HTML Mobile App: il funzionamento delle view

Nell'ultimo post ho discusso l'importanza della *view* in ambito web, e i motivi secondo cui secondo me tale concetto dovrebbe essere preso in considerazione da qualunque sviluppatore *front-end.*

In questo articolo voglio approfondire il discorso delle **view** e di come poterle utilizzare più concretamente in un'eventuale **applicazione javascript,** usando come punto di riferimento uno dei **framework MVC** più utilizzati: **Backbone.js.**

Le **view** (o **viste)** sono **rappresentazioni visuali dei model** (ovvero la definizione dei **dati dell'applicazione),** e li mostrano **nell'interfaccia** secondo il loro **stato attuale.** Il loro scopo è quello di costruire e mantenere aggiornati uno o più elementi DOM.

Il lavoro principale di una view è quindi quello di **osservare un model** e di **aggiornare** il proprio **markup HTML** di conseguenza ogni volta che quel model subisce dei cambiamenti, in modo da mantenere il **DOM dell'interfaccia** costantemente **sincronizzato** con i dati dell'applicazione.

Lo sviluppatore dà la possibilità all'utente di interagire con le view per visualizzare gli attributi di un modello e poterli modificare.

Se i dati subiscono delle modifiche, il **markup della stessa view** o quello di altre view che stanno "osservando" il modello modificato, viene automaticamente aggiornato.

Il tutto accade solitamente secondo una **logica user-friendly,** (ovviamente questo dipende dallo sviluppatore).

HTML5 Mobile App: la logica degli osservatori

Per comprendere meglio il concetto degli osservatori autonomi, prendiamo l'esempio concreto di Tint, una delle mie HTML5 Mobile App distribuite su App Store.

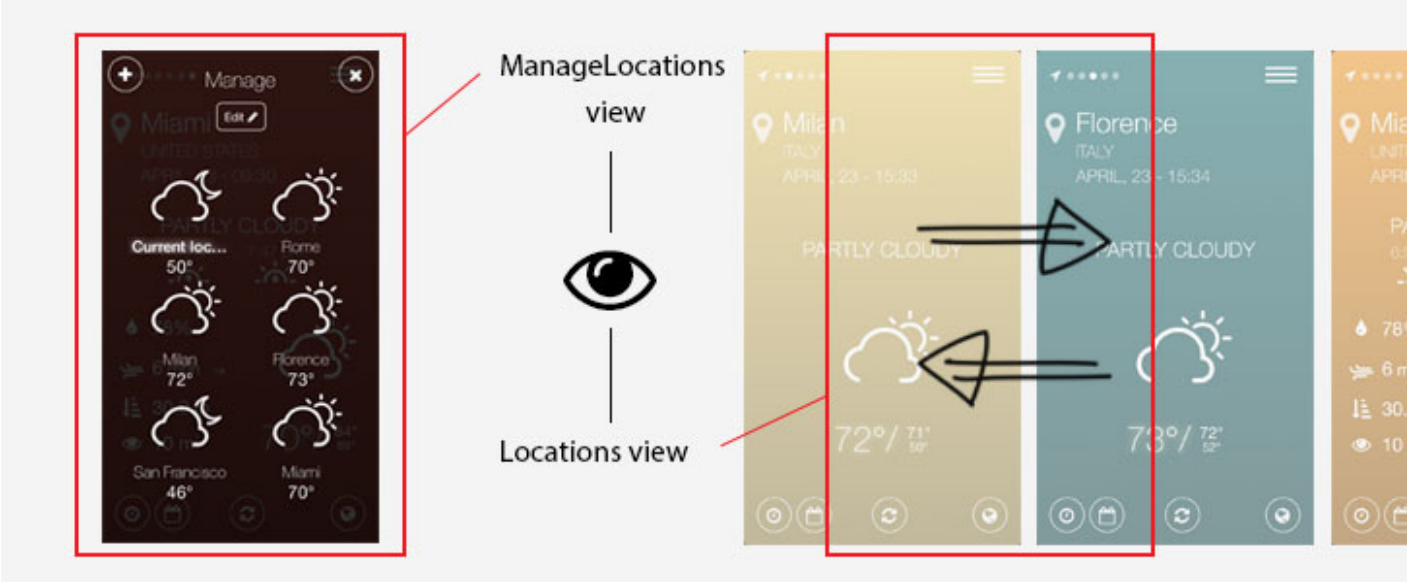
Se vuoi realizzare anche tu la tua mobile app in HTML5, CSS3 e Javascript, forse potrebbe interessarti l'e-book che sto scrivendo sull'argomento: HTML Mobile Accelerato.

Consideriamo, tra le varie view che compongono l'interfaccia, 2 views in particolare:

1. la view *"ManageLocations":* quella che consente di aggiungere, eliminare o modificare l'ordine delle località

2. la view *"Locations":* che viene mostrata subito dopo l'avvio dell'applicazione ed offre all'utente la possibilità di scorrere orizzontalmente le varie località, secondo l'ordine con cui esse sono state organizzate nella view *"ManageLocations".* Questa view si preoccupa di creare, per il model di ogni località, la relativa sotto-view, inserendola al suo interno.

L'applicazione è impostata in modo tale che la view **"Locations"** osservi i cambiamenti di una collection denominata **"LocationCollection",** ovvero il raggruppamento di tutti i model di ciascuna località aggiunta dall'utente.



Così, non appena l'utente aggiunge, elimina o modifica l'ordine di una località usando la view delle impostazioni **("ManageLocations"),** questa agisce di conseguenza:

1. aggiungendo **all'interfaccia** il **markup relativo** alla **view** che rappresenta la nuova località
2. **rimuovendo** quella della località eliminata
3. qualora si trattasse di un riordinamento, effettuando il **refresh** dello **slider** orizzontale

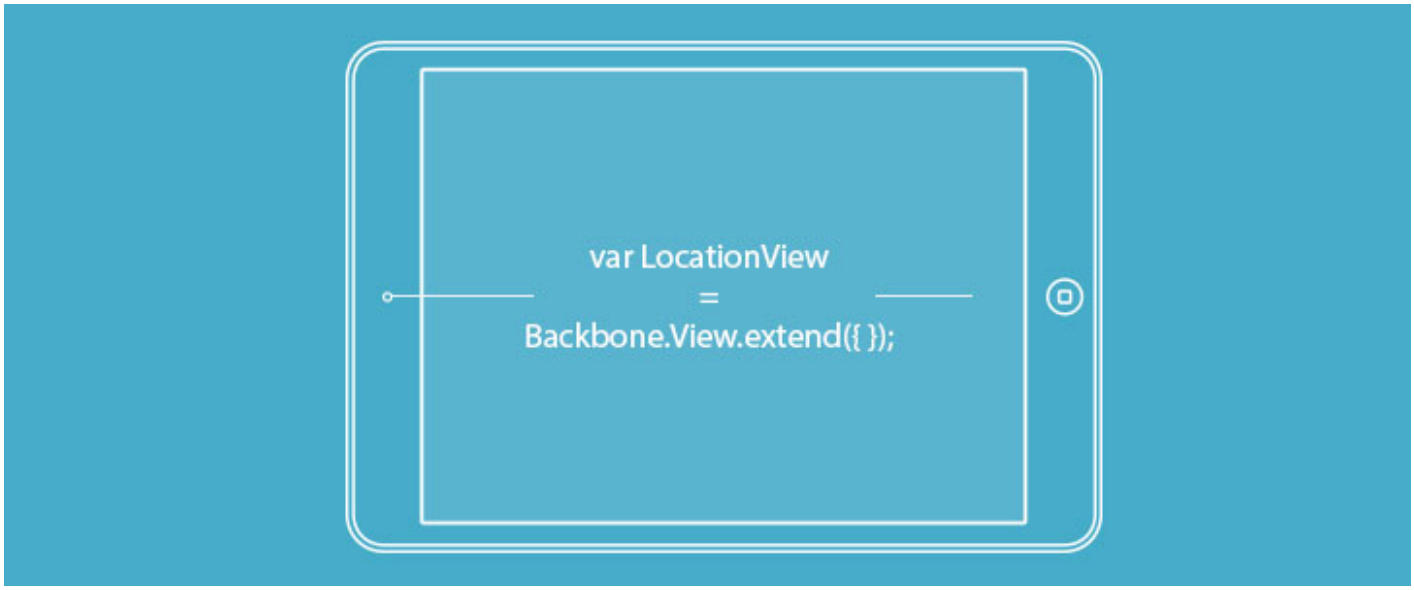
Tramite la view *"ManageLocations",* l'utente ha quindi la possibilità di modificare l'ordine di una **località** rispetto alle altre.

Quando ciò accade, viene automaticamente aggiornato il parametro "order" per ogni view presente nella collection *"LocationCollection",* facendo in modo che la view "locations" si aggiorni di conseguenza.

Ciò rappresenta il classico esempio in cui l'utente interagisce con l'interfaccia per **modificare** i **dati dell'applicazione.**

Solitamente la modifica dei dati da parte dell'utente viene gestita dai **controllers,** anche se su **Backbone.js** tale componente è assente, almeno per quanto riguarda la terminologia (più avanti vedremo come funziona Backbone per controllare le interazioni dell'utente).

HTML5 Mobile App: come definire una view



Ma vediamo come definire la possibile view di una località utilizzando Backbone.js, sempre restando nell'ottica dell'esempio precedentemente esposto.

Come prima cosa definiamo la funzione ***render()*** , che è responsabile di generare il **markup HTML** della **view** utilizzando i dati del modello **"LocationModel"** (quello che rappresenta i dati di ogni località), associato alla stessa **view.**

Per farlo ci serviamo del sistema di templating di underscore.js, che ci permette di generare markup dinamico a seconda dei dati passati in formato **JSON.**

Per avere maggiori informazioni sui sistemi id javascript templating, ti consiglio di leggere questo articolo:

- I sistemi di javascript templating e le migliori librerie

Una volta generato il markup, esso sarà usato come contenuto della *view,* ed inserito all'interno dell'elemento-contenitore che la rappresenta, definito sotto la proprietà **el**.

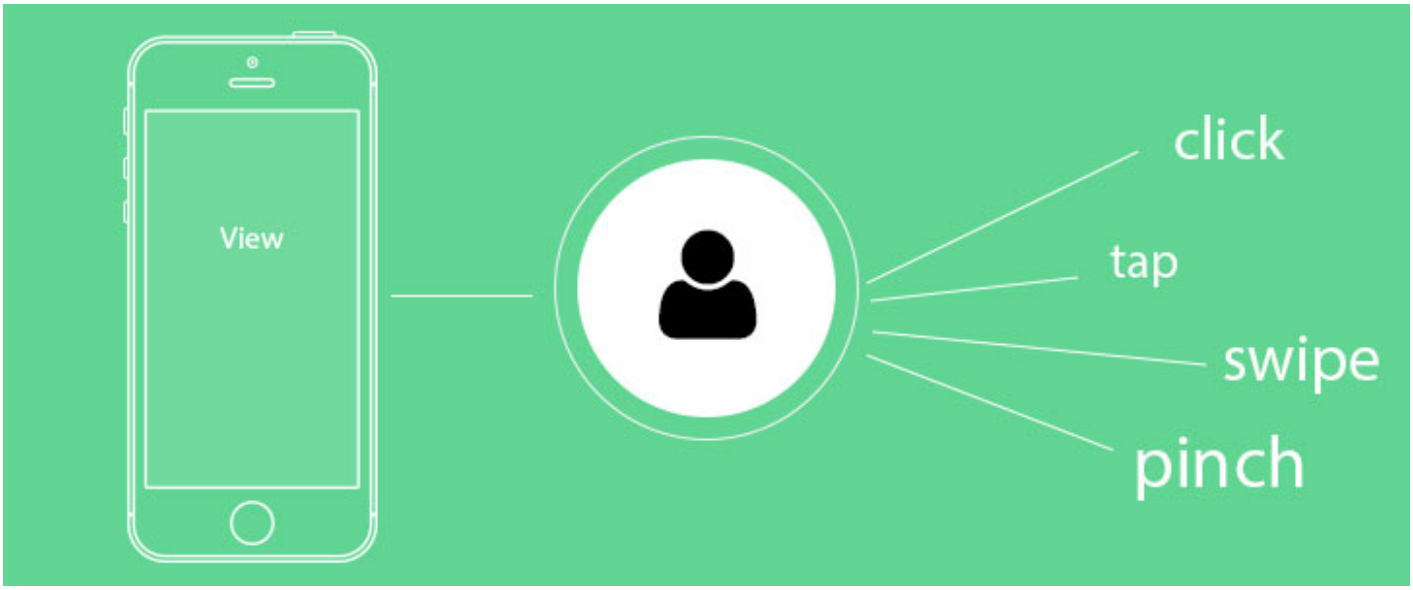
Aggiungiamo quindi la funzione **initialize(),** richiamata automaticamente quando viene creata una nuova istanza della view in questione (quindi ogni volta che l'utente aggiunge una località).

All suo interno impostiamo poi l'"osservatore":

Facciamo in modo che quando il *model* associato alla *view* subisce delle modifiche (magari anche per mezzo di un'altra *view),* sia richiamata di nuovo la funzione **render(),** al fine di eseguire l'aggiornamento del *markup* della *view* in base ai nuovi dati appena modificati.

Ciò permetterà di tenere costantemente aggiornata la view, in modo completamente autonomo.

HTML5 Mobile App: come gestire le interazioni dell'utente



Ora, in molti tipi di frameworks gli eventi di interazione dell'utente sono gestiti all'interno del controller, in modo tale che ogni componente che definisce la struttura del progetto abbia il rispettivo compito, separatamente dagli altri componenti.

Tuttavia **Backbone** rappresenta una variante del **pattern MVC,** in cui i compiti solitamente gestiti dal controller sono delegati alla **view.** Pertanto questo framework permette di definire in ogni view anche gli eventi associati ai vari elementi DOM presenti all'interno del **markup generato,** tramite l'attributo **events.**

Vediamo come gestire un'ipotetico evento **"click"** per un elemento, in modo da modificare i dati

del *model* associato alla *view,* e scatenando di conseguenza la funzione **render()** per l'aggiornamento del *markup* interno:

Ma cosa intendo esattamente per **"modello associato"?** O meglio, quando questo modello viene associato alla view?

La risposta è semplice, nel momento in cui la view è inizializzata, ovvero quando un nuovo model necessita di essere rappresentato nell'interfaccia.

Per tornare al nostro esempio concreto, consideriamo la situazione in cui utilizzando Tint, un utente decide di aggiungere una nuova località alla lista: appena egli ha selezionato il luogo di cui vuole conoscere le previsioni, ci basterà eseguire il seguente **codice Javascript:**

Conclusioni

Scrivere il codice di qualsiasi progetto secondo un termine che i **software engeener** americani chiamano **"spaghetti code"** (usano proprio un termine italiano!), ovvero scrivere il proprio codice come capita e **senza alcuna logica** o **architettura,** può andare bene per normali siti web o applicazioni dalla complessità minima.

Ma appena ci affacciamo in progetti più complessi, oppure qualora quelli che abbiamo realizzato necessitano di un'estensione, il codice prodotto tramite questo metodo di sviluppo risulta molto difficile sia da rileggere che da mantenere.

Pertanto diventa un imperativo per qualsiasi sviluppatore Javascript moderno capire l'importanza delle funzionalità che la logica di sviluppo MVC mette a disposizione.

Utilizzando il pattern MVC e le web views insieme a molte altre tecniche ho creato per i miei clienti oltre 10 applicazioni mobile, tra cui due progetti personali: Tint Weather App e Dieta SI o NO?.

E tu hai mai usato il **pattern MVC** nei **tuoi progetti?** Quali **framework MVC** (o varianti di MVC, MV* in generale) hai **utilizzato** o stai utilizzando?

Fammelo sapere nei commenti!