di queste specifiche normalmente non è semplice. Innanzitutto di frequente esse nascono o vengono modificate quando l'applicazione è già stata implementata, quindi si tratta di modificare un codice già testato e funzionante. C'è un fattore in più: il codice che modifica lo stato dell'interfaccia utente in base al ruolo potrebbe interagire con altro codice già scritto che ne gestisce lo stato in maniera autonoma. Si tratta quindi di due fattori diversi che si trovano in conflitto sull'uso delle medesime risorse.

Per spiegare meglio facciamo un esempio: il pulsante *Inizia modifica* di una videata di dettaglio deve essere gestito a livello di stato della videata, cioè se il documento mostrato è già in fase di modifica il pulsante non deve apparire, altrimenti sì. Allo stesso tempo, la visibilità dello stesso pulsante potrebbe essere controllata da un sistema di configurazione che ne vieta l'utilizzo ad un determinato tipo di utenti.

Per gestire correttamente questa situazione è necessario che tutti i punti in cui viene gestita la visibilità del pulsante vengano centralizzati e che si tenga conto anche delle specifiche relative al ruolo dell'utente. Questo può comportare un refactoring piuttosto costoso, tenendo anche conto della necessità di aggiornare il sistema di test dell'applicazione.

Un ulteriore fattore di complicazione risiede nel fatto che le specifiche relative ai ruoli molto spesso sono ricche di eccezioni, magari legate ad uno specifico utente. Per ogni singola modifica si tratta quindi di modificare il codice dell'applicazione in una parte critica: un errore di programmazione potrebbe esporre funzioni ad utenti che non dovrebbero utilizzarle e non è facile avere la certezza che questo non avvenga.

Il foglio di controllo accessi

Il framework di Instant Developer Cloud contiene una soluzione definitiva a queste problematiche: un meccanismo di configurazione basato su file di testo che permette di descrivere lo stato degli elementi dell'interfaccia utente in funzione di ruoli utente, username o lingua della sessione.

Tale meccanismo, denominato ACS, acronimo di *Access Control Sheet*, funziona in modo analogo alla configurazione grafica tramite CSS. I principi di funzionamento sono quindi i seguenti:

1. È possibile definire a runtime lo stato degli elementi dell'interfaccia utente semplicemente modificando un file di testo: non è necessario installare una nuova versione dell'applicazione.
2. Non richiede alcuna modifica al codice dell'applicazione.
3. Le specifiche ACS hanno la precedenza rispetto a quanto scritto nel codice: un pulsante nascosto a causa dei ruoli utente non può essere più reso visibile dal codice dell'applicazione nemmeno impostando a *true* la proprietà *visible*.
4. La definizione delle specifiche è molto semplice, simile alla sintassi CSS, e può agire anche a livello di intera applicazione oltre che di singola videata.

Per vedere in azione il framework ACS è disponibile il progetto di esempio ACS Design Patterns.

Configurazione del framework ACS

Per attivare il framework ACS è necessario inserire nell'evento *app.onStart,* o comunque prima di far apparire videate da configurare, alcune righe di codice che ne forniscono la configurazione.

Le proprietà ed i metodi del framework ACS sono disponibili tramite l'oggetto *app.acs*. In particolare, durante l'inizializzazione della sessione è necessario fornire lo username

dell'utente e l'elenco dei ruoli applicativi ad esso applicati. Un esempio di codice è il seguente:

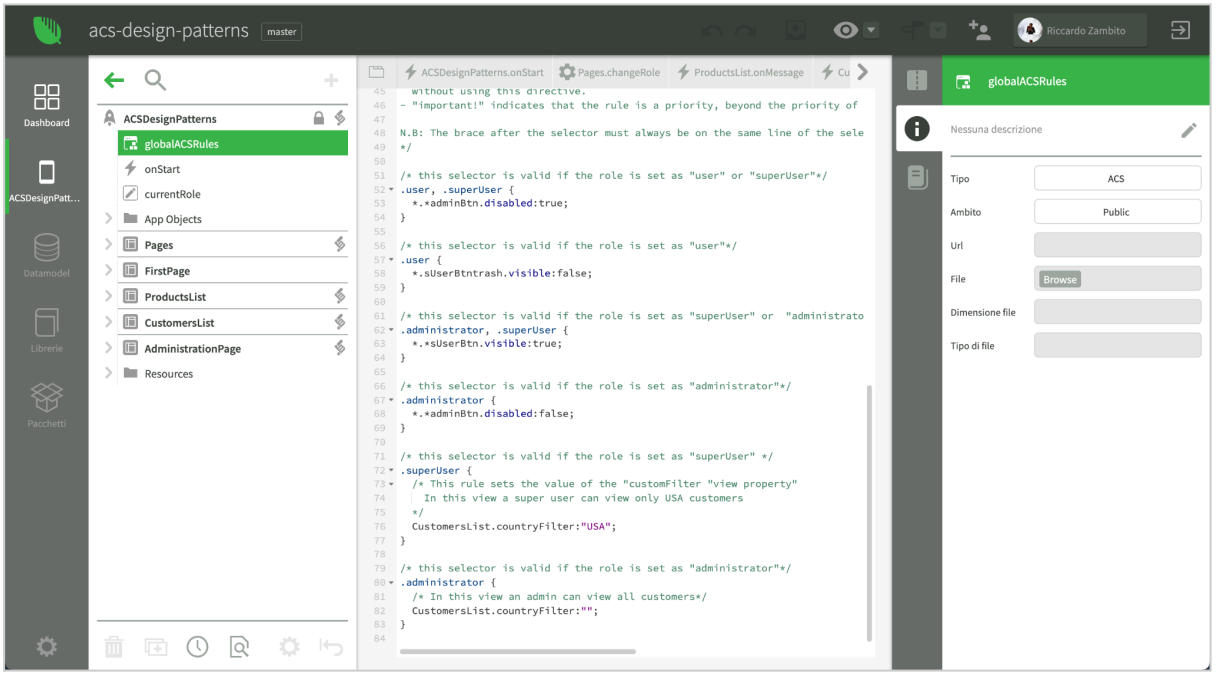
```
App.Session.prototype.onStart = function (request)
{
  ...
  app.acs.username = "rossim";
  app.acs.addRole("mailing-admin");
  app.acs.addRole("account-manager");
  ...
}
```

È importante notare che è possibile configurare per la sessione un solo username; è invece possibile aggiungere diversi ruoli utente. In questo modo è possibile gestire moduli applicativi diversi e avere ruoli specifici in ognuno di essi. L'array dei ruoli applicati alla sessione è disponibile, in sola lettura, tramite *app.acs.userRoles*. Oltre ad aggiungere ruoli è possibile rimuoverli tramite il metodo *app.acs.removeRole*.

Configurazione dell'insieme delle regole da applicare

Vediamo ora come è possibile fornire al sistema le regole da applicare. Le regole possono essere fornite come risorsa caricata nell'IDE, come se fossero un file CSS, oppure direttamente a runtime.

Per definire le regole direttamente nell'IDE è possibile aggiungere all'applicazione una risorsa di tipo ACS e poi definire le regole direttamente nell'editor. Nell'immagine seguente viene mostrata la risorsa ACS del progetto di esempio [ACS Design Patterns].



Quando viene aperta una sessione, verranno caricate tutte le risorse ACS inserite all'interno dell'applicazione o delle sue videate.

Se invece si preferisce caricare le regole ACS a runtime, è possibile utilizzare i metodi *app.acs.load* per caricare un file di testo che contiene le regole, oppure *app.acs.loadByString* per caricare una stringa di testo che contiene le regole.

È importante considerare che le regole non vengono memorizzate a livello di sessione ma di applicazione. Quindi se si carica più volte la stessa risorsa tramite *load*, oppure si applica più volte la stringa identificata dal medesimo *id* tramite *loadByString*, non si ottiene alcun effetto, a meno di non impostare a *true* il parametro *reload* dei suddetti metodi, che si occupa di ri-elaborare per l'intera applicazione l'insieme di regole passato.

Dopo aver modificato per una sessione l'insieme delle regole, è necessario chiamare il metodo *app.acs.getRules*, che seleziona le regole da applicare alla sessione corrente in base ai parametri della stessa ed infine se ci sono videate già aperte da riconfigurare, è necessario chiamare il metodo di *view.refreshRoles* sull'istanza della videata.

Sintassi del file ACS

Un file ACS è simile come struttura ad un file CSS. È possibile inserire dei blocchi di regole, inframmezzati da commenti identificati da /* */. Un blocco di regole è costituito da un insieme di selettori e dall'insieme di regole da applicare.

Vediamo subito un primo esempio:

```
/* La mia prima regola ACS */
.admin {
  MenuPage.*.adminGroup.visible: true;
}
```

Nell'esempio i commenti sono evidenziati in verde, i selettori in nero e le regole in colore blu. È importante che la parentesi graffa aperta appaia sulla medesima riga in cui sono definiti i selettori.

Selettori

I selettori disponibili sono i seguenti:

- #<*username*> per selezionare uno specifico utente. Ad esempio *#rossim* identifica regole applicate se *app.acs.username* è uguale a *rossim*.
- .<*userrole*> per selezionare uno specifico ruolo utente. Ad esempio *.admin* identifica regole applicate se *app.acs.userRoles* contiene *admin*.
- @<lang> per selezionare una specifica lingua. Ad esempio *@en*, identifica regole applicate se *app.langCode* è uguale a *en*.
- * (asterisco) identifica regole applicate in ogni condizione.

I selettori possono essere composti in *and* scrivendoli uno di seguito all'altro separati da spazio, oppure in *or* separandoli da virgola. Ad esempio:

- *.admin @en* identifica regole applicata se per la sessione è attivo il ruolo *admin* e se la lingua è *en*.

- *.admin, .manager* identifica regole applicate se per la sessione è attivo il ruolo *admin* oppure il ruolo *manager*.

Si ricorda che i selettori devono essere scritti sulla stessa riga e che la parentesi graffa di apertura blocco deve essere sulla medesima riga.

Regole

Una regola ACS specifica il valore di una proprietà di un elemento visuale di una videata. La sintassi è la seguente:

```
<targetView>.<element chain>. … .<property>:<value><!important|!default>;
```

targetView identifica il nome della videata a cui deve essere applicata la regola. Inserendo *, la regola verrà applicata a tutte le videate.

<element chain> identifica la catena dei nomi degli elementi visuali che identificano uno specifico elemento. È possibile utilizzare * per referenziare un qualunque elemento visuale a qualunque livello.

<property> è il nome della proprietà dell'elemento da impostare.

<value> è il valore a cui verrà impostata la proprietà. Deve essere nel formato nativo della proprietà, cioè se la proprietà è di tipo numerico, deve essere specificato un numero e così via.

<!important> è un modificatore opzionale che indica che la regola ha una priorità maggiore delle altre che non hanno il modificatore.

<!default> è un modificatore opzionale che indica che la regola imposta solo il valore di default della proprietà e consente al codice dell'applicazione di modificarla successivamente. Di solito, invece, una proprietà impostata tramite ACS risulta bloccata al valore assegnato.

Vediamo alcuni esempi:

Nasconde l'elemento di nome *firstItem* posizionato nella catena indicata della videata *Pages*:
```
Pages.split.menu.menucontent.menuList.firstItem.visible:false;
```

Nasconde tutti gli elementi che si chiamano *firstItem* nella videata *Pages*:
```
Pages.*.firstItem.visible:false;
```

Nasconde tutti gli elementi che si chiamano *btnSave* in tutte le videate:
```
*.*btnSave.visible:false;
```

Nasconde tutti gli elementi il cui nome contiene *btn* in tutte le videate:
```
*.*btn.visible:false;
```

Imposta la proprietà *innerText* di ogni elemento chiamato *banner*, ma il codice dell'applicazione può cambiarla:

```
*.*.banner.innerText:"Questo è un banner" !default;
```

Priorità delle regole

La priorità delle regole applicate dipende dal tipo di selettori e dal flag *!important* specificato nella regola stessa.

L'ordine di priorità dei selettori è il seguente:

1) # (utente)
2) . (ruolo)
3) @ (lingua)
4) * (sempre valido)

Le regole vengono applicate dalla priorità più bassa verso la più alta, in modo che le maggiori prevalgono sulle altre.

Per maggiori dettagli sull'uso di ACS, si consiglia di studiare il progetto di esempio ACS Design Patterns.