

Premessa

“Raccolta Titanium #1” è un pdf contenente le versioni testuali e, ovviamente, il codice sorgente 9 videotutorial sulle **basi di Appcelerator Titanium 3.2** pubblicati gratuitamente da Francesco Milanese sul suo canale Youtube e sul sito www.francescomilanese.com .

L'ebook, che consta di 95 pagine in formato A4, ha quindi un prezzo volutamente basso perché si rivolge a chi ha seguito i videotutorial gratuiti, li ha trovati interessanti e desidera avere un testo in pdf (con screenshots dei videotutorial) da poter consultare facilmente, mediante le funzioni di ricerca o stampandolo... o desidera ringraziare in maniera concreta l'autore dei videotutorial, ottenendo comunque una piccola “guida – reference” sulle funzioni di base di Titanium.

Come detto nei video, questa serie di tutorial è quindi a chi ha già una conoscenza di base di XML, CSS e Javascript (i linguaggi utilizzati in Titanium); non è un corso di programmazione da zero, ma una serie sulla creazione di app con Titanium, per chi deve “muovere i primi passi” in questo ambiente di sviluppo di app mobile (Android, iOS, ...) e web browser (HTML5).

Per il sommario completo: <http://www.francescomilanese.com/titanium-01-sommario.pdf>

SOMMARIO

1. Introduzione. Installazione	1
2. "Hello World". Template, emulatori. Schema MVC	9
3. Stili (CSS). Buttons (pulsanti) ed evento "tocco"	23
4. Elementi dell'interfaccia: TextField, TextArea, Slide, SearchBar e immagini	32
5. Creazione e modifica dinamica di elementi della UI	43
6. Memorizzare i dati in un DB SQLite. Le TableView	54
7. Windows e Views. Layout TabbedViews. Passare da una Window ad un'altra	71
8. Rilevare le Gestures (Shake, Swipe, Touch, Pinch) e i dati dell'accelerometro	82
9. Aprire una pagina web dall'app. Utilizzare i Moduli. Inserire delle Mappe nell'app	88

Titanium - Tutorial 0

Introduzione. Installazione

Questo è il primo tutorial – o, meglio: il **Tutorial 0**, come vedremo – su **Appcelerator Titanium**, per la realizzazione di **app per dispositivi mobile** (Android, iOS, Blackberry e Tizen) e **HTML5** (web browser).



Gli screenshot di questa guida servono solo come riferimenti, per confrontare le schermate qui presenti con quelle ottenute provando i vari tutorial; NON è necessario cercare di ricopiare gli script dagli screenshot: sono tutti presenti, con un font diverso dal resto, all'interno della guida

Titanium è un **framework**, ossia un programma contenente un ambiente e un insieme di strumenti di sviluppo, realizzato da **Appcelerator**, che consente di sviluppare applicazioni Android e iOS

Raccolta di tutorial **“Titanium #1: le basi”** dal sito *francescomilanese.com* --- © 2015

(quindi per dispositivi mobile) e HTML5 (quindi web browser), utilizzando in particolare i linguaggi **XML**, **Javascript** e **CSS**.

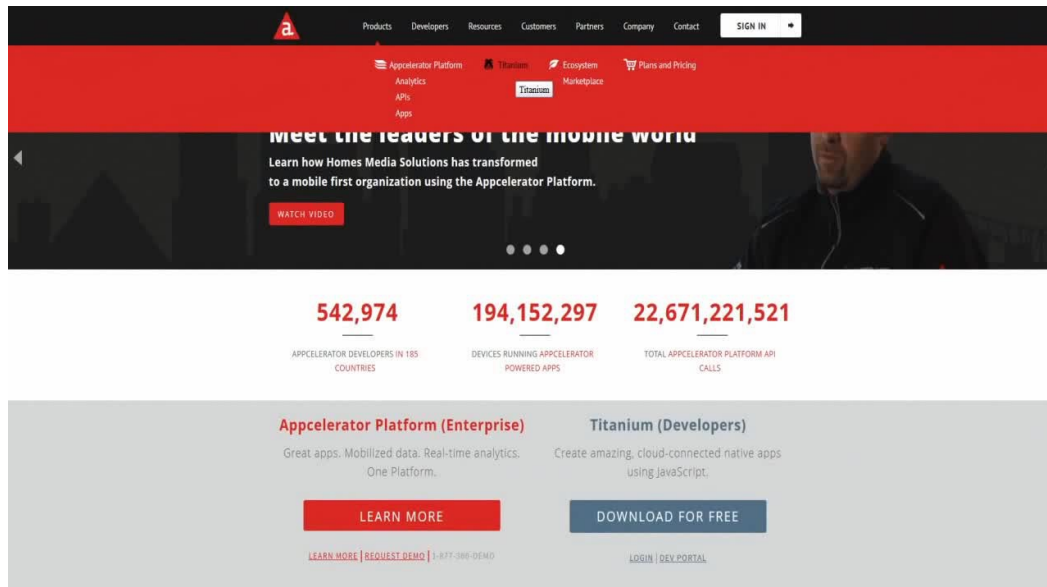
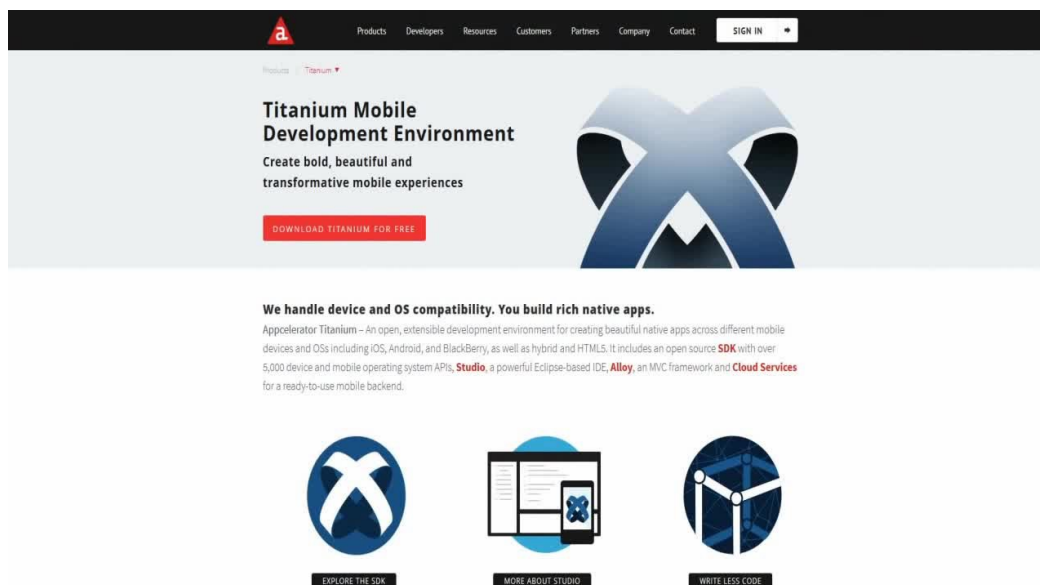


Immagine tratta dal sito web di Appcelerator, dal quale è possibile scaricare Titanium

Questa fondamentale caratteristica lo rende perfetto per chi conosce già, appunto, questi tre linguaggi del web, e magari non ha tempo – o non ha voglia – di addentrarsi in uno dei **linguaggi mobile**, infatti ci penserà **Titanium** a compilare il progetto per le varie piattaforme di destinazione.



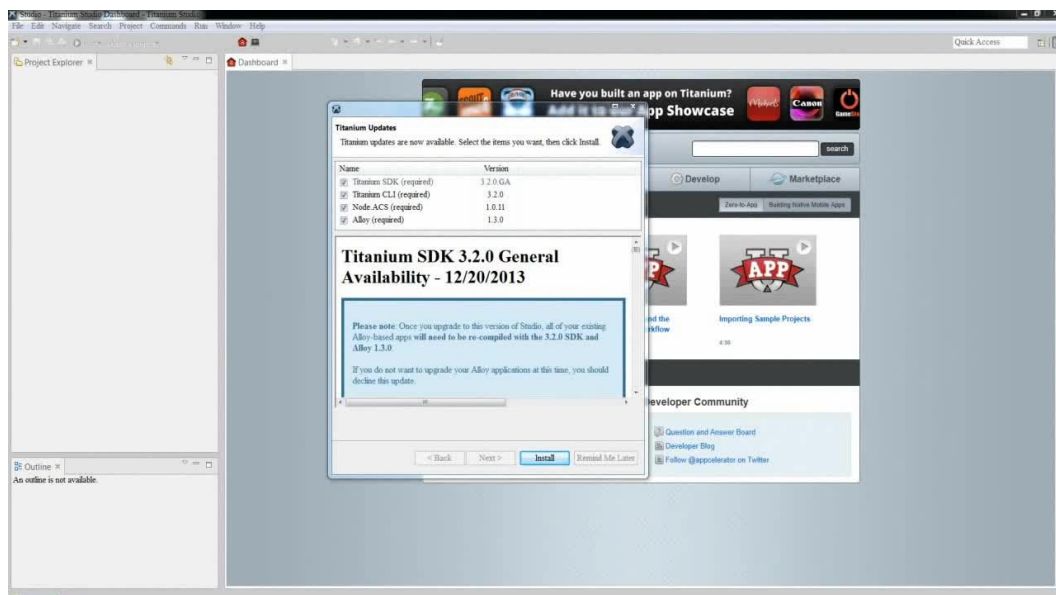
Immagin tratta dalla sezione "Titanium" del sito web di Appcelerator

Questa serie di tutorial è quindi rivolta **a chi ha già una conoscenza di base di XML, CSS e Javascript**; non è un corso di programmazione da zero, ma una serie sulla **creazione di app con Titanium**.

Non sono richieste, quindi, conoscenze dei linguaggi per **Android** e **iOS**, ma è necessario scaricare e installare – oltre al pacchetto **Appcelerator Titanium**, ovviamente – il kit di sviluppo **Java** di Oracle (32 bit per sistemi Windows, anche se si tratta di Win a 64 bit), il kit di sviluppo (SDK, System Development Kit) **Android** e **iOS** --- e su questo punto va detto che, almeno per il momento, **la compilazione per dispositivi iOS può avvenire solo su sistemi Mac**; al momento, ci sono soluzioni non ufficiali per compilare per iOS da sistemi Win o Linux... in futuro, vedremo!

Altra nota, prima di iniziare: questo **non è un corso COMPLETO su Titanium**, ma vuole essere piuttosto un “punto di ingresso” alla conoscenza e allo studio – che potrete approfondire consultando le API del linguaggio o altri tutorial – di questo ambiente; in pratica, è rivolto a chi ha conoscenze di base di programmazione e vuole imparare a muovere i primi passi in tale ambiente.

La **versione di Titanium** utilizzata in questa serie di tutorial è la **3.2**.

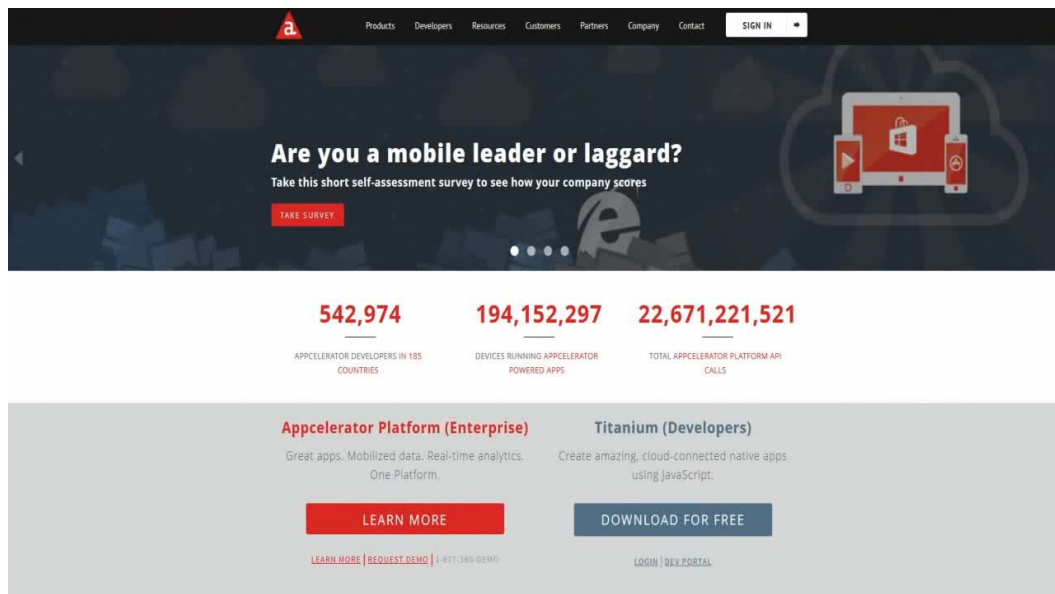


Come si presenta Titanium, una volta effettuata l'installazione del programma

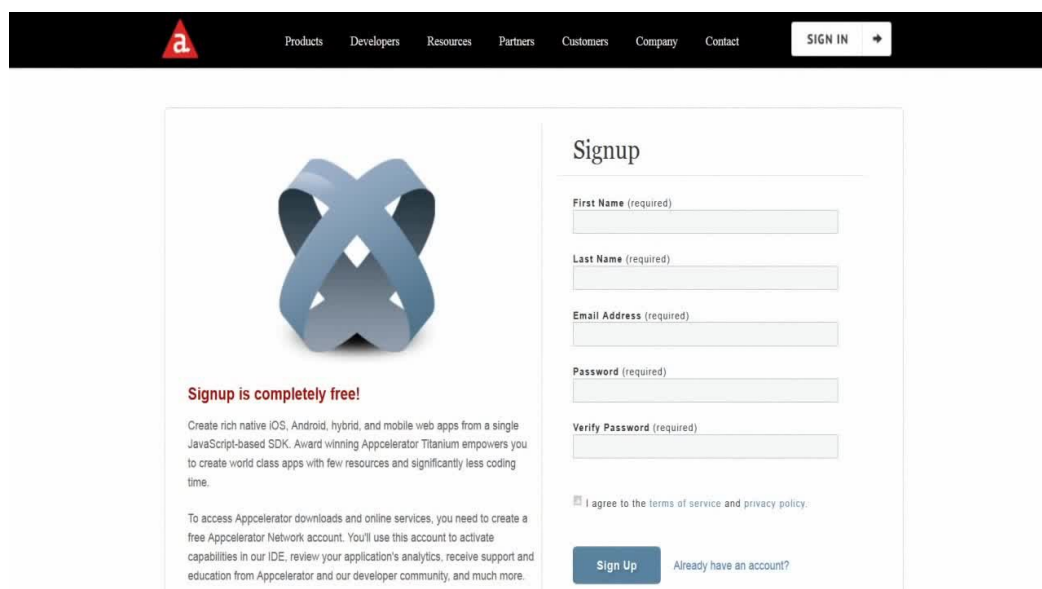
Vediamo quindi come recuperare e installare ciò che ci serve (nel mio caso, **Titanium** e il kit di sviluppo per **Android**; il kit **Java JDK** è scaricabile e installabile dal sito **Oracle**).

Raccolta di tutorial **“Titanium #1: le basi”** dal sito *francescomilanese.com* --- © 2015

Il download di **Titanium** avviene da *www.appcelerator.com* ; per scaricare la versione **Developers**, gratuita, è necessario **registrarsi**, comunque non sono richiesti dati sensibili, per cui sbrighiamo questa pratica (ci verrà inviata un'email con un link da cliccare, per l'attivazione dell'account) e passiamo al download e all'installazione di **Titanium** sul nostro sistema da **“Prodotti – Titanium Studio”**.

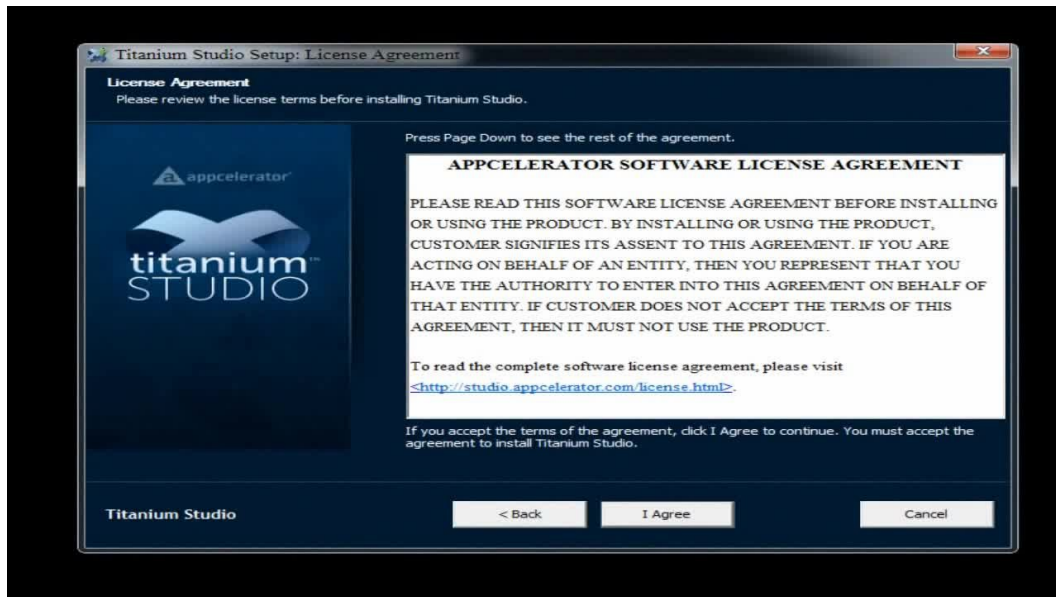


Procedere con "Download for free", sotto "Titanium", dal sito ufficiale Appcelerator



Registrazione (necessaria per poter scaricare e installare Titanium) sul sito Appcelerator

L'installazione avviene semplicemente confermando i vari passaggi.

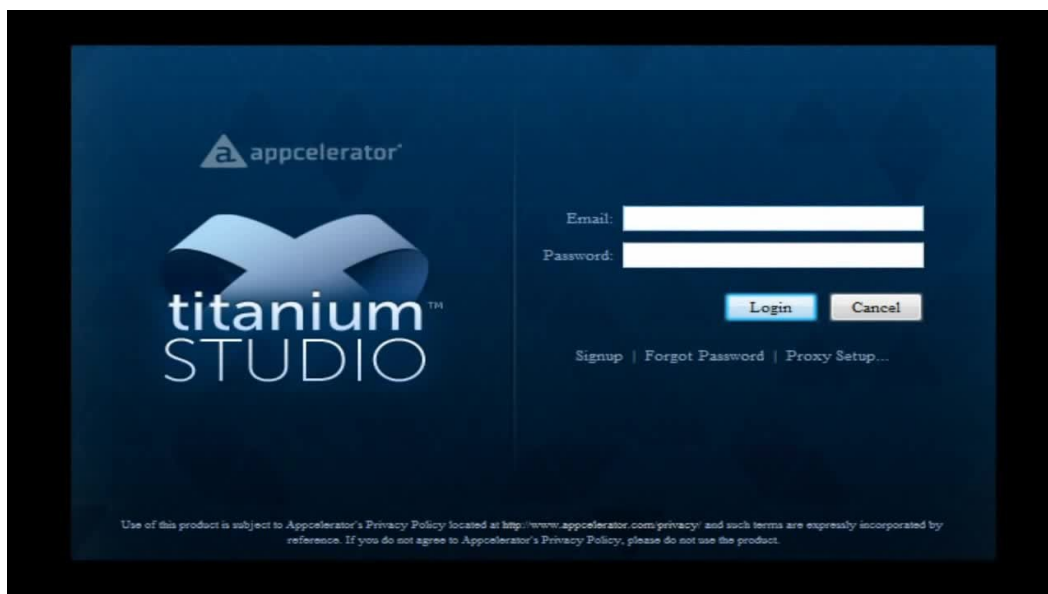


Installato **Titanium**, eseguiamolo!

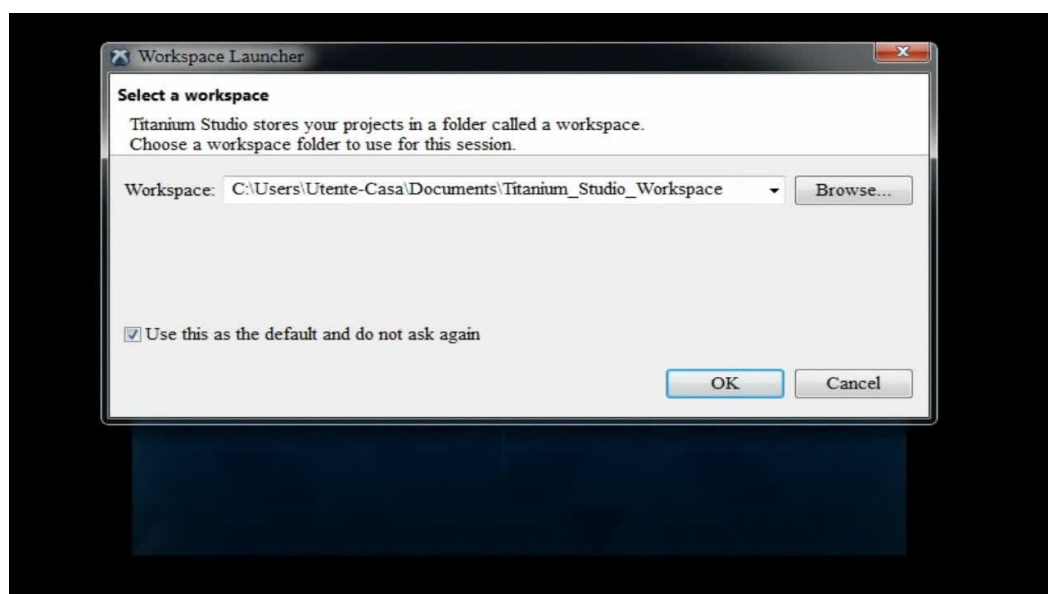


Per avviare l'ambiente dovremo inserire le credenziali del nostro **account** (quello **registrato** un attimo fa, prima di scaricare il pacchetto).

Titanium studio richiede quindi una **connessione web** per l'autenticazione presso l'**Appcelerator Cloud**.



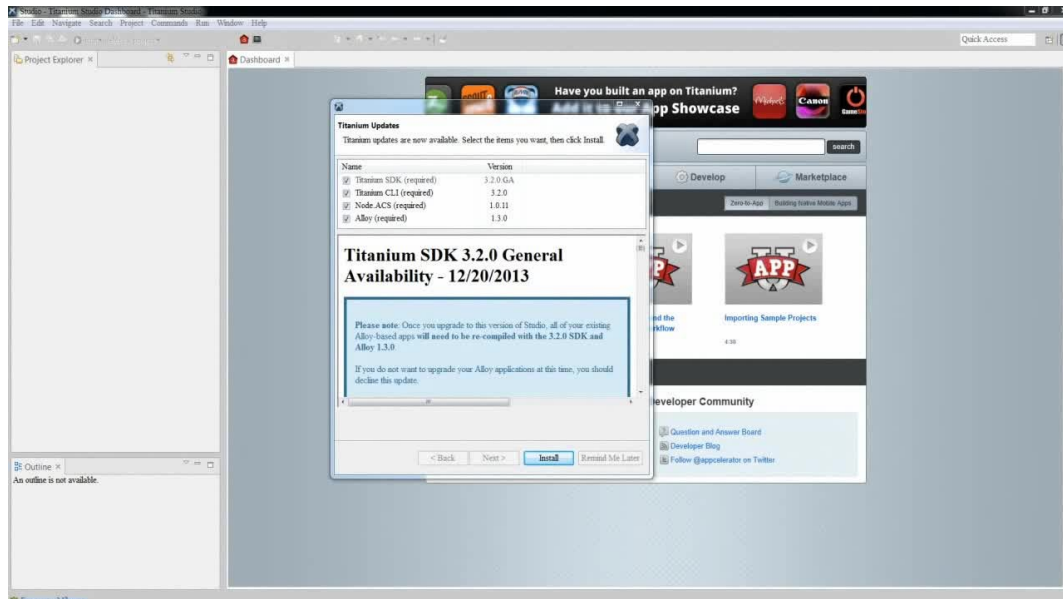
All'avvio ci verrà chiesto di specificare il **Workspace**, ovvero il **percorso su disco** ove **Titanium** salverà le cartelle dei nostri progetti; possiamo lasciare tranquillamente quello di default, magari selezionando la casella *“Use this as the default”*, per non doverlo confermare ogni volta, o specificarne uno diverso... come preferite.



Attenzione però, una nota importante per gli utenti **Windows** (come me): se il percorso del **Workspace** fa riferimento ad una **cartella protetta**, sarà necessario **avviare Titanium come Amministratore**, con click destro sull'icona del collegamento e click su *“Esegui come Amministratore”*, altrimenti il programma non sarà in grado di creare cartelle e, in generale, scrivere

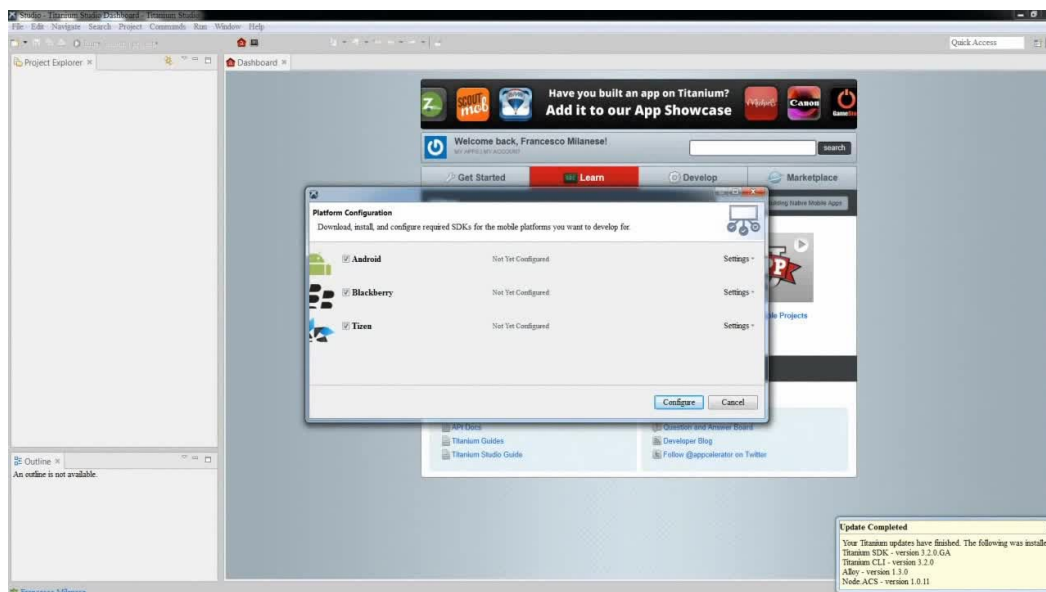
su disco; in alternativa, potete creare una cartella di **Workspace** sul **Desktop** e mettere lì i progetti, in un percorso per così dire “tranquillo”, senza problemi di autorizzazioni, così da non dover eseguire come *Amministratore* ogni volta.

Procediamo quindi con l'installazione degli **aggiornamenti**, se **Titanium** ce ne indica qualcuno (come nel mio caso).

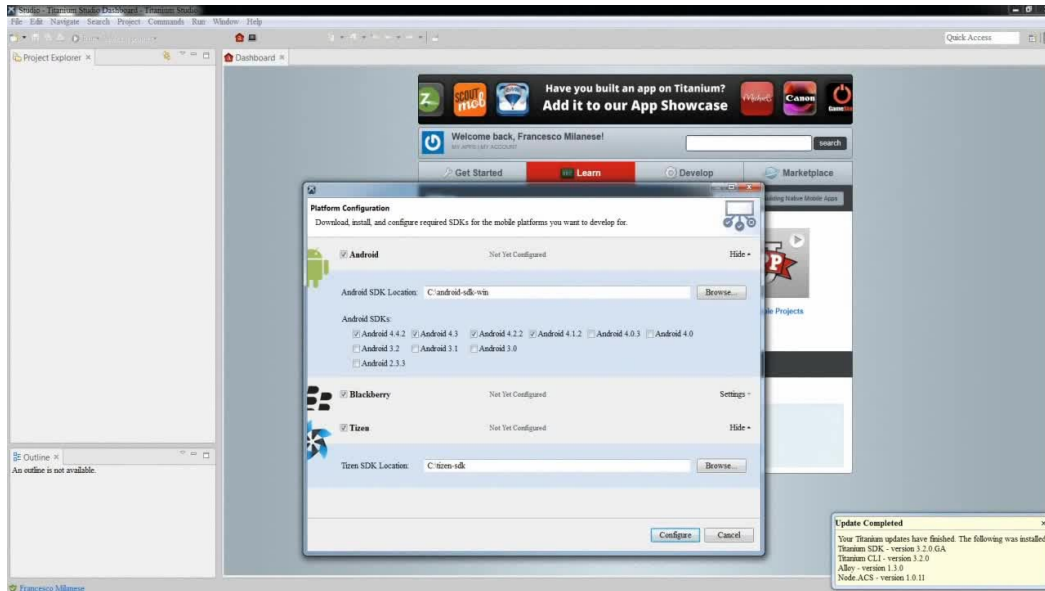


Schermata che indica che sono disponibili degli aggiornamenti, da scaricare e installare, per Titanium o qualche suo componente

Al primo avvio del programma apparirà una finestra di configurazione della **piattaforma di destinazione (Platform Configuration)**, con indicati gli SDK presenti sul dispositivo.



Non resta quindi che scegliere l'**SDK** che ci interessa (ad esempio **Android**), specificando eventualmente versioni e percorso di installazione nella sezione *Settings* e far fare a **Titanium** download e installazione del kit di sviluppo.



Possiamo verificare l'esistenza di **versioni più recenti** degli strumenti da **“Help – Check for Titanium SDK Updates”** ed eventualmente procedere con l'installazione guidata degli stessi.

Ok, finalmente abbiamo tutto quello che ci serve per sviluppare le nostre App!

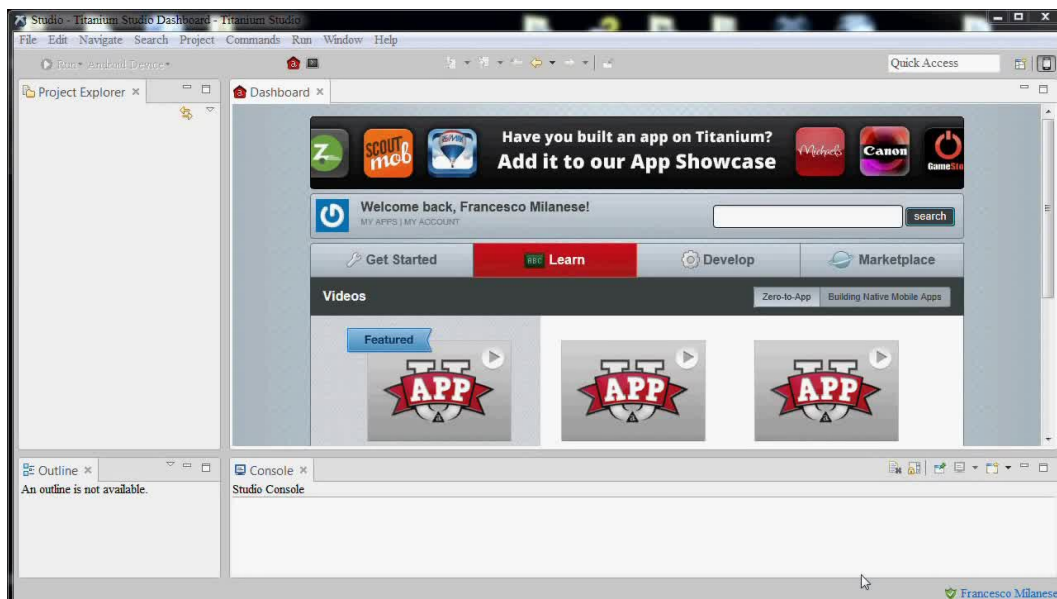
Questa “lezione 0”, dedicata alla presentazione, al download e all'installazione di quello che ci serve per sviluppare, si conclude qui; nella prossima realizzeremo il nostro primo progetto, il nostro **“Hello World!” in Titanium.**

* * *

Titanium - Tutorial 1

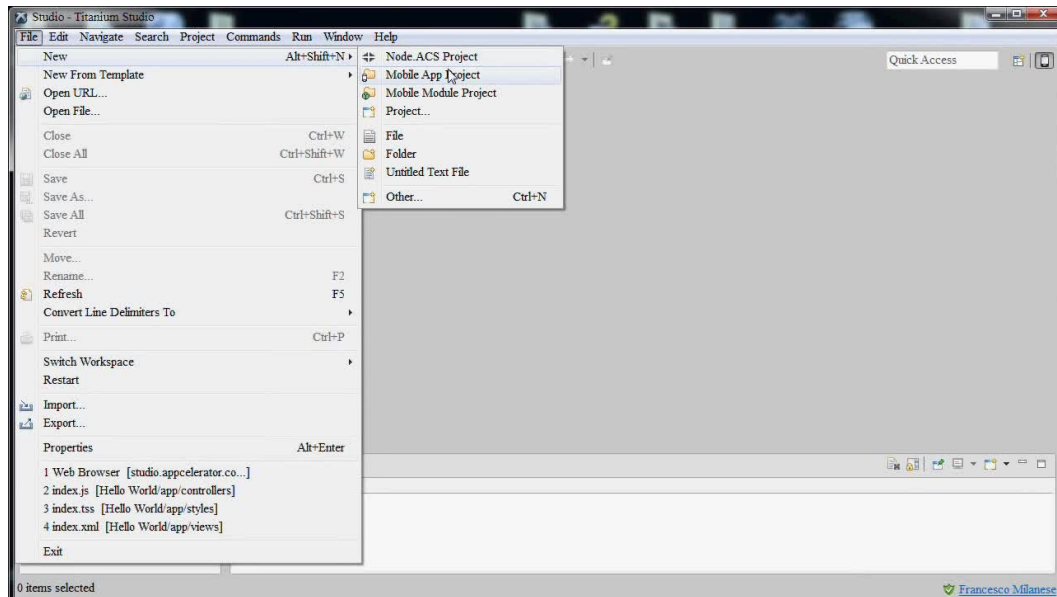
“Hello World”. Template, emulatori. Schema MVC

In questo tutorial (il primo “vero” tutorial, dopo il capitolo precedente dedicato all'installazione del framework) vedremo come creare un progetto, il nostro **“Hello World”**, in **Titanium**.

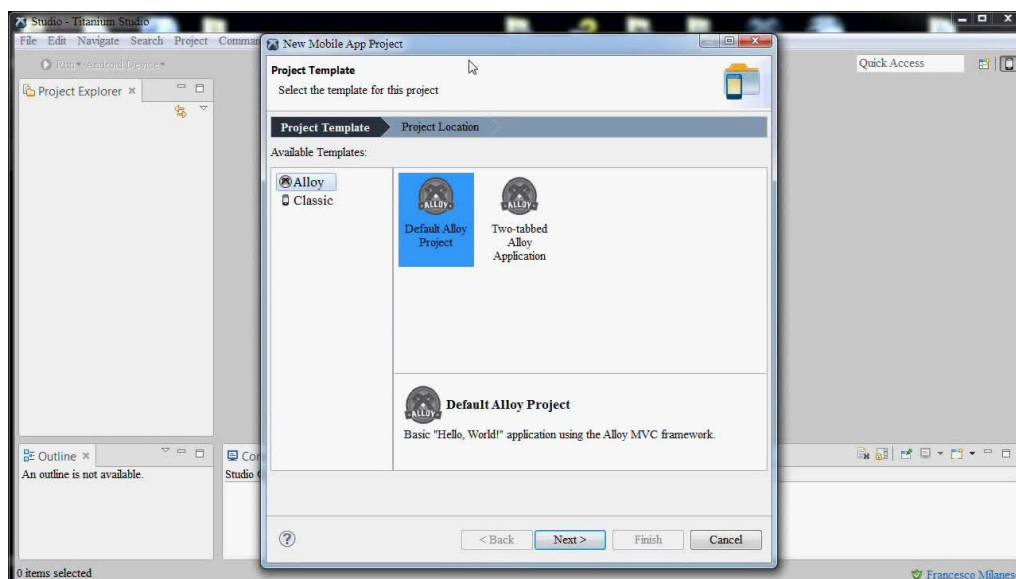


Come si presenta la schermata principale di Titanium Studio, una volta avviato il programma

Dal menù **File** scegliamo **New – Mobile App Project** (shortcut CTRL N o CMD N).



Adesso dobbiamo specificare il tipo di **Template** (il “modello” di progetto che vogliamo); la scelta è tra **Alloy** e **Classic**.

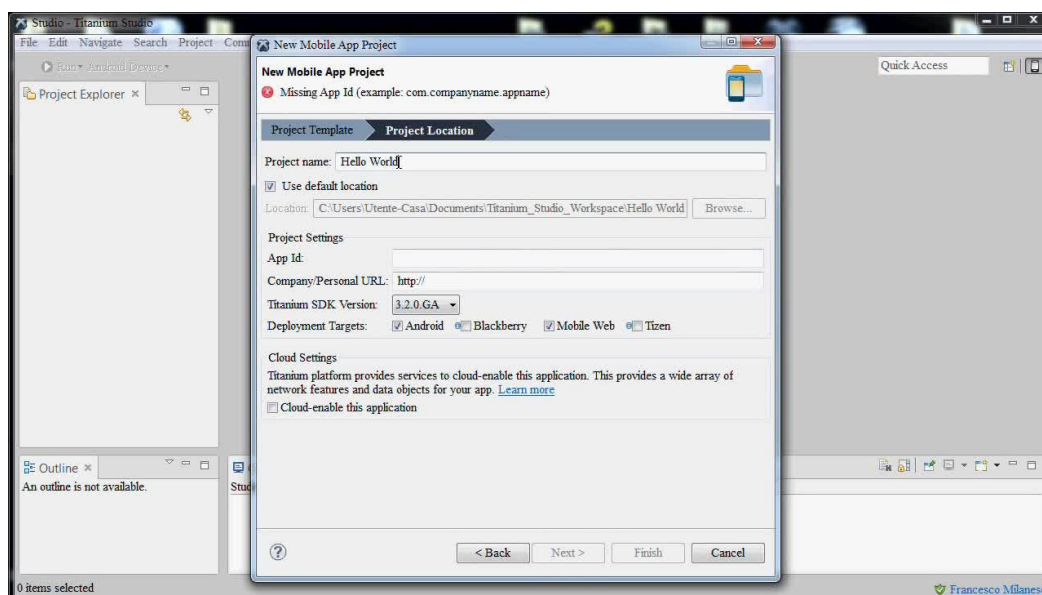


Alloy consente uno sviluppo più veloce e, in generale, semplice, con meno codice da scrivere, in quanto struttura il progetto secondo lo **schema MVC (Model View Controller)**, ossia separando il **Modello** (che fornisce i metodi per accedere a dati e funzioni dell'applicazione), la **Vista** (che visualizza, appunto, i dati del modello, mostrandoli all'utente) e il **Controllo** (che riceve e gestisce le interazioni dell'utente, agendo quindi sia sul Modello – modificando i dati – che sulla Vista, aggiornando la presentazione degli elementi).

Classic crea un progetto con schema... classico, più incentrato sulla programmazione di tutti gli elementi.

Scegliamo il modello **Alloy**, in particolare **Default Alloy Project**.

Specifichiamo quindi, in **Project Location**, alcune informazioni sul progetto che vogliamo sviluppare, come il **Nome** (ad esempio, Hello, da “Hello World”), l'**App ID** (identificativo dell'applicazione) e i **dispositivi o piattaforme di destinazione** (*Deployment Targets*).

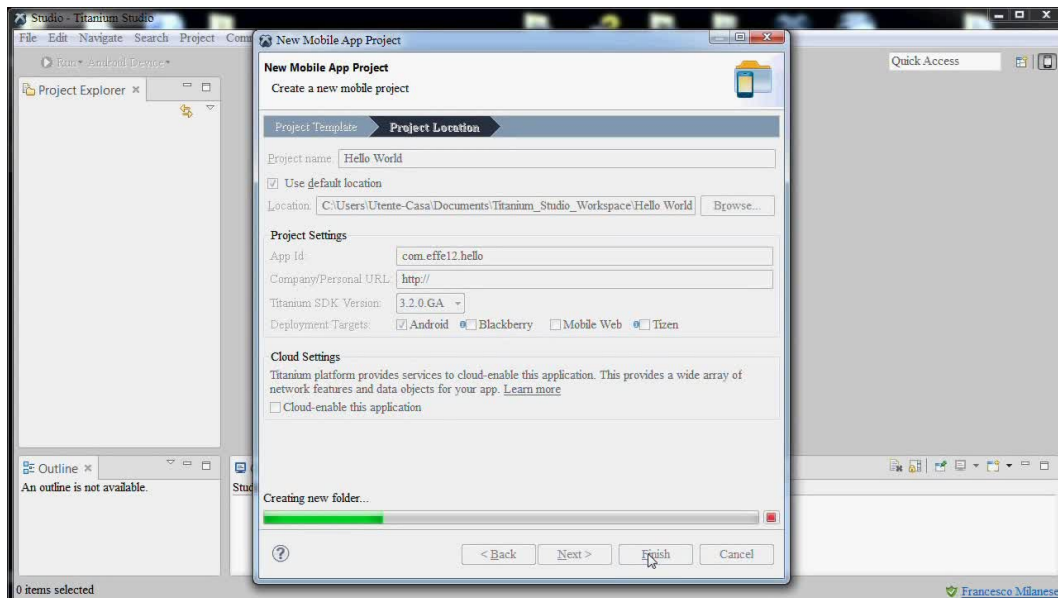


L'**App ID** è una stringa, un identificativo: il nome della vostra applicazione per i dispositivi che la eseguiranno. **Va scritto nella forma *com.publisher.nomeprogetto***, ad esempio *com.effe12.hello*.

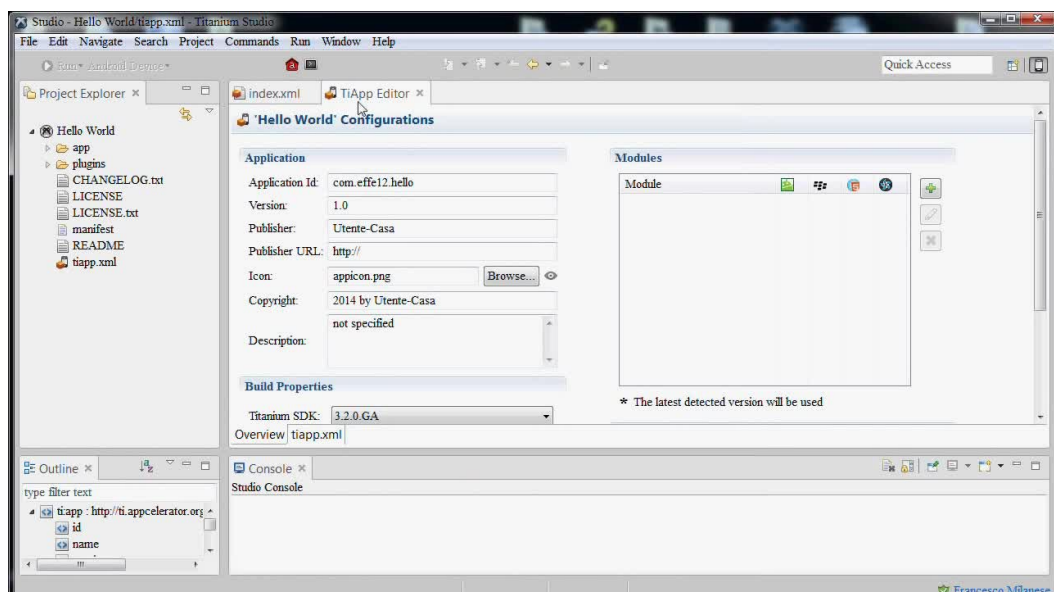
Specificate quindi l'App ID digitando come campo *Publisher* il vostro nome, un vostro nickname o il nome del vostro gruppo o società.

Per quanto riguarda i **dispositivi di destinazione** sto selezionando *Android*; non ho a disposizione iPad e iPhone in quanto sto utilizzando un sistema Win.

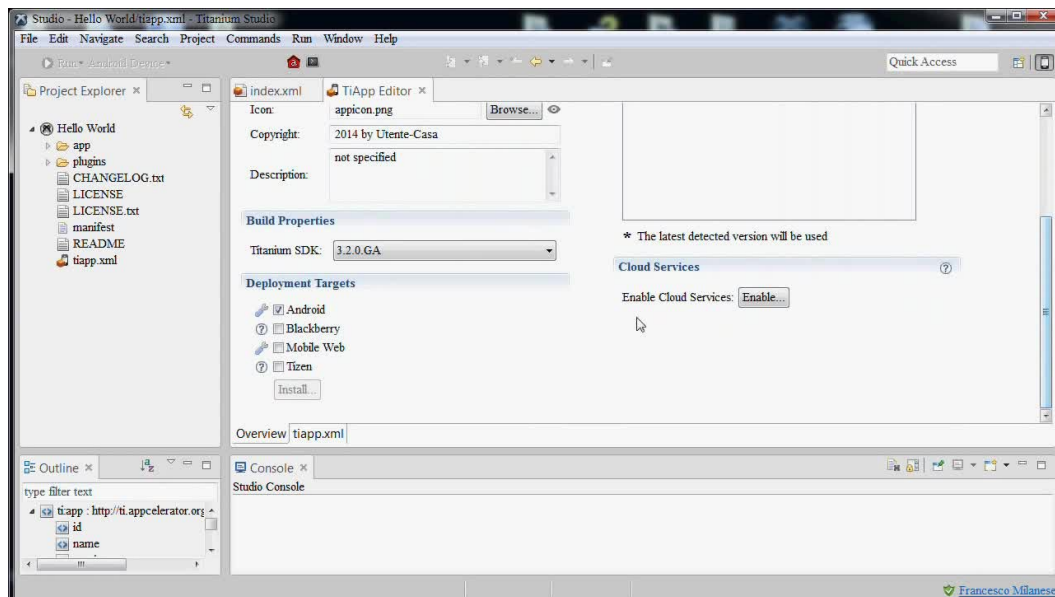
Cliccando su **Finish** avvieremo la creazione del progetto da parte di **Titanium**, che creerà diverse cartelle e file di default per consentire l'avvio di almeno una vista con **un'etichetta (Label)** con contenuto **“Hello World”**.



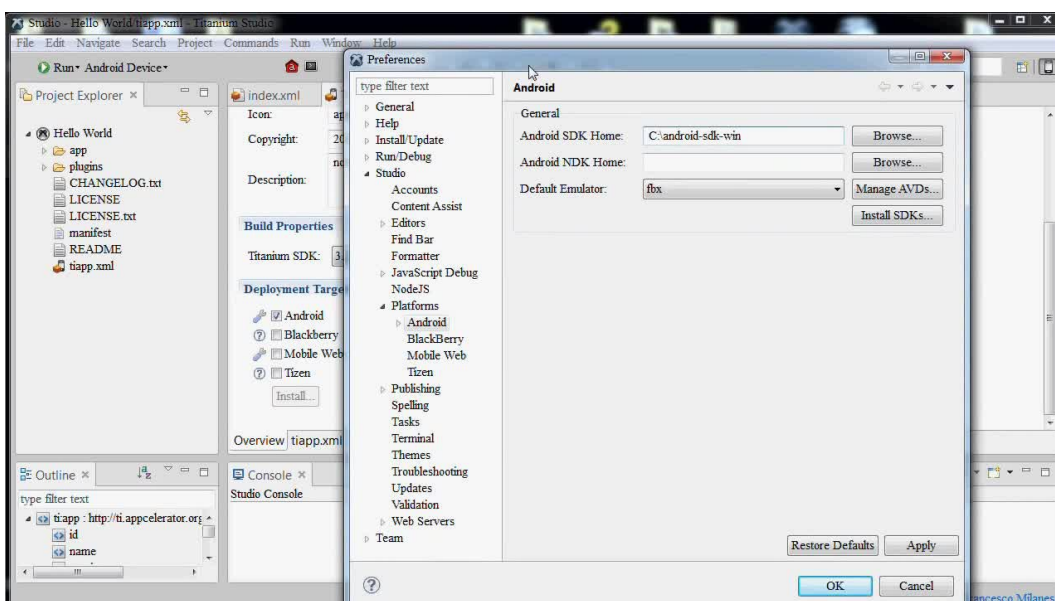
La scheda **TiApp Editor** contiene alcune informazioni e impostazioni interessanti, come l'ID, la versione e il Publisher dell'app, oltre alla definizione dell'icona, del Copyright e una descrizione dell'applicazione.



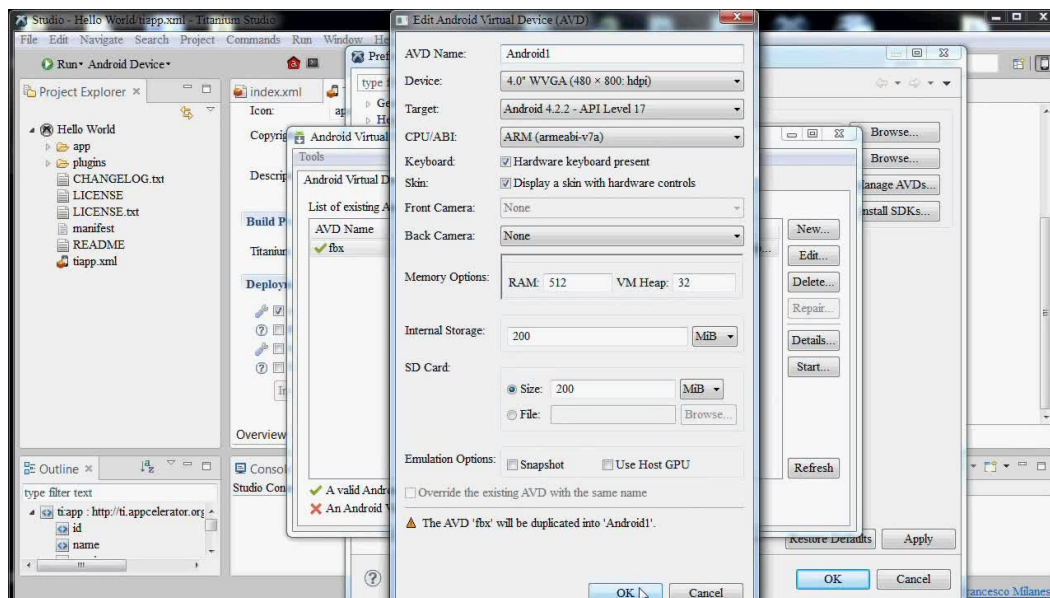
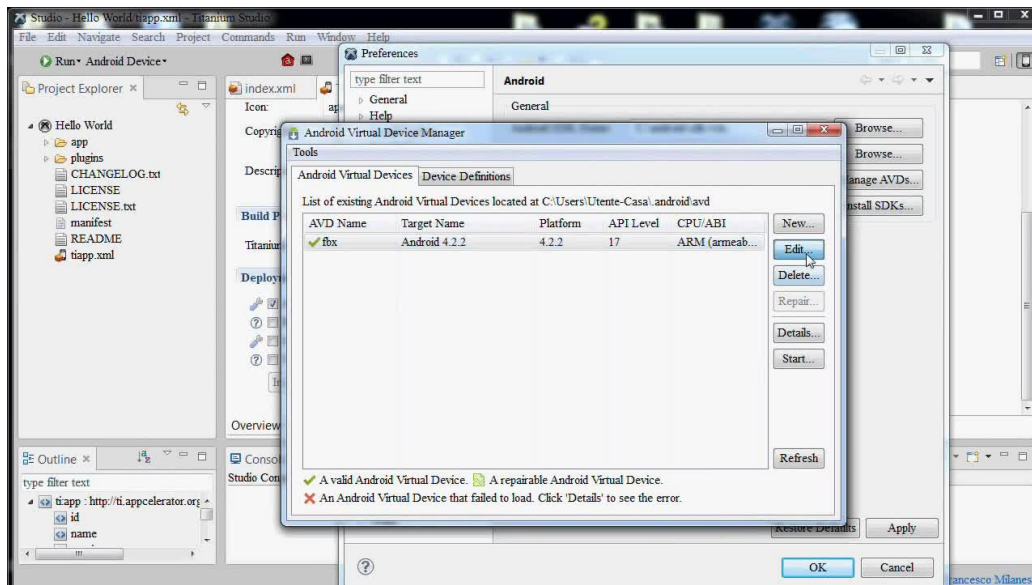
Abbiamo, poi, la sezione dei **Deployment Targets**, dove possiamo specificare le impostazioni degli **emulatori** per i vari dispositivi; qui da me, come detto, non sono visibili iOS e iPad.



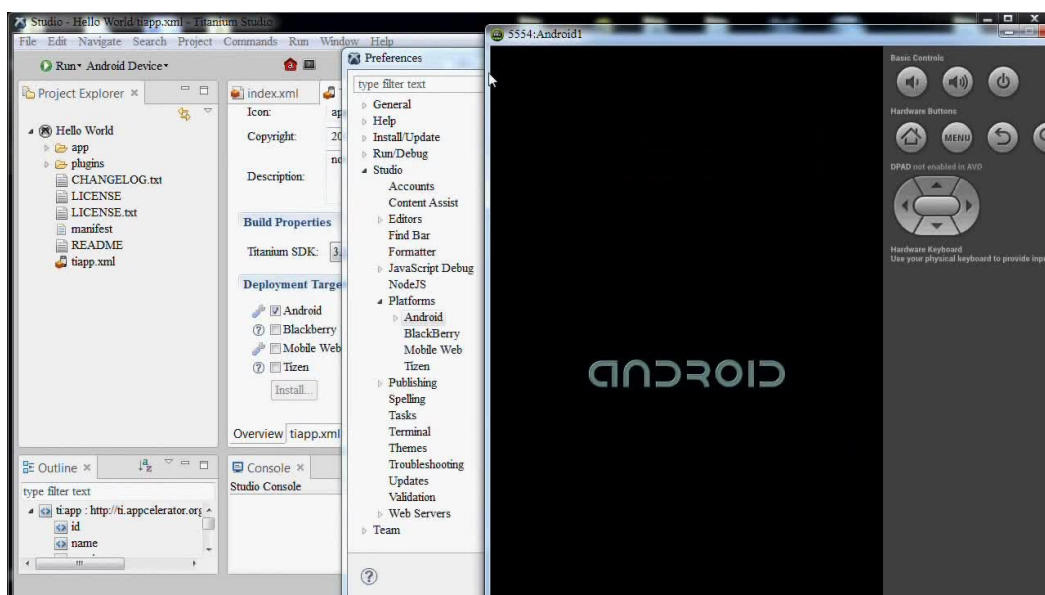
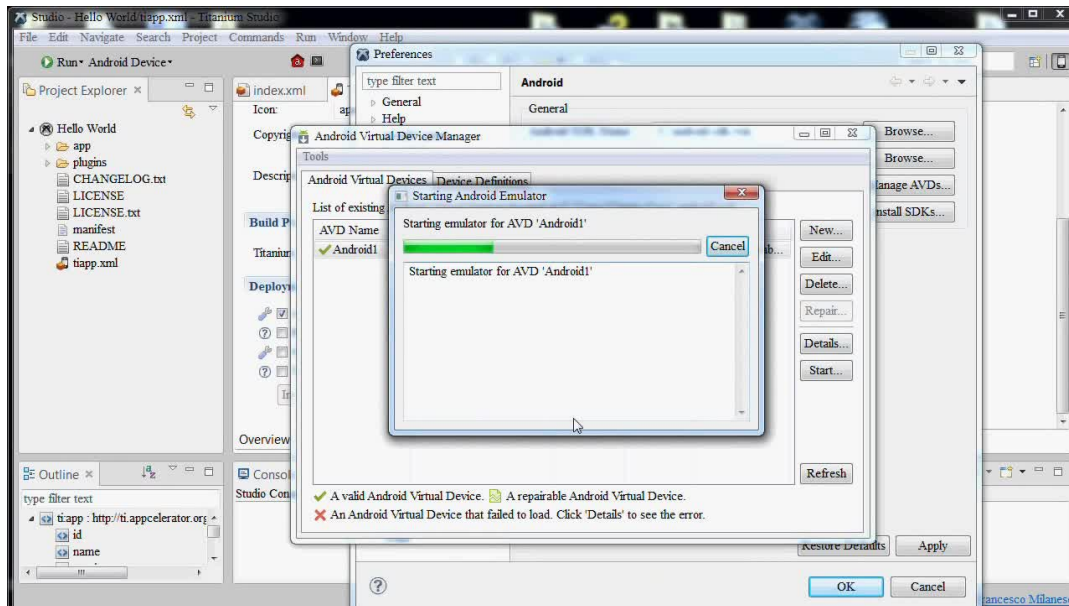
Una cosa da fare subito è **impostare almeno un emulatore** per il dispositivo di destinazione; clicco quindi sull'icona delle impostazioni **Android** (simbolo degli “attrezzi”, a sinistra della checkbox “Android”, nell'immagine precedente) e, nella finestra che appare, mi assicuro che ci sia il **percorso dell'Android SDK** (dovrebbe averlo installato e impostato **Titanium**, come visto nel capitolo precedente), quindi clicco su **Manage AVD**, con “AVD” che sta per “**Android Virtual Device**”: l'emulatore, il dispositivo virtuale **Android**.



Cliccando su **New** possiamo installare anche **più emulatori, con caratteristiche diverse**, per testare le nostre app su devices diversi, ad esempio per risoluzione o grandezza... io ne ho creato uno con la versione 4.2.2 di Android come Target, API Level 17, CPU ARM, risoluzione 480x800 ed emulando 512 MB di RAM e 200 MB di spazio per la scheda SD.



Possiamo verificare il corretto funzionamento dell'emulatore con **Start**, nella scheda dell'AVD.

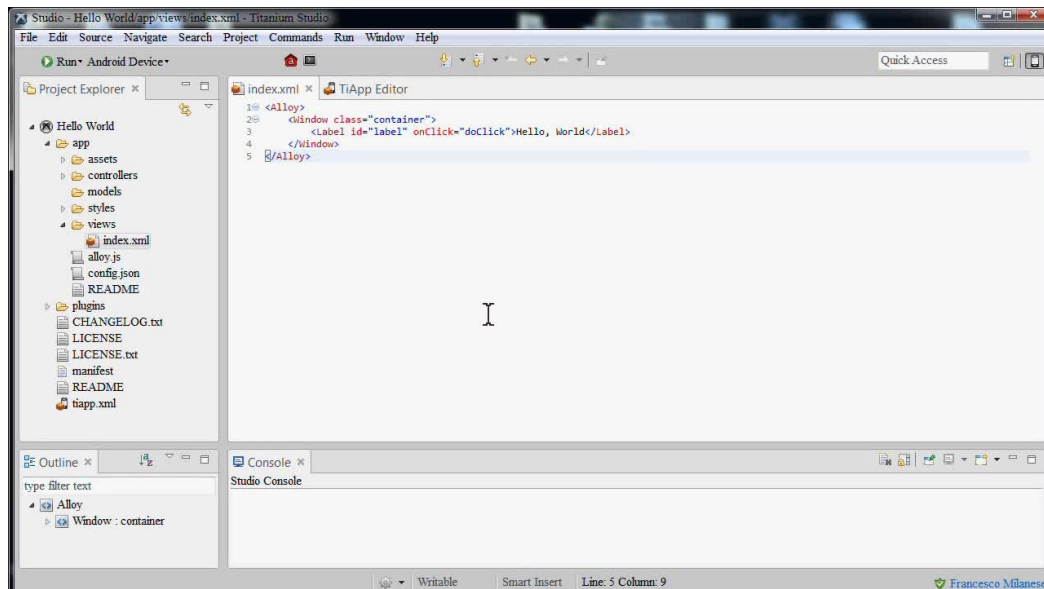


L'emulatore andrà quindi selezionato accanto al **pulsante Run**, o in **Run As**, sopra il **Project Explorer**, per avviare un progetto.

Consiglio di **avviare l'emulatore una sola volta** e di lasciarlo aperto quando si lavora, per fare i test successivi, facendo fare a **Titanium** solo la compilazione del progetto e il passaggio dello stesso al **Virtual Device**, in modo tale cioè da evitare di dover lanciare l'emulatore da zero ogni volta, ma solo la compilazione del progetto... comunque, lo vedremo meglio in seguito con gli esempi pratici.

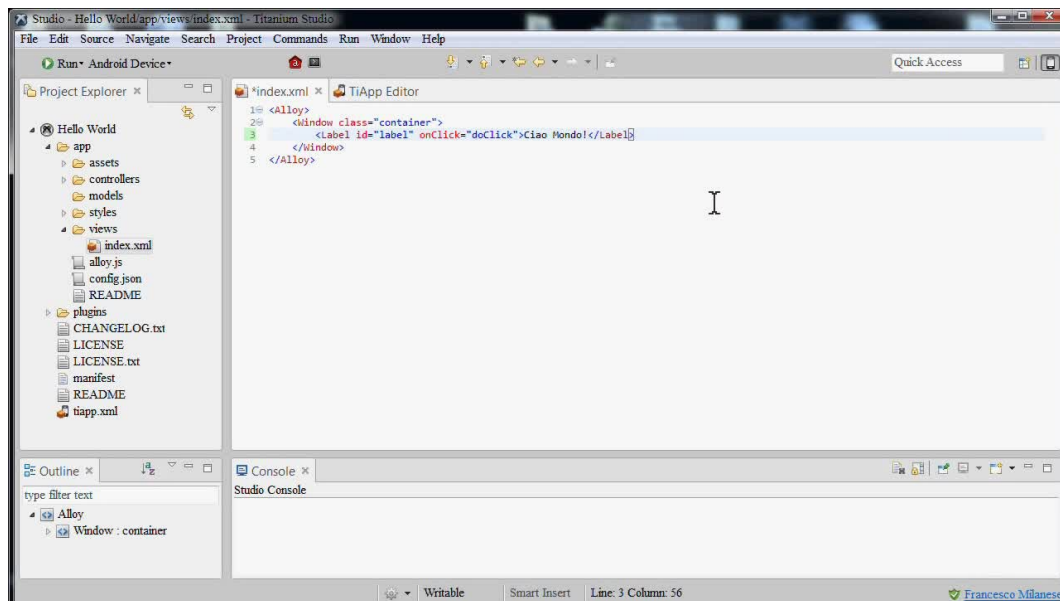
Passiamo quindi ad esaminare i **file di codice** che compongono la nostra **App**.

L'aspetto visivo della vista iniziale, in particolare, viene definito nel file ***index.xml***, che si trova nella cartella **Views**: i file ***xml*** qui presenti servono a specificare solo l'aspetto visivo degli elementi (ricordate la distinzione **MVC** fatta poco fa? Qui ci occupiamo della *View*, la visualizzazione).



Per identificare i vari elementi visivi, vengono utilizzati specifici tag ***xml***, che esamineremo nei vari tutorial; in ***index.xml***, comunque, possiamo riconoscere i tag di apertura e chiusura del template ***Alloy***, un tag ***Window*** (il contenuto della vista o schermata) e il tag ***Label*** (un'etichetta testuale, la classica stringa: una scritta a video).

Cambiamo ad esempio la scritta da visualizzare, tra i tag ***Label*** di apertura e chiusura, da **“Hello World”** a **“Ciao Mondo!”**, e salviamo le modifiche.

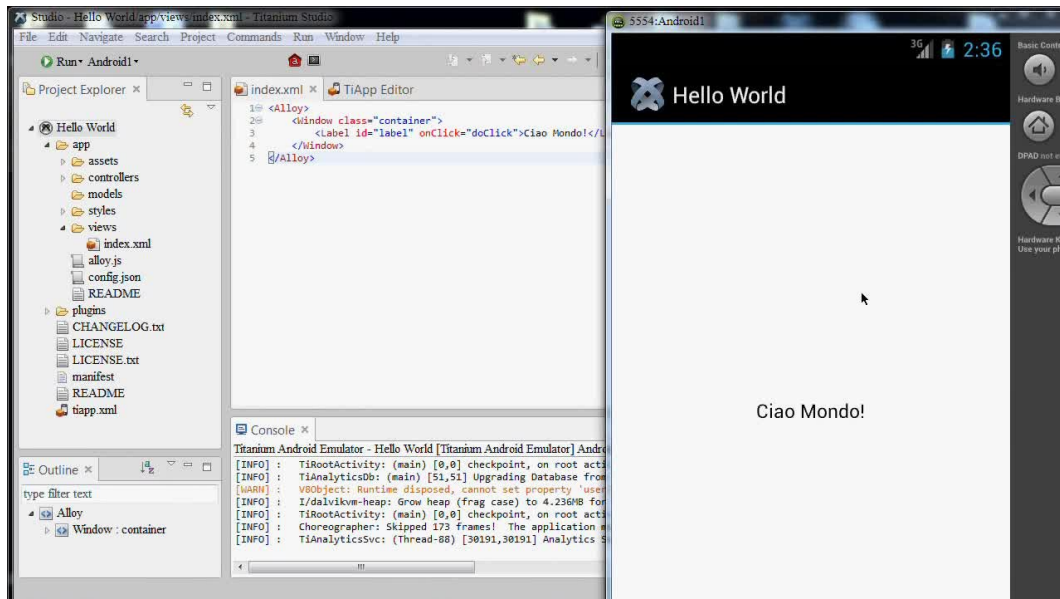


So che avrete notato l'attributo **onClick** con associata la funzione **doClick**, ma per il momento saltiamo questa parte per vedere un esempio di esecuzione di questo progetto.

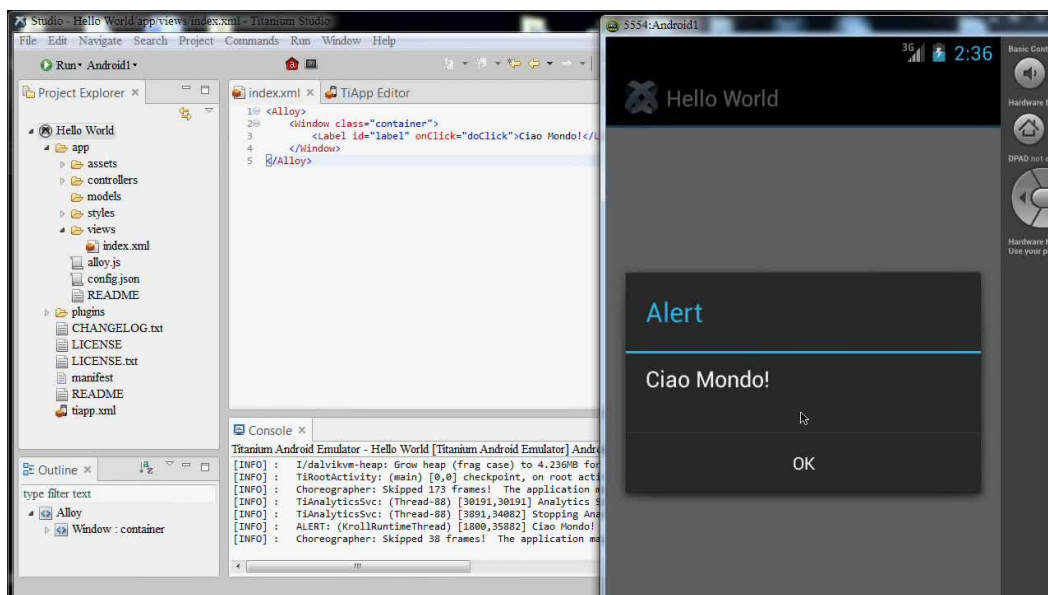
Selezioniamo il progetto nella scheda **Project Explorer**, quindi in alto scegliamo **Run (Esegui)** nella finestra **Launch Mode** e, nella finestra a scomparsa **Target**, selezioniamo il simulatore che ci interessa (nel mio caso, **Android1**).

Facciamo quindi **click destro** sul progetto, nel *Project Explorer*, per scegliere **Run As Android1**.

L'operazione può richiedere un po' di tempo: **Titanium** realizzerà un pacchetto **Android APK**, poi lo passerà all'emulatore.

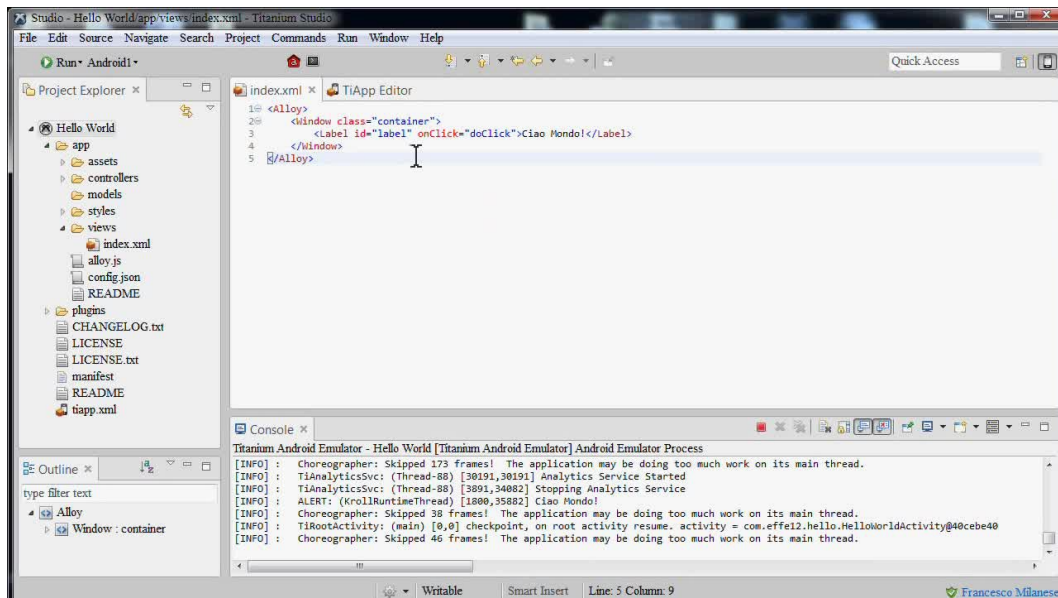


Facciamo click col mouse (simulando, in effetti, il tocco sul display del dispositivo) sulla scritta **“Ciao Mondo!”**: verrà visualizzato a video un **Alert**, per cui in effetti c'è un'azione associata al click – o tocco – sulla **Label “Ciao Mondo!”**.

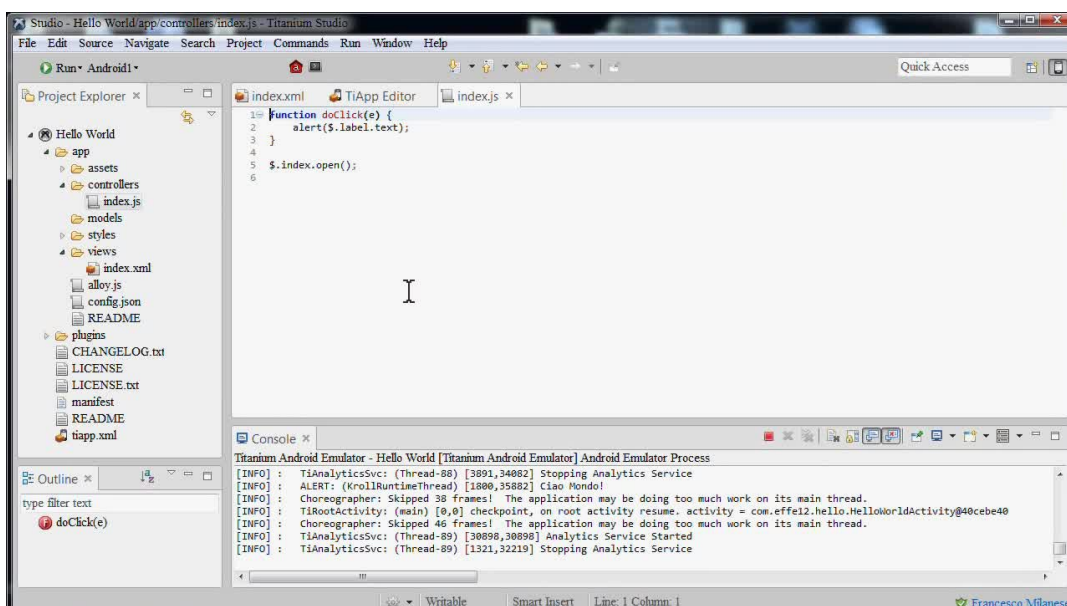


Chiudiamo il simulatore e torniamo al **framework**.

Rivediamo ***index.xml***: nel tag del **Label** abbiamo l'attributo **onClick**, quindi l'evento da generare in caso di click o tocco sulla scritta; in particolare, l'evento associato è la funzione **doClick**, per cui da qualche parte nel progetto dev'esserci **un file di codice** (in linguaggio **Javascript**, come detto nella puntata 0) contenente questa funzione...



Ricordando lo schema **MVC** implementato dal template **Alloy**, nella sezione *Project Explorer* cerchiamo la cartella **Controllers** che, intuitivamente, conterrà gli script e i codici del nostro progetto, quindi apriamola e apriamo il file **index.js** con un doppio click.



Il file contiene, effettivamente, la definizione della **funzione doClick**, composta da una sola riga di codice:

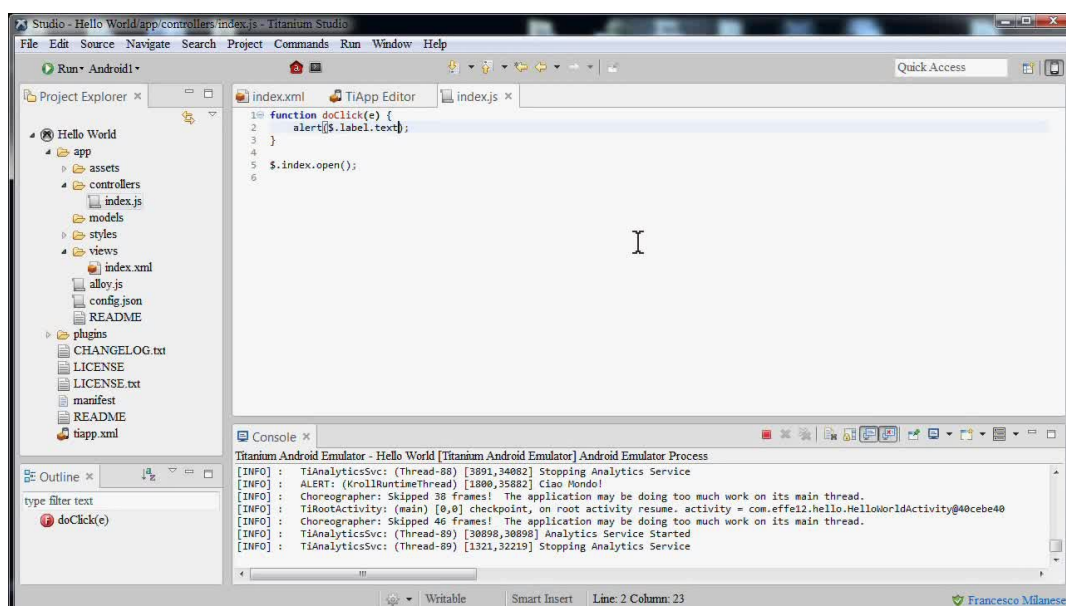
```
alert ($.label.text);
```

ovvero: la chiamata ad **alert**, una **funzione nativa** che mostra a video una finestra pop-up (un messaggio, un “*alert*”, appunto); tale finestra contiene uno o più elementi da mostrare, specificati come argomenti: nel nostro caso, ***\$.label.text***, per cui stiamo accedendo alla **label**, definita in *index.xml*, e di tale elemento stiamo prendendo l'**attributo text**, il testo, ovvero il messaggio “*Ciao Mondo!*”.

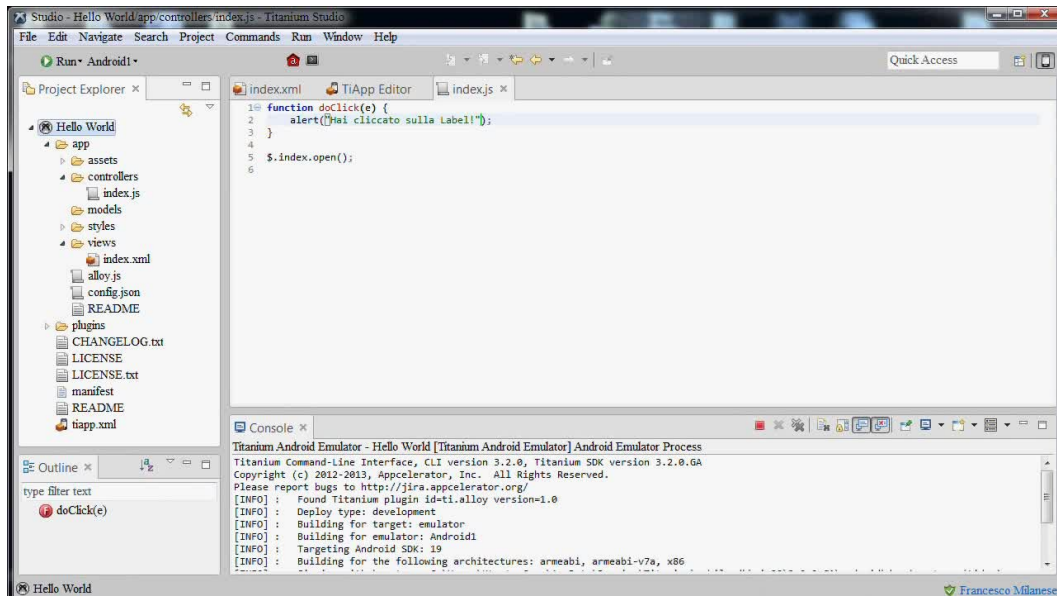
Si può quindi accedere ad **elementi della schermata con \$.**, il nome dell'elemento (nel nostro caso, **label**) e il campo o **parametro** che ci interessa (**text** prende il testo dell'etichetta).

Attenzione: **label** è scritto in minuscolo perché non si tratta del tag label ma, ovviamente, **del nome o identificativo di tale elemento**, che nell'xml abbiamo specificato con “*id=label*”, con la l minuscola, per cui è al **valore di id** che ci riferiamo con ***\$.label***.

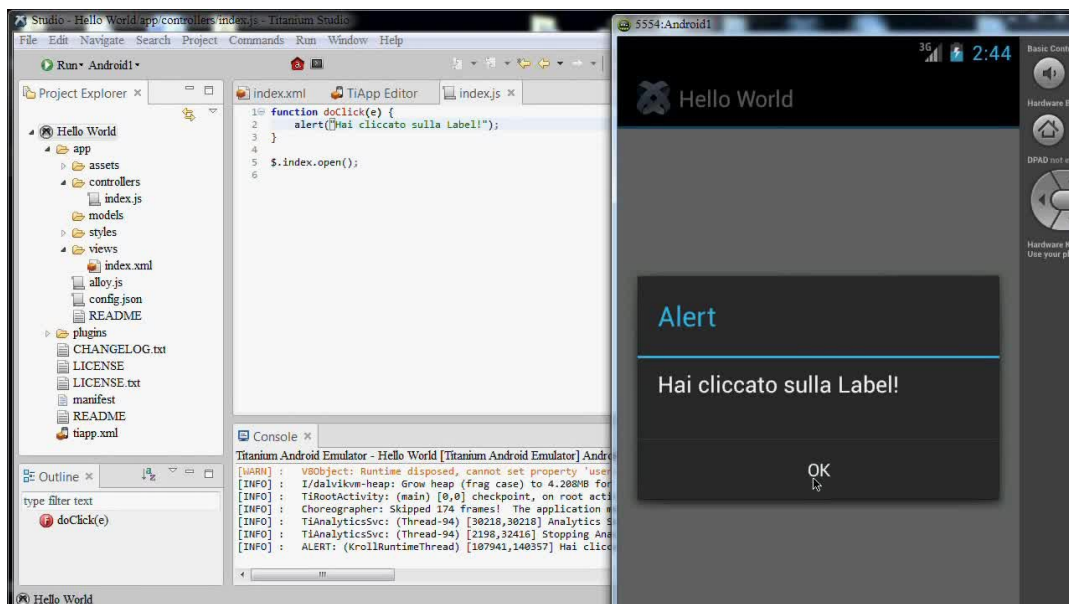
Notiamo inoltre che nel file Javascript è presente un'istruzione finale, ***\$.index.open()***, fuori da qualsiasi blocco funzione e, quindi, eseguita **automaticamente all'avvio** di *index.js* che, intuitivamente, apre la schermata corrente (***\$.index*** accede a questa schermata, mentre la funzione ***open*** la apre).



Proviamo a fare una modifica: sostituiamo `$.label.text` con **“Hai cliccato sulla label”**, mettendo i doppi apici in quanto si tratta di una stringa esplicita, non di una variabile o di un riferimento ad altri elementi.

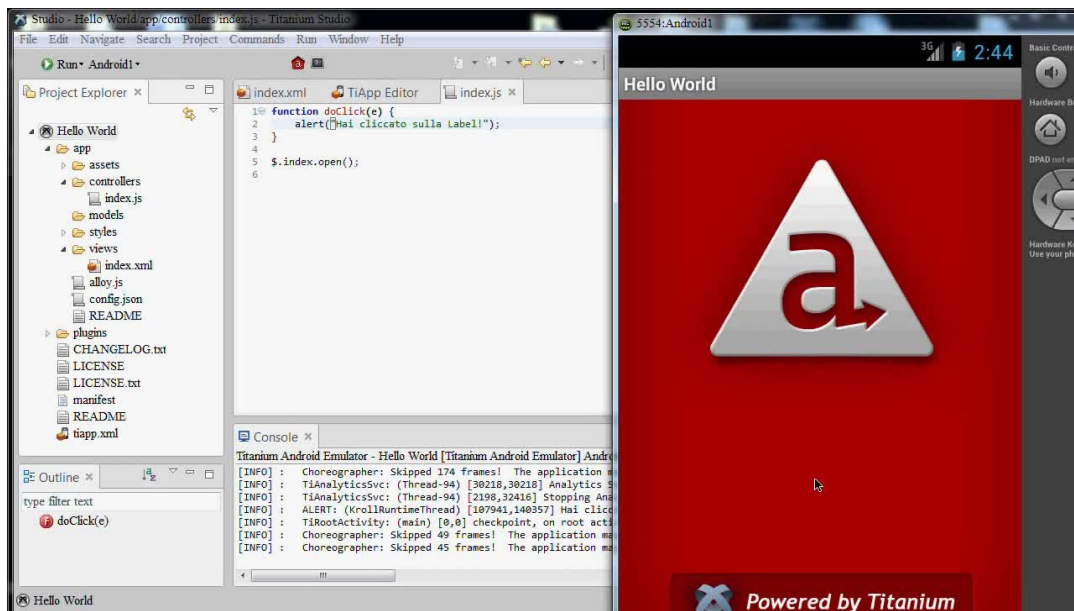


Salviamo ed eseguiamo l'app con **Run As – Android 1**, sul progetto, per osservare le modifiche apportate.



Ricapitolando: utilizzando **Alloy** suddividiamo le risorse del progetto separando l'interfaccia, definita mediante file xml per le varie viste, nella cartella *Views*, dal codice, con i file Javascript

nella cartella *Controllers*. Abbiamo visto come definire e mostrare una **label** a video, associandole una funzione che lancia un **alert** con un messaggio da noi definito.



* * *

Titanium - Tutorial 2

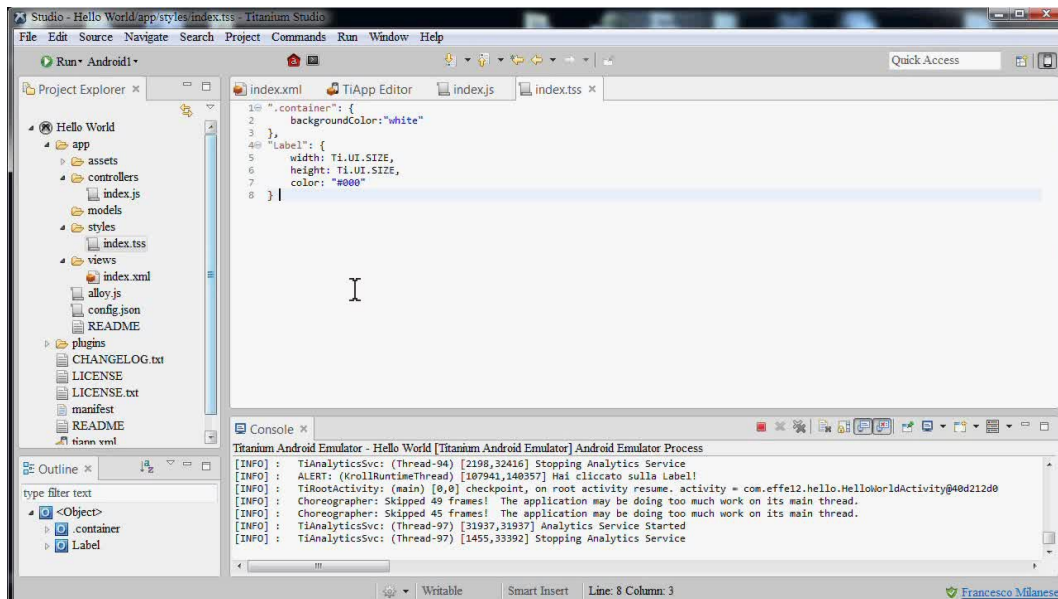
Stili (CSS). Buttons (pulsanti) ed evento "tocco"

In questo tutorial parleremo degli “*Stili*”, ovvero dell'utilizzo dei **CSS** per definire l'aspetto visivo degli elementi in **Titanium**, utilizzando il template **Alloy** per realizzare le nostre app.

Abbiamo visto, infatti, come definire un elemento, una *label*, associandole un evento (la chiamata alla funzione *doClick*) e come definire una funzione per effettuare un'operazione, ma non abbiamo parlato di dimensioni, posizionamento, colore e altri attributi di visualizzazione degli elementi.

Queste informazioni vanno specificate in particolari fogli di stile che, in **Alloy**, vengono detti **TSS** (*Titanium Style Sheet*), situati nella cartella *Styles*.

Apriamo ad esempio *index.tss*, il *Titanium Style Sheet* associato alla prima e unica schermata dell'applicazione vista nel tutorial precedente.



Le informazioni sullo stile di un elemento vengono inserite mediante blocchi così composti: il nome dell'elemento tra doppi apici, due punti e, a seguire, una coppia di parentesi graffe che identificano, appunto, il blocco di caratteristiche dell'elemento.

Tali caratteristiche vengono specificate con: nome caratteristica, due punti e valore della caratteristica, il tutto seguito da virgole, tranne per l'ultima informazione del blocco. Anche i blocchi sono separati da virgole.

Vediamo subito degli esempi pratici.

Il primo blocco fa riferimento all'elemento **“container”**, che è il **contenitore visivo della Vista**, nel senso che tutti gli altri elementi visivi della schermata verranno inseriti in questo elemento.

Il container ha, qui, solo una caratteristica: **backgroundColor** (colore di sfondo), con valore **“white”**, tra doppi apici in quanto stringa esplicita, per cui sappiamo che è qui che stiamo definendo il colore di sfondo della schermata.

Abbiamo poi il blocco relativo all'elemento **label**. Qui troviamo le caratteristiche **width** (larghezza), **height** (altezza) e colore, con le prime due che hanno, associato, un **valore nativo** di **Titanium**: **Ti.UI.SIZE**, ossia le dimensioni minime richieste dall'interfaccia (**UI** sta per *User Interface*), senza doppi apici perché si tratta di veri e propri campi di **Titanium**, mentre il colore è specificato con una stringa che indica il valore esadecimale (#000 è quindi il nero).

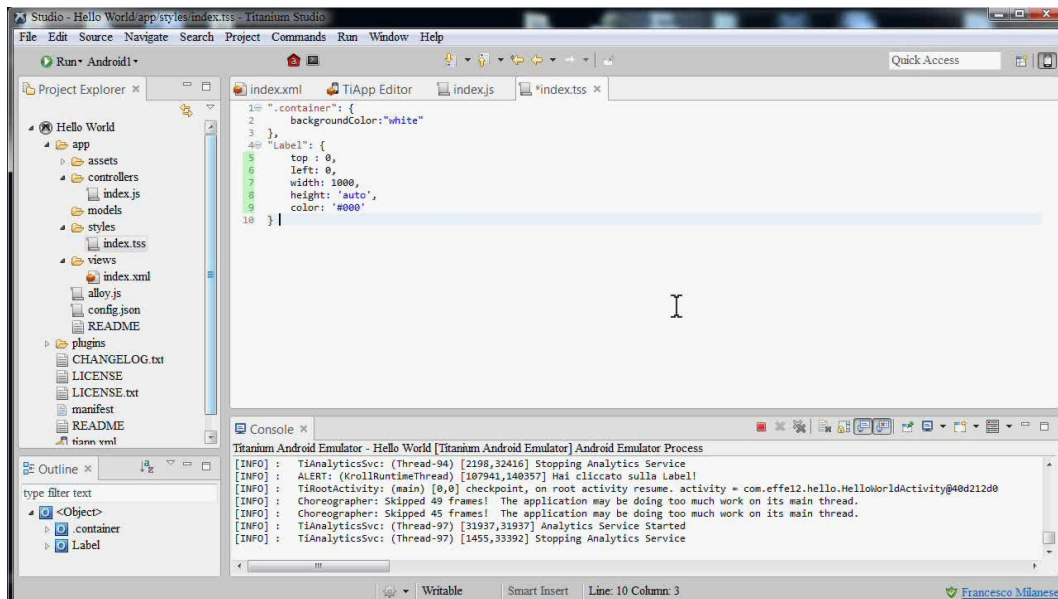
In questo tutorial, per analizzare le interazioni tra elementi visivi e stili, faremo le seguenti operazioni: sposteremo e ridimensioneremo la **label** e, in seguito, creeremo un **pulsante** che, al tocco, cambierà alternativamente il colore di sfondo tra giallo e blu.

Iniziamo dal primo compito, quello più semplice: **spostare e ridimensionare la label**.

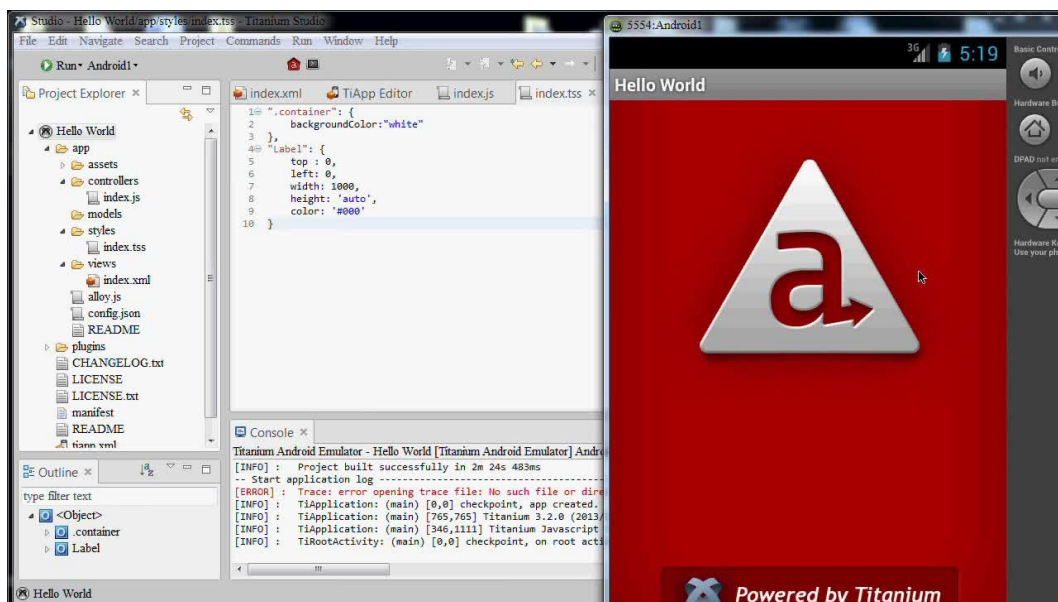
Diciamo di voler posizionare la **label** nell'angolo in alto a sinistra della schermata, dandole come dimensione orizzontale 1000px e come **dimensione verticale AUTO**.

Cambiamo quindi le istruzioni nel blocco Label del TSS come segue:

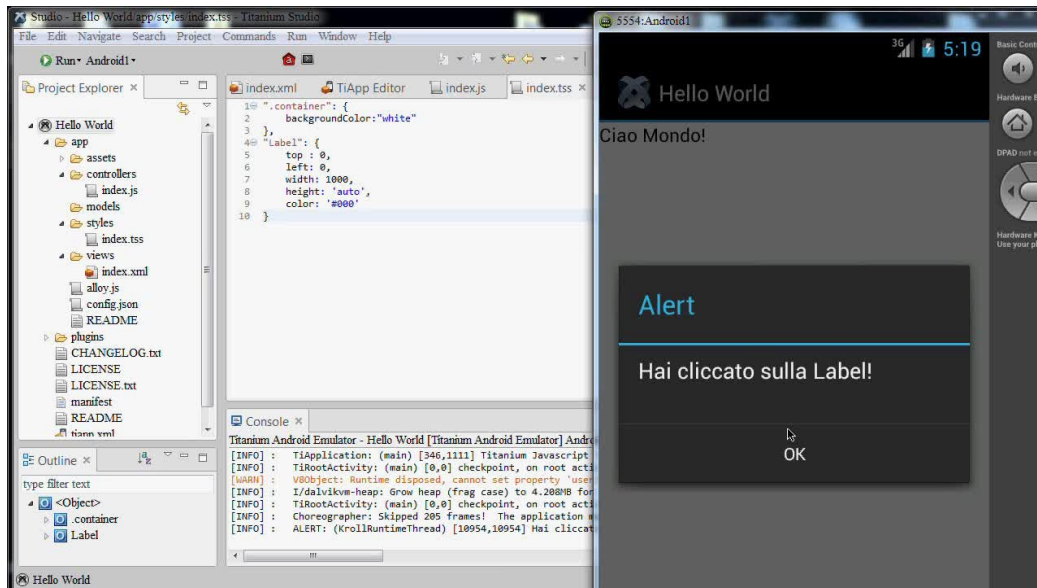
```
"Label": {  
    top : 0,  
    left: 0,  
    width: 1000,  
    height: 'auto',  
    color: '#000'  
}
```



Salviamo e avviamo l'applicazione per esaminare quanto fatto.



L'aver dato una **width** esageratamente grande per la **label** si traduce nel fatto che tale elemento occuperà praticamente tutta la riga sulla quale si trova, nonostante il testo sia relativamente breve, per cui con un click del mouse a destra di **“Ciao Mondo!”** apriremo ugualmente l'*alert*, perché l'area fa parte della label.



Questo non avviene cliccando, invece, **sotto la label**, perché l'altezza, specificata con **'auto'** per adattarsi al testo contenuto, è limitata alla sola riga di testo, quindi **la label non 'sfora' verticalmente**.

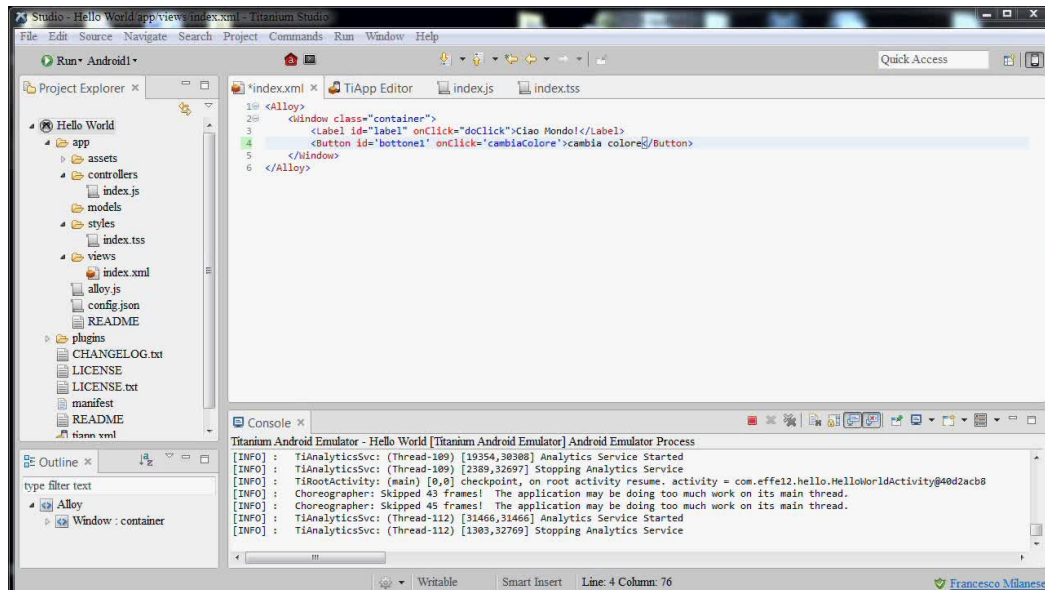
Per evidenziare l'area occupata dalla *label* provate a mettere un **background color** diverso dal bianco (il colore del container) per questo elemento nel **TSS**, salvate e ri-eseguite l'applicazione.

La seconda operazione è decisamente più interessante per almeno due motivi: il primo è che introdurremo un **Button**, un nuovo oggetto dell'interfaccia; il secondo è che modificheremo il valore di un attributo di uno stile TSS da uno script di codice Javascript, per cui vedremo come **accedere ad un campo di un CSS** e come **modificarlo dinamicamente**, durante l'esecuzione dell'app.

Iniziamo aprendo l'**index.xml** per definire il pulsante, scrivendo dopo la **label**:

```
<Button id='bottonel' onClick='cambiaColore'>cambia colore</Button>
```

quindi passiamo a **index.js** per specificare l'evento da eseguire a seguito di un click sul bottone, ossia la funzione **cambiaColore**.



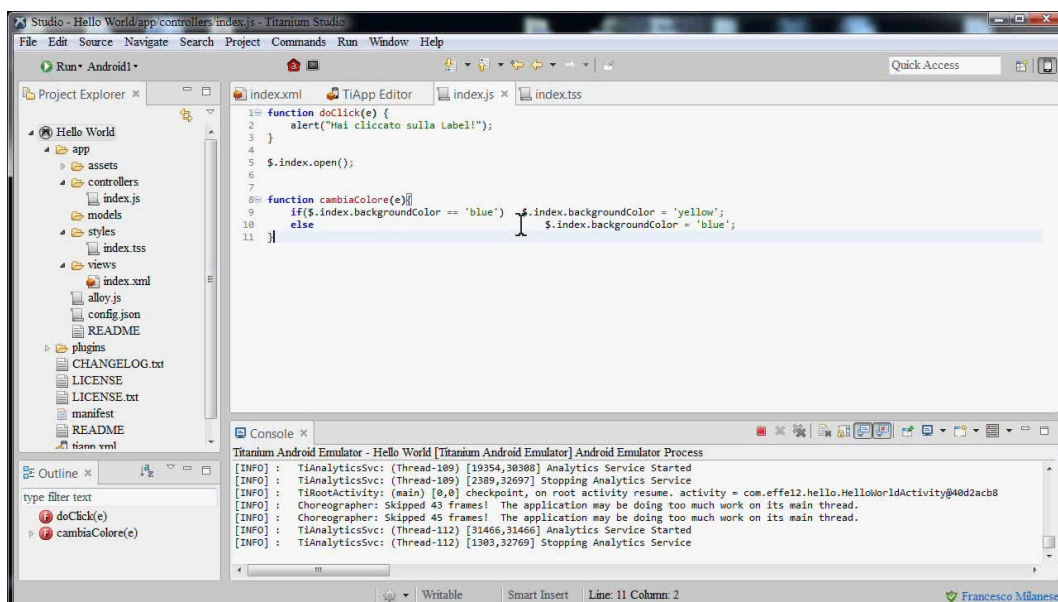
Qui dobbiamo accedere al campo **backgroundColor** della schermata, verificando qual è il valore attuale e modificandolo; lo facciamo scrivendo, come corpo della funzione, le seguenti istruzioni:

```
function cambiaColore(e) {

    if($.index.backgroundColor == 'blue') $.index.backgroundColor =
    'yellow';

    else      $.index.backgroundColor = 'blue';

}
```



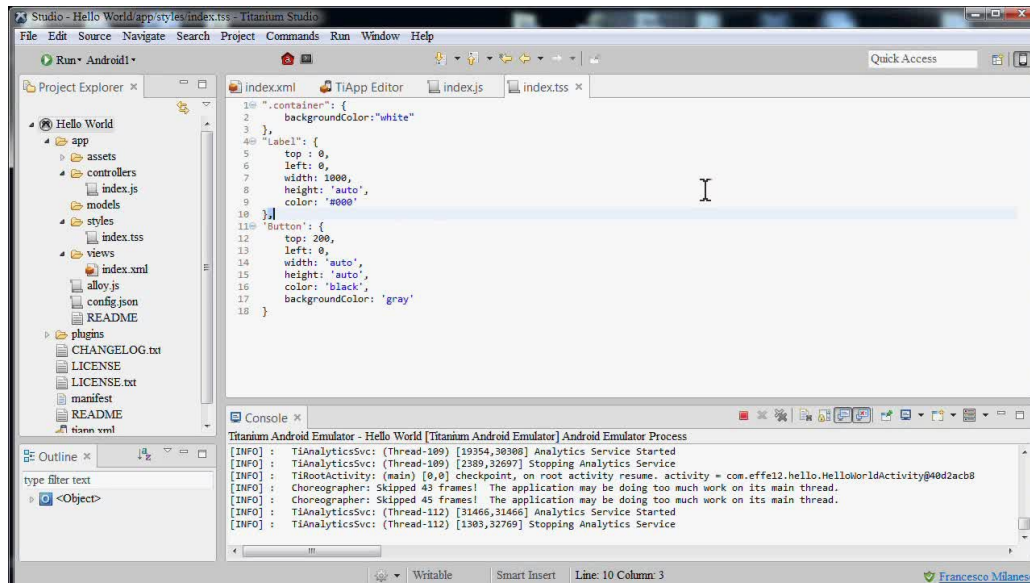
Si accede agli elementi della **View** con **\$**. e questa non è una novità, l'abbiamo visto già nel tutorial precedente (accedendo al **text** della Label di id “*label*” con **\$label**), l'unica cosa da notare è che **non stiamo accedendo a container, ma a index**: il *container* è un elemento di *index*, che è la nostra **View** (la schermata corrente), per cui stiamo impostando il valore per il parametro **backgroundColor** della Vista *index*.

Da notare quindi che al primo avvio la schermata avrà **colore di sfondo bianco**, perché non l'abbiamo modificato nell'*xml*, comunque per via dell'*else* imposteremo, al primo click, il colore blu, quindi nessun problema.

Non abbiamo ancora finito: abbiamo specificato, infatti, la creazione di un **Button**, con id “*bottonel*” e **funzione onClick** associata a “*cambiaColore*”, nell'*xml*; abbiamo specificato poi la **funzione cambiaColore** nel **Javascript**, nella schermata *index*... manca, quindi, **lo stile del Button**.

Nel **TSS di index** scriviamo il seguente blocco:

```
'Button' : {  
    top: 200,  
    left: 0,  
    width: 'auto',  
    height: 'auto',  
    color: 'black',  
    backgroundColor: 'gray'  
}
```

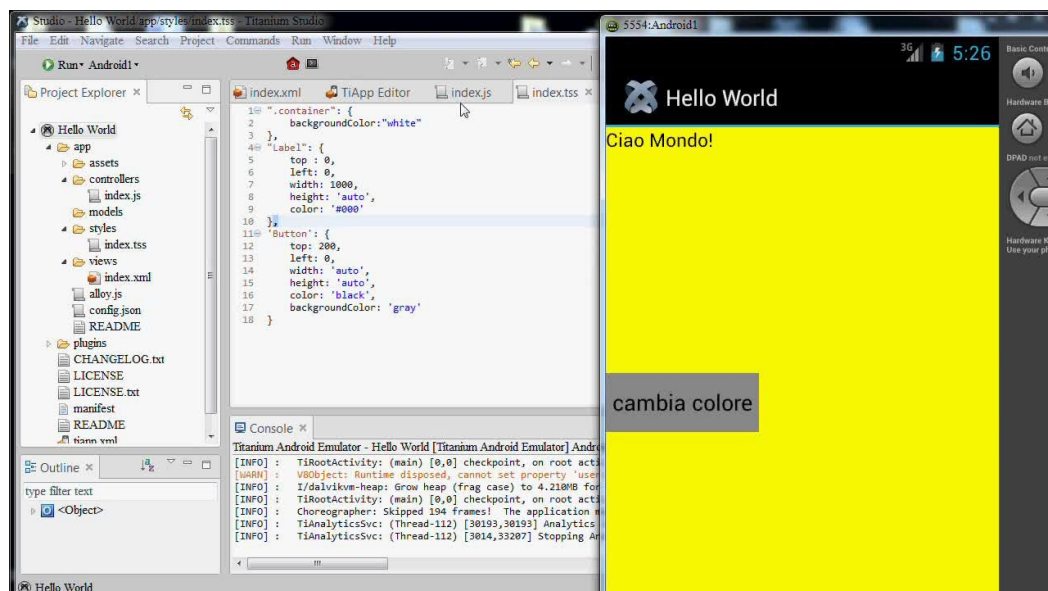
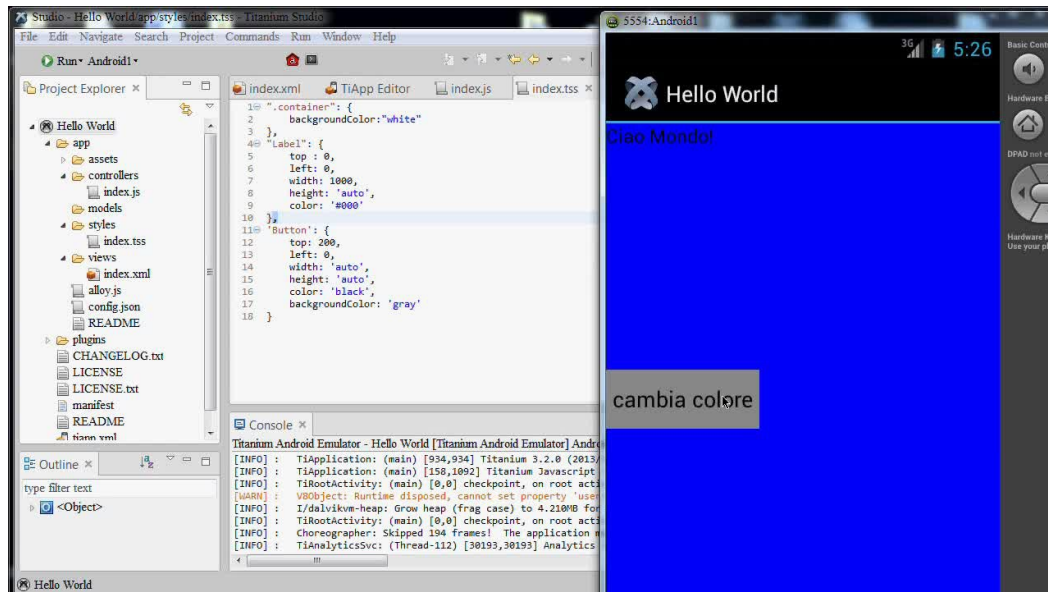


Un appunto sui **TSS**: qui notiamo che **Label** e **Button** hanno le iniziali maiuscole, per cui salta gli occhi la differenza con *“label”*, con la l minuscola, nel **Javascript** del tutorial precedente.

Qui nel **TSS** non stiamo specificando gli **stili** per i singoli elementi, ma per le loro **classi**.

Definendo altri elementi di **classe di stile** **“Label”** o **“Button”**, questi avranno le stesse caratteristiche, per cui per differenziare l'aspetto di due pulsanti, *label* o altri elementi, si può procedere specificando **classi di stile diverse** (e quindi blocchi diversi nel TSS), oppure impostando i valori dei vari parametri esplicitamente via codice... o, ancora, con **#id**, come vedremo nel prossimo capitolo.

Salviamo tutto e avviamo l'applicazione per verificare quanto fatto.



Ricapitolando: abbiamo visto dove e come vengono definiti gli **stili** dei vari elementi di una schermata, ossia nei file **TSS**; inoltre, abbiamo visto come creare un **pulsante**, definendone l'esistenza nell'**xml**, la funzione da eseguire nel **JavaScript** e l'aspetto nel **TSS**, infine, abbiamo visto come accedere ad un **parametro di stile** di un oggetto e modificarlo, mediante **JavaScript**, per realizzare delle operazioni *a runtime* (ovvero durante l'esecuzione dell'applicazione).

* * *

Titanium - Tutorial 3

Elementi dell'interfaccia: TextField, TextArea, Slide, SearchBar e immagini

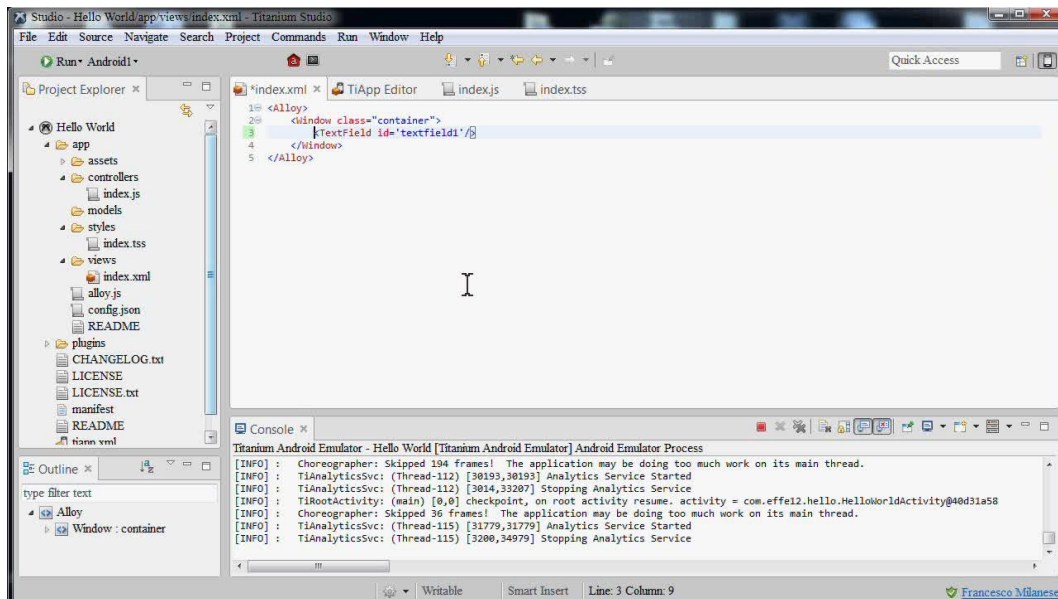
In questo tutorial vedremo come definire alcuni **elementi dell'interfaccia** delle app, ossia *TextField*, *Slide*, *SearchBar* e **immagini**; abbiamo già trattato *Label* e *Button*, definendo anche le **azioni onClick**, nei tutorial precedenti.

In questo tutorial, inoltre, faremo solo una semplice panoramica di questi elementi, in maniera “statica”, senza cioè reperire i loro valori e senza crearli o riposizionarli durante l'esecuzione del programma, ma solo all'inizio, per cui non toccheremo il codice **Javascript** per concentrarci solo su **xml** e **tss** degli elementi.

Apriamo i file tss e xml del nostro progetto e iniziamo dal **TextField**: un semplice campo per **l'immissione del testo** da parte dell'utente, che mostrerà la tastiera virtuale al tocco sull'area di testo.

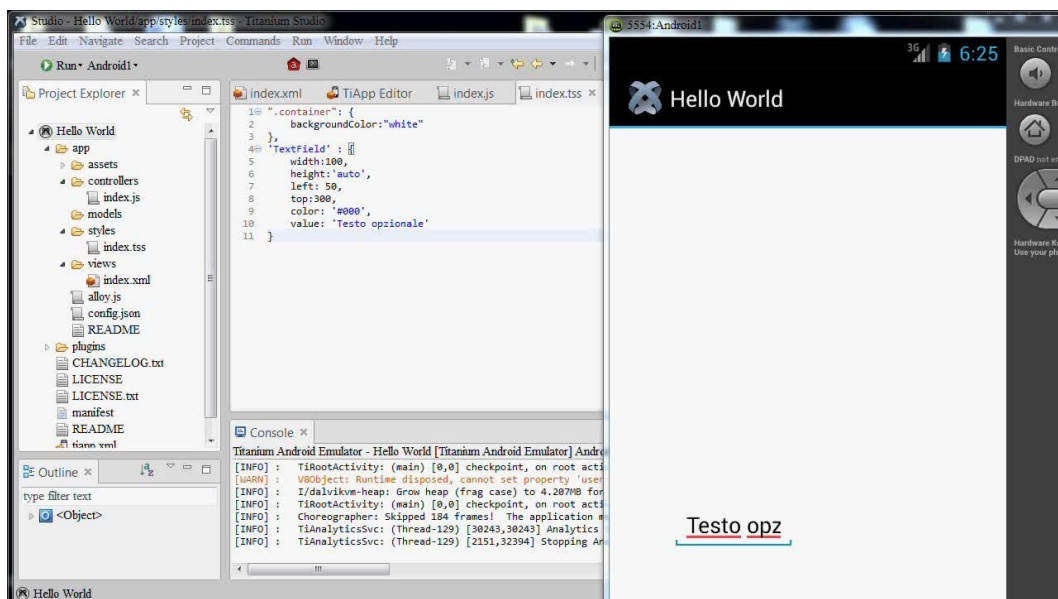
Nel file **xml** scriviamo:

```
<TextField id='textfield1' />
```



mentre nel **tss** scriviamo (con valori per **width**, **left** e **top** d'esempio, ovviamente):

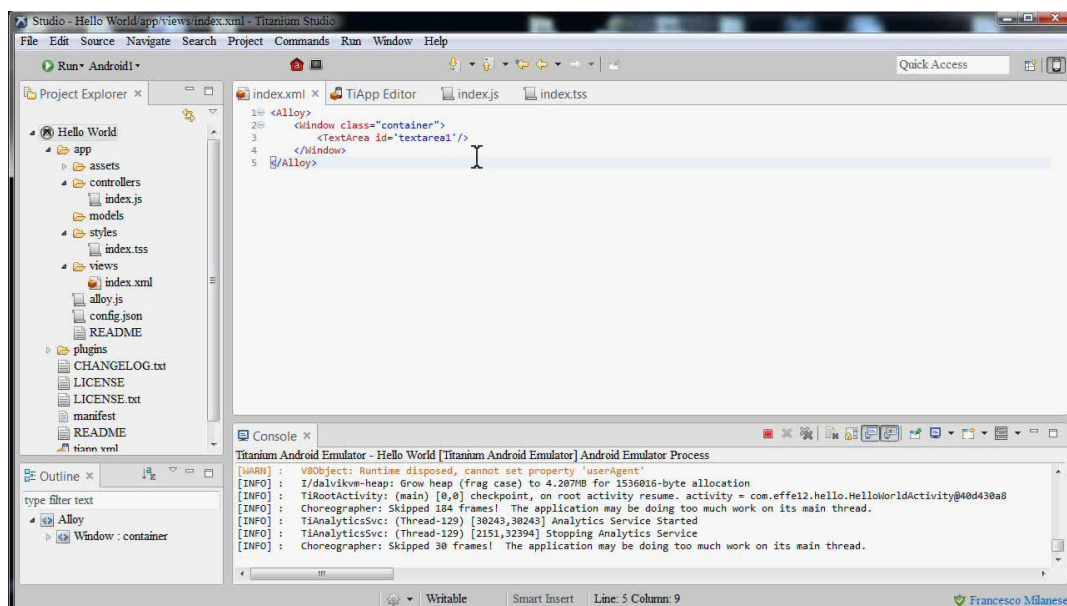
```
'TextField' : {
  width: 100,
  height: 'auto',
  left: 50,
  top: 300,
  color: '#000',
  value: 'Testo opzionale'
}
```



Simile al **TextField** è la **TextArea** che, anziché mostrare una sola riga per l'immissione del testo, visualizza appunto **un'area, ossia più righe**.

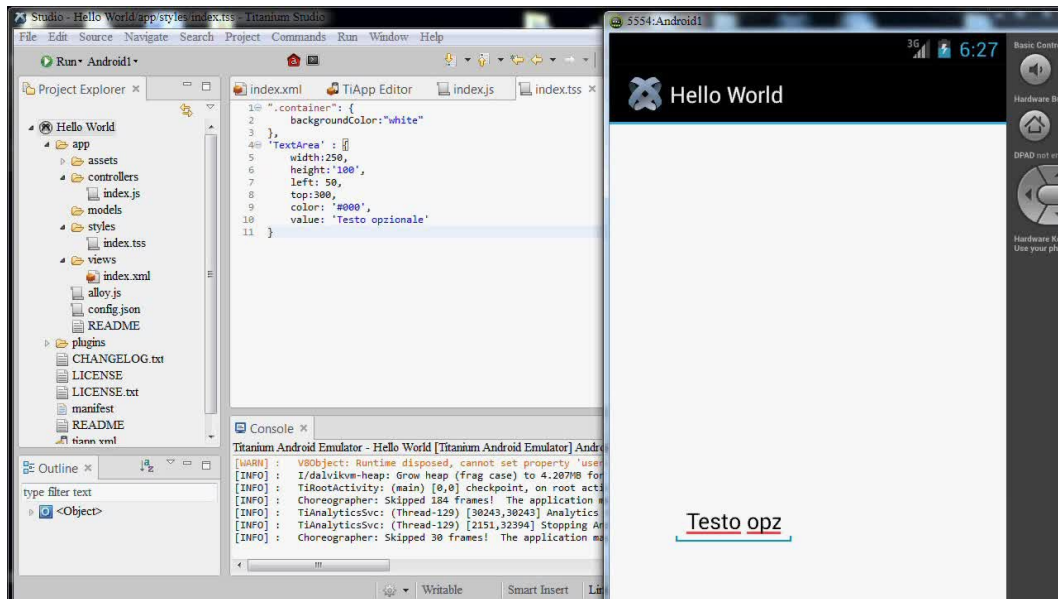
Le istruzioni **xml** e **tss** sono molto simili a quelle del **TextField**, tranne ovviamente che per il tipo (**TextArea**), per cui ad esempio nell'**xml** avremo:

```
<TextArea id='textareal' />
```



e nel **tss** avremo:

```
'TextArea' : {  
    width: 250,  
    height: 100,  
    left: 50,  
    top: 300,  
    color: '#000',  
    value: 'Testo opzionale'  
}
```



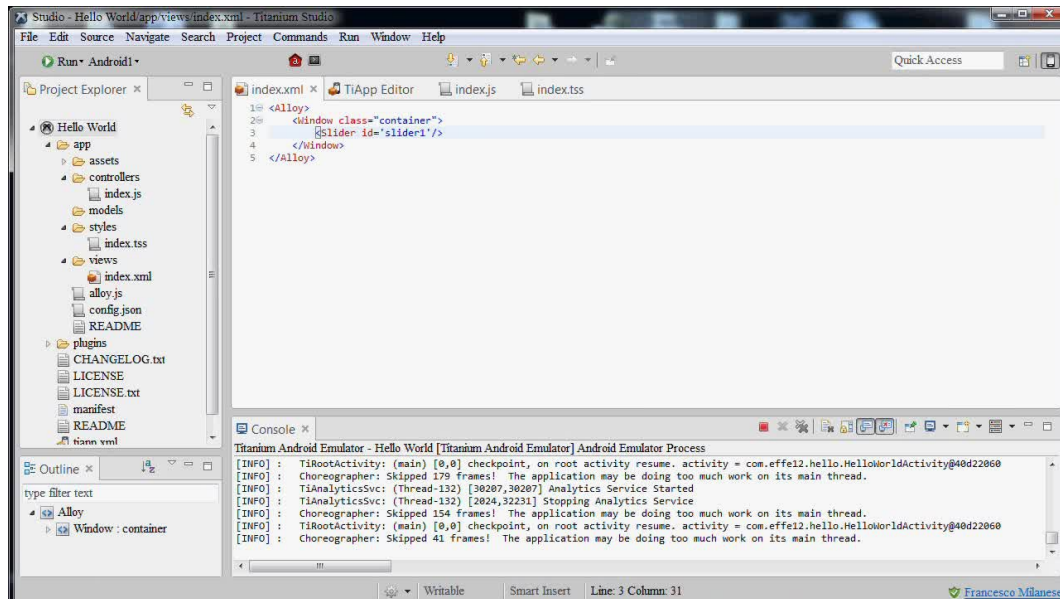
[NOTA: TextField, TextArea e gli altri oggetti dell'interfaccia mettono a disposizione tantissimi campi e metodi per personalizzarne aspetto e funzioni, come padding, bordi, trasparenza, allineamento e altro; elencarli tutti sarebbe solo noioso e poco utile, per cui qui mi limito a mostrare i parametri fondamentali, mentre per gli approfondimenti rimando alla documentazione online di Titanium, all'indirizzo:

<http://docs.appcelerator.com/titanium> .]

Passiamo quindi allo **Slider**: una barra che consente di specificare un **valore numerico in un dato range**, da un minimo ad un massimo, mediante **tocco e trascinamento**.

Nel file **xml**, la riga che dobbiamo scrivere è:

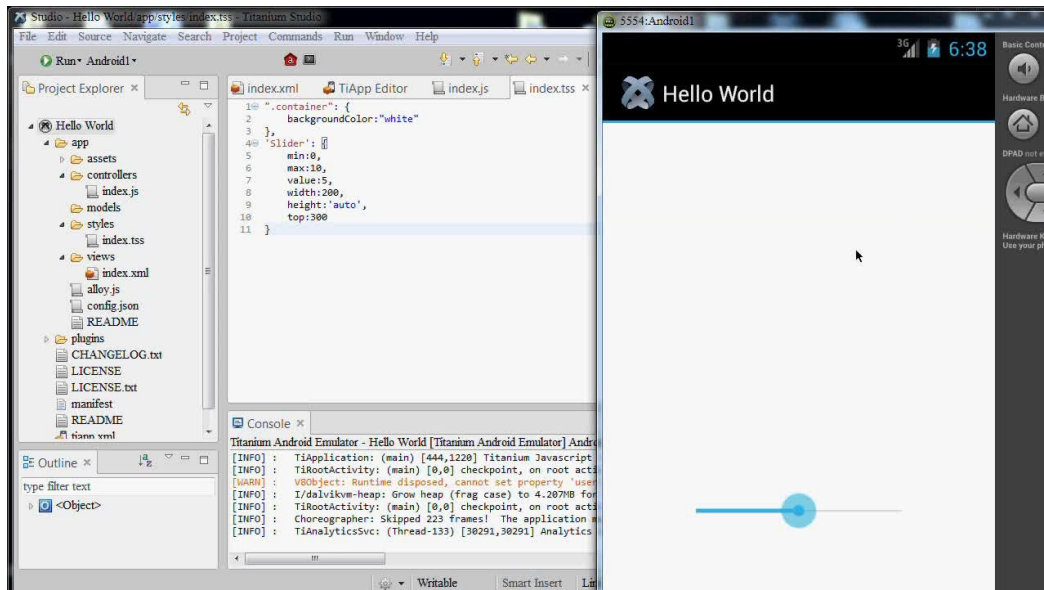
```
<Slider id='slider1' />
```



mentre per quanto riguarda il **tss** i parametri utili sono il **min**, il **max** e il **value**, che servono al componente per dirgli dove posizionare il **marker** che indica il valore attuale e per recuperare tali valori via codice.

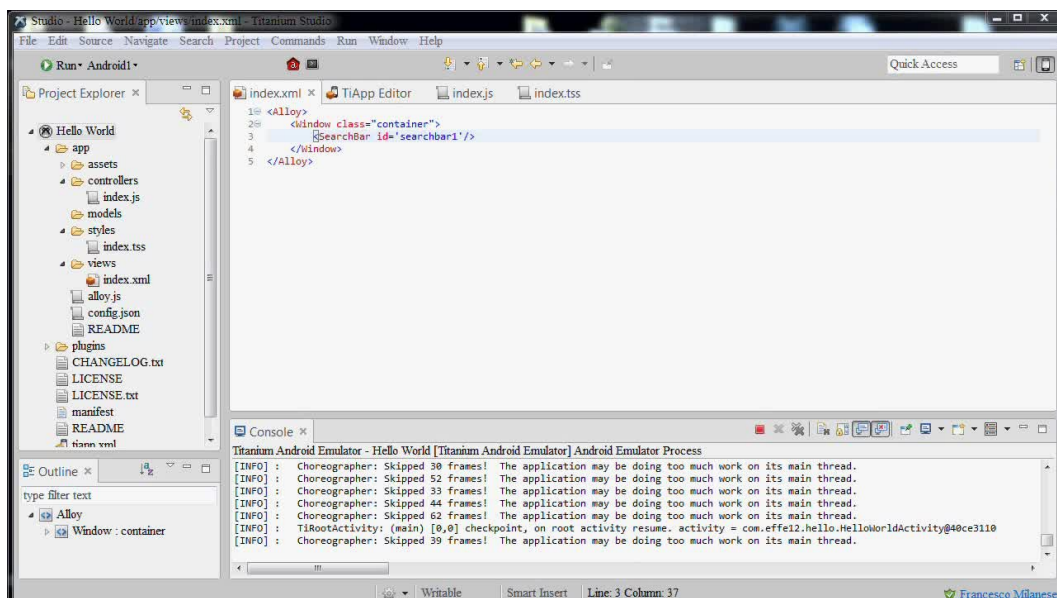
Con la porzione di codice che segue inserisco lo **Slider** al posto del **TextField** di prima, cioè con **left 50** e **top 300**:

```
'Slider' : {  
    min : 0,  
    max : 10,  
    value : 5,  
    width : 200,  
    height : 'auto',  
    top : 300  
}
```



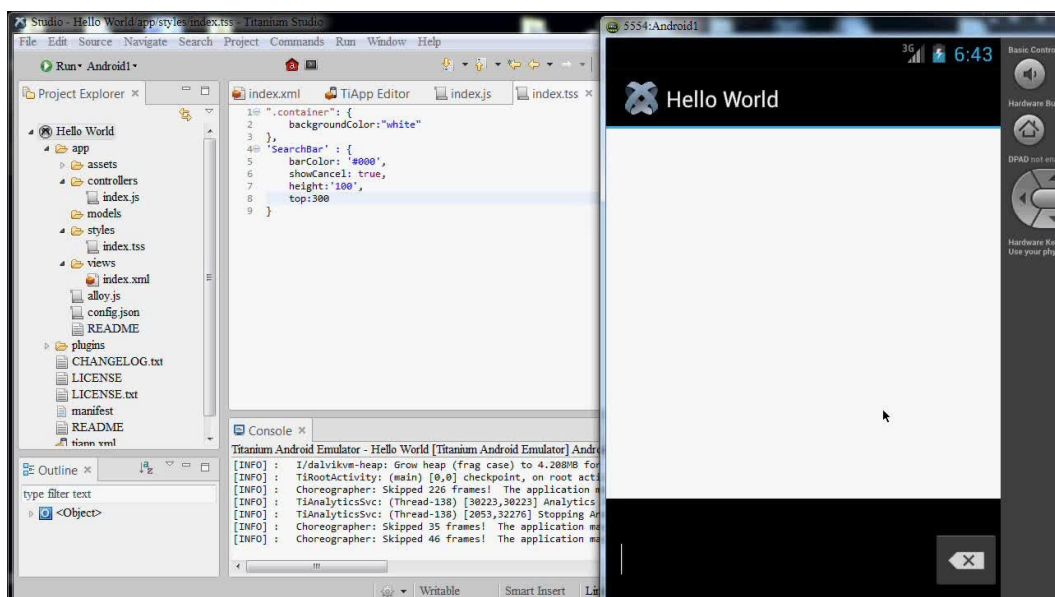
Togliamo quindi anche questo componente e passiamo ad una **Search Bar**, sostanzialmente una **Text Field** per inserire testo per ricerche, ad esempio in coppia con una **Table View**; avremo ad esempio nell'**xml**:

```
<SearchBar id='searchbar1' />
```



e nel **tss**:

```
'SearchBar' : {  
    barColor : '#000',  
    showCancel : true,  
    height : '100',  
    top : 300  
}
```



Ovviamente, questa **SearchBar** non esegue **azioni**: quelle, basate sul valore immesso, dobbiamo impostarle nel **JS** che vedremo un'altra volta; va detto, inoltre, che la **SearchBar** è modellata sull'omonimo componente di **iOS**, per il quale ha piena compatibilità, mentre **non funziona altrettanto bene su Android**, dove ad esempio il **pulsante di cancellazione**, abilitato mediante **ShowCancel**, non funziona, e dove il valore dell'attributo **Value** non può essere specificato nel metodo **CreateSearchBar**.

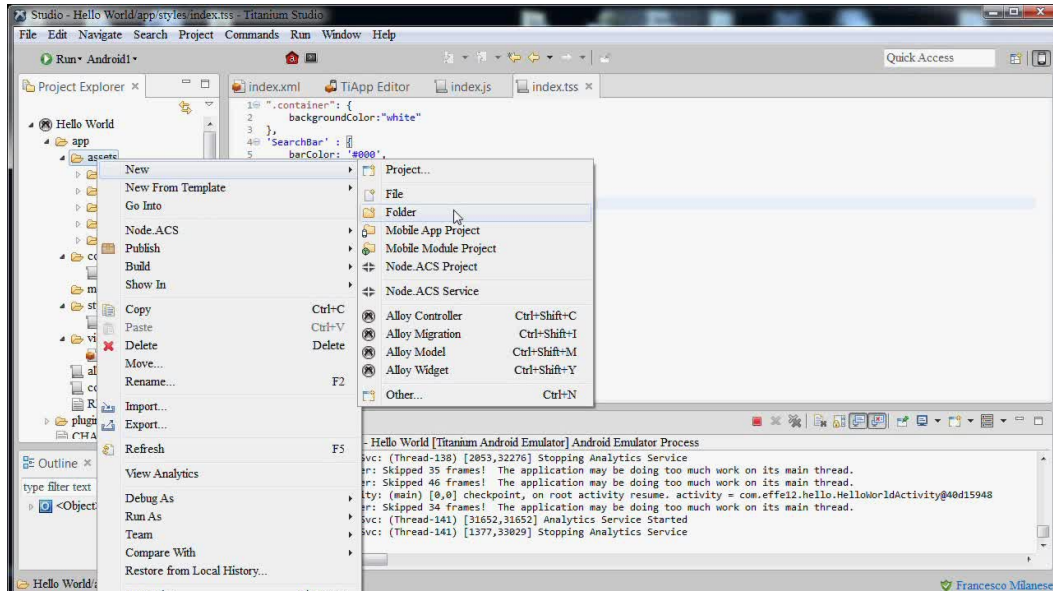
Passiamo quindi all'ultimo elemento da esaminare in questa puntata: le **immagini**.

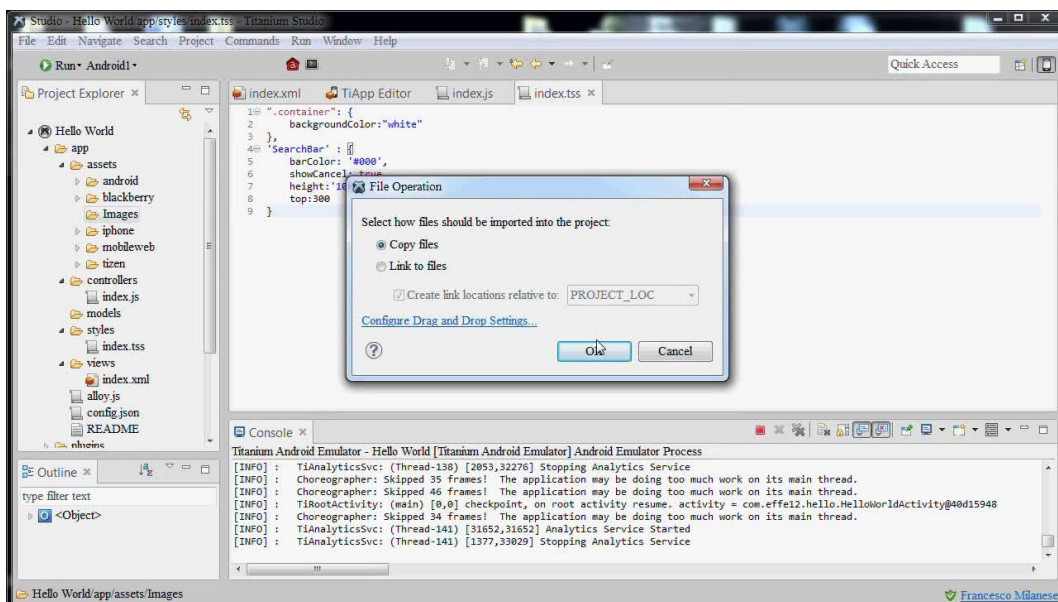
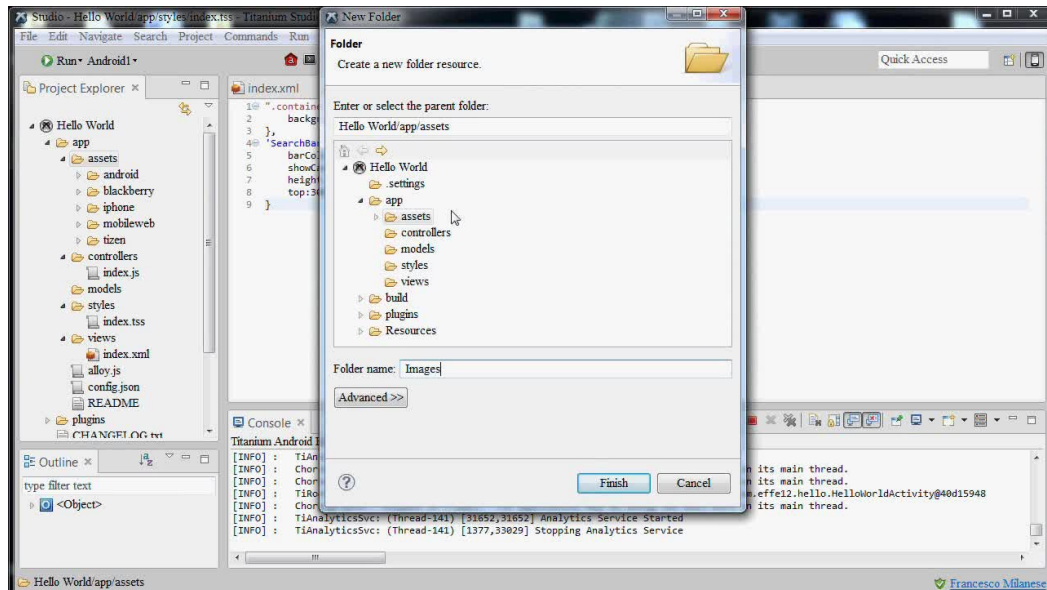
Nel **Project Explorer** notiamo, nella vista ad albero del progetto corrente, **la cartella Assets**.

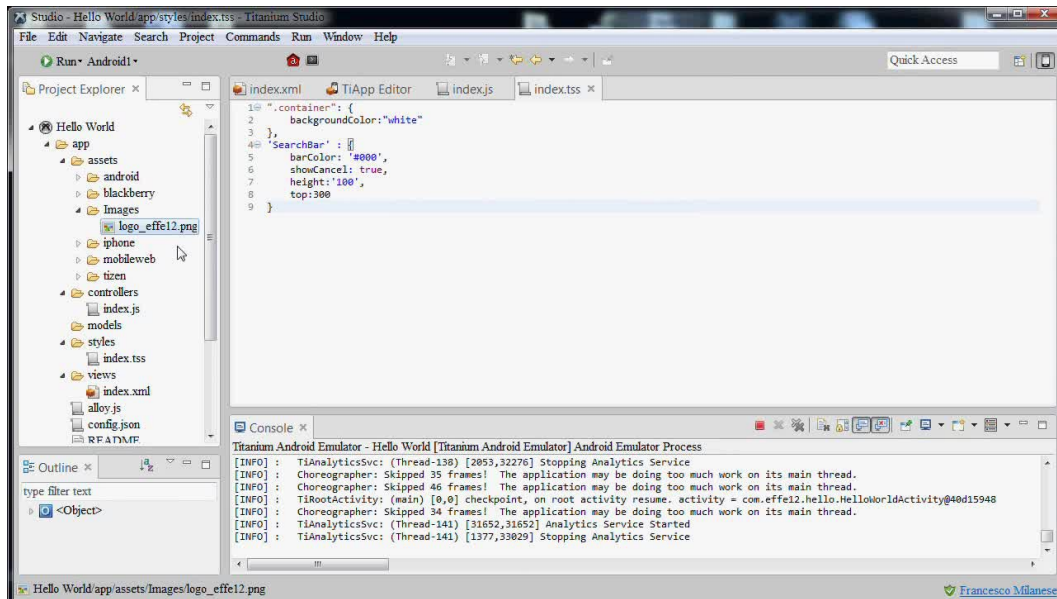
Questo è il **percorso** riconosciuto dai progetti **Titanium** per andare a recuperare i **file delle risorse** (gli *Assets*, appunto) del progetto, come ad esempio immagini, filmati, clip audio, eccetera.

Per mantenere ordinato il progetto, piuttosto che inserire i file direttamente nella cartella conviene **creare delle sottocartelle** in base al tipo di dati, alla *View* che li contiene o ad altri criteri, in modo da ricordarli facilmente e non fare confusione.

Nel nostro caso, visto che il progetto è molto semplice (infatti contiene una sola *View*, con zero *Assets*), aggiungiamo solo una **sottocartella** chiamata **Images**, per differenziare le immagini dagli altri tipi, quindi trasciniamoci dentro – mediante **click e trascinamento** da disco – un'immagine, che nel mio caso si chiama “*logo_effel2.png*”.

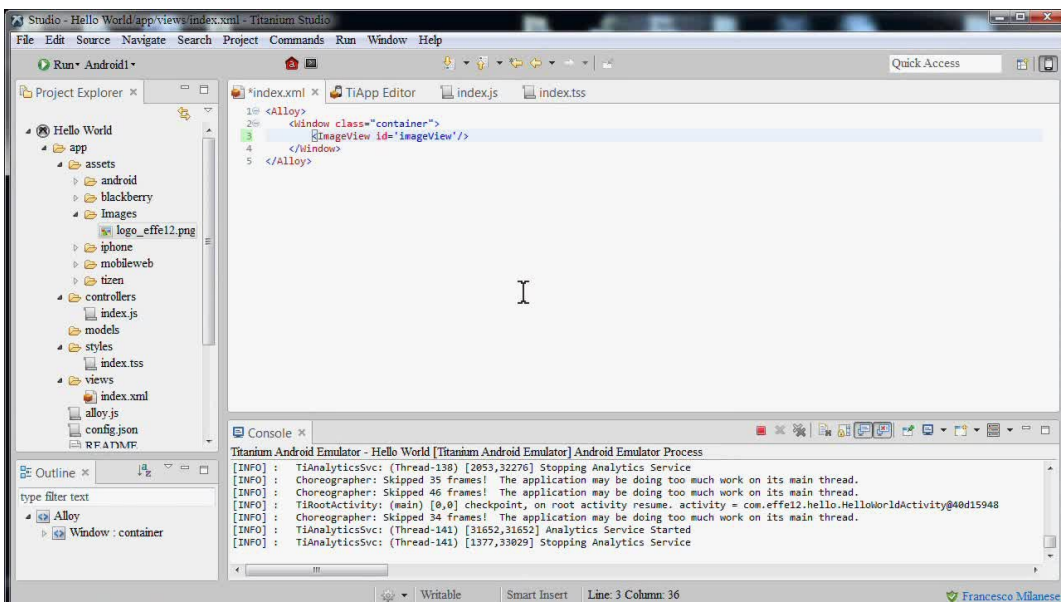






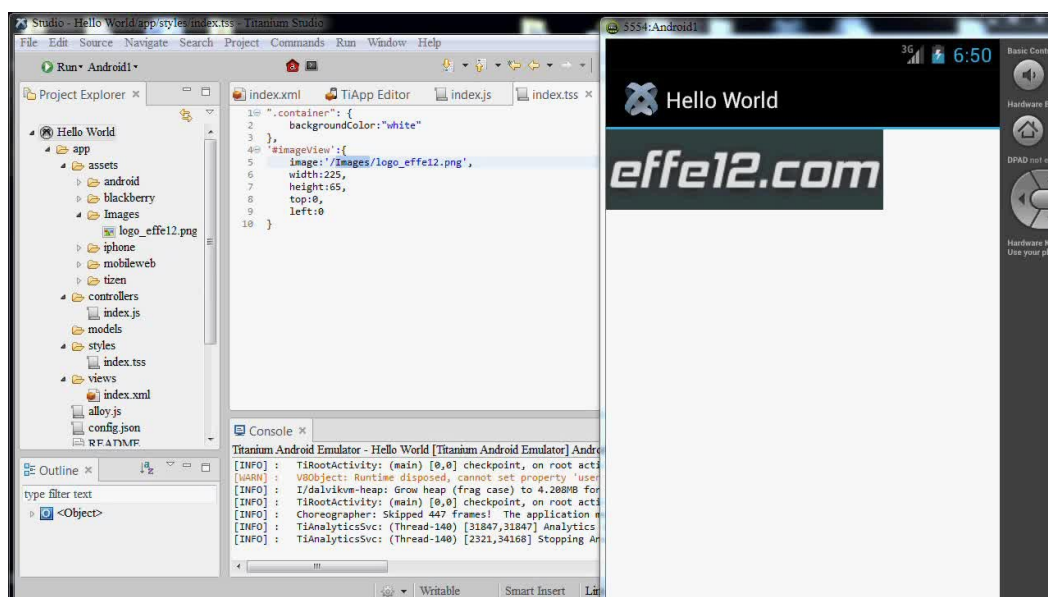
Nell'**xml** scriviamo solo:

```
<ImageView id='imageView1' />
```



mentre nel TSS scriviamo ad esempio:

```
'#imageView1' : {  
    image : '/images/logo_effel2.png',  
    width: 225,  
    height: 65,  
    top: 0,  
    left: 0  
}
```



Da notare il **#imageView1**, per accedere allo **Stile dell'elemento di id 'imageView1'**.

Ricapitolando: abbiamo visto come definire, in maniera statica, **elementi di interfaccia utente** come **TextField**, **TextArea**, **Slider**, **SearchBar** e **immagini**, più le **Label** e i pulsanti **Button** visti nei capitoli precedenti. Nel prossimo capitolo vedremo come creare *dinamicamente* questi elementi.

Vi invito ovviamente ad esplorare caratteristiche e funzionalità, partendo dall'evento **onClick** per i nuovi elementi, per poi approfondire con la **documentazione online di Titanium**.

* * *

Titanium - Tutorial 4

Creazione e modifica dinamica di elementi della UI

In questo capitolo vedremo come creare dinamicamente degli **elementi della UI** (*User Interface*, interfaccia utente) e come recuperarne e modificarne i parametri via codice **Javascript**; in pratica, **un'elaborazione dinamica**, a *runtime*, dell'interfaccia.

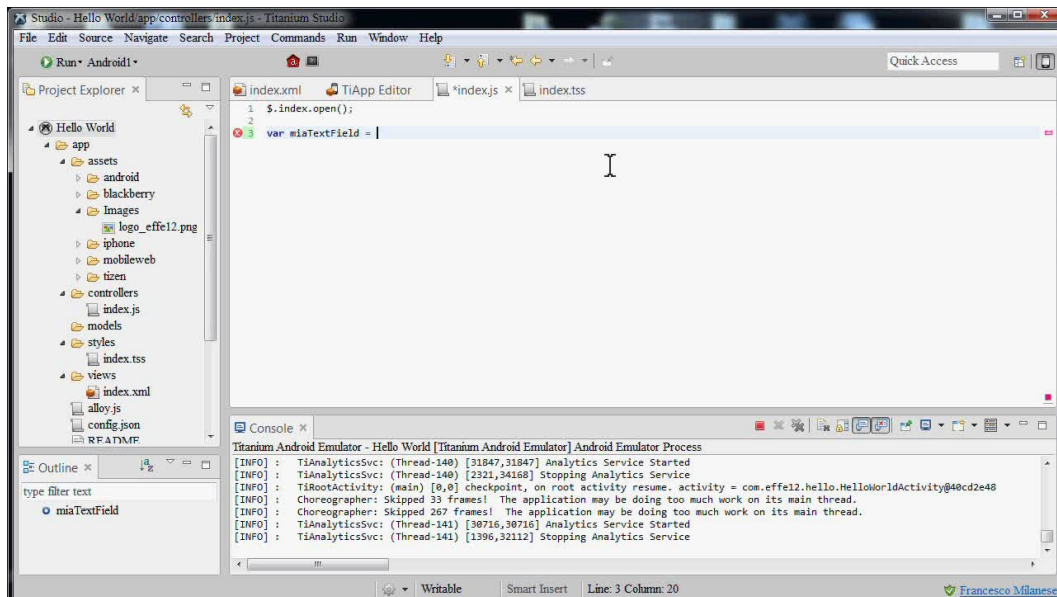
Prima di passare al codice, una **premessa**: quella che voglio darvi qui è un'idea generale di come si procede, anche perché – come vedrete – **la logica per la creazione al volo di un elemento** è sempre **la stessa**: possono cambiare le parole chiave e certi parametri propri di ciascun elemento, ma concentratevi per il momento sui punti fermi; poi, magari con la documentazione alla mano, passerete con calma ai dettagli di implementazione.

In particolare, un elemento di interfaccia creato al volo nel codice va messo nel **Javascript**, non nell'**xml** (dove definiamo gli elementi presenti **all'avvio**, prima della compilazione e dell'esecuzione).

Quando una cosa viene creata nel codice le si associa una **variabile**, per identificarla, inizializzandola con un valore nel caso di tipi primitivi come interi o stringhe, o il costruttore di una classe per creare oggetti non banali... bene, facciamolo!

Nel file ***index.js***, il *Javascript* della nostra applicazione, sotto la riga che apre la schermata corrente (ovvero `$.index.open()`), come descritto in un tutorial precedente), dichiariamo una **variabile da associare ad una *TextField***, con:

```
var miaTextField =
```



Le funzioni di **creazione di elementi di interfaccia** hanno tutte **la stessa forma** ed è per questo che vi ho detto di fare caso al modello generico, in questo momento, perché poi si tratterà di guardare la documentazione per passare al resto.

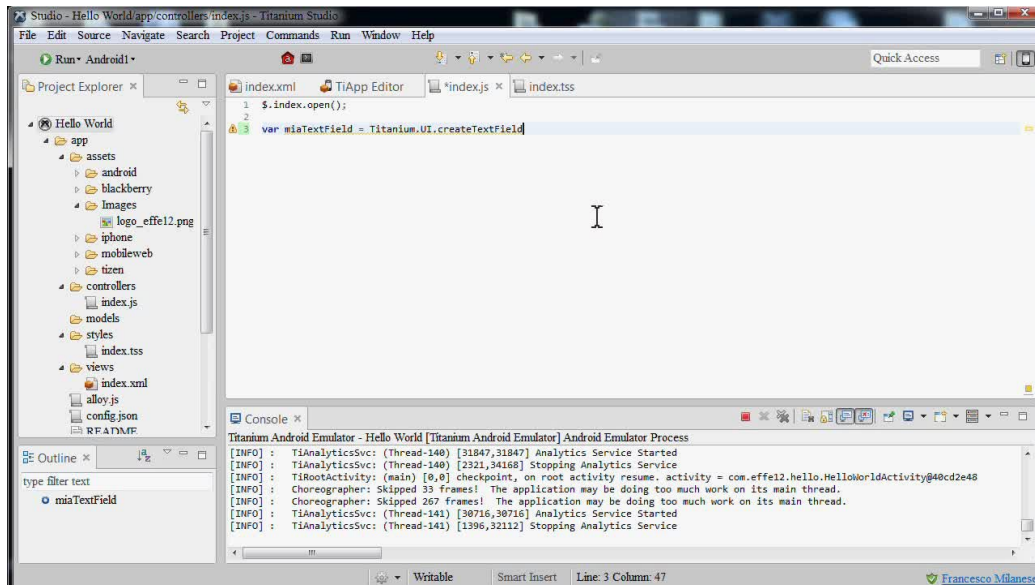
In particolare, si tratta di **funzioni di interfaccia utente (UI, User Interface)** di **Titanium**, per cui abbiamo:

```
var miaTextField = Titanium.UI.create
```

e il nome della funzione create si completa con il nome dell'elemento che vogliamo creare; nel nostro caso, per inserire una **TextField** la funzione è ***createTextField***:

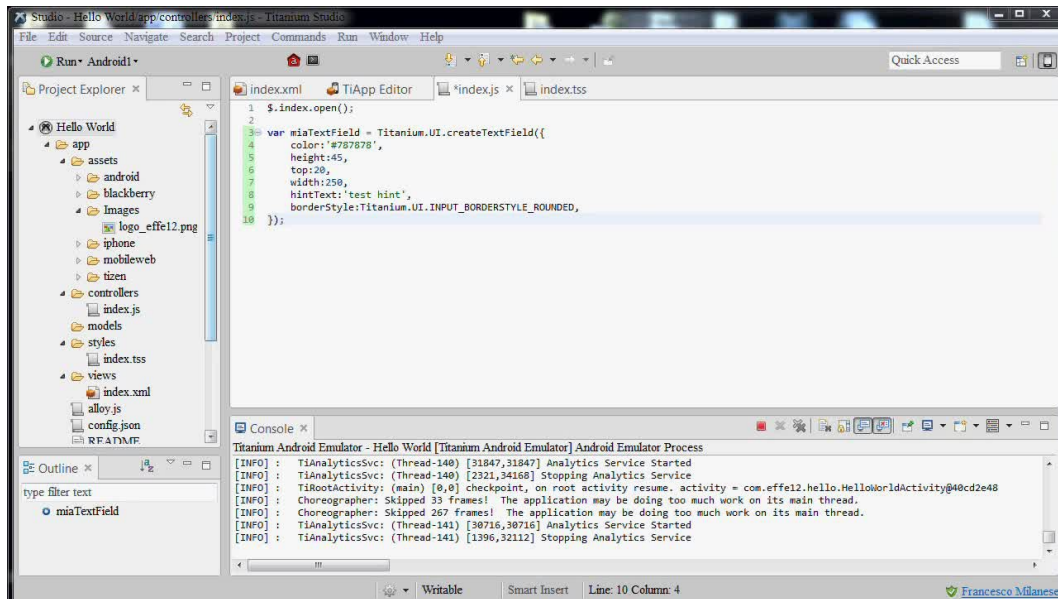
```
var miaTextField = Titanium.UI.createTextField
```

mentre l'argomento di questa funzione è sostanzialmente il TSS, ovvero l'aspetto visivo del componente da creare, specificato tra le parentesi tonde della funzione scrivendo **“in linea”** i **parametri di stile**, così come li definiremmo in un TSS, **parentesi graffe incluse**.



Ecco quindi che la nostra istruzione diventa:

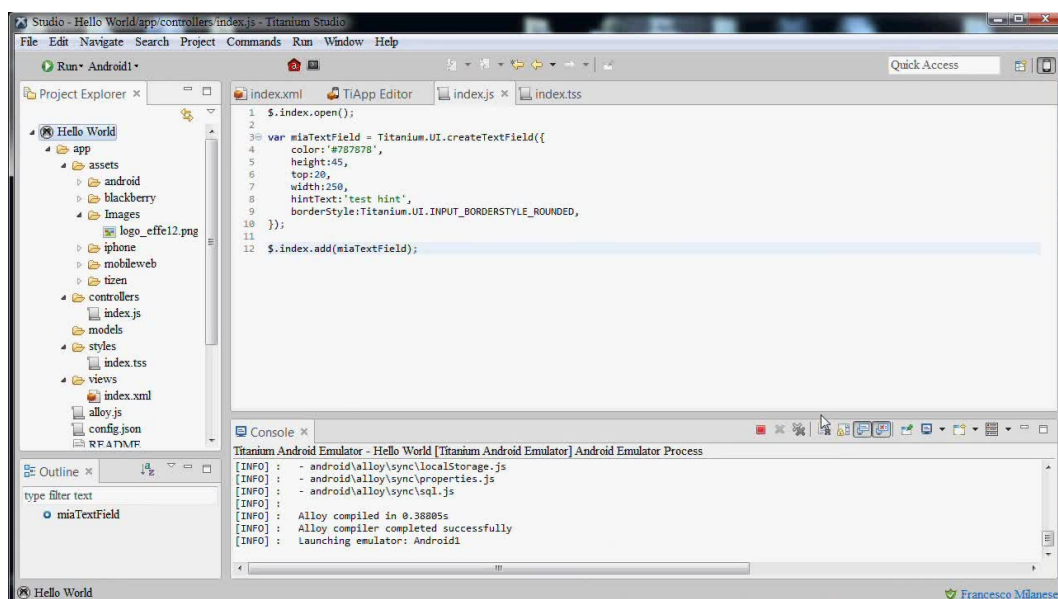
```
var miaTextField = Titanium.UI.createTextField({
    color : '#787878',
    height : 45,
    top : 20,
    width : 250,
    hintText : 'test hint',
    borderStyle : Titanium.UI.INPUT_BORDERSTYLE_ROUNDED,
});
```



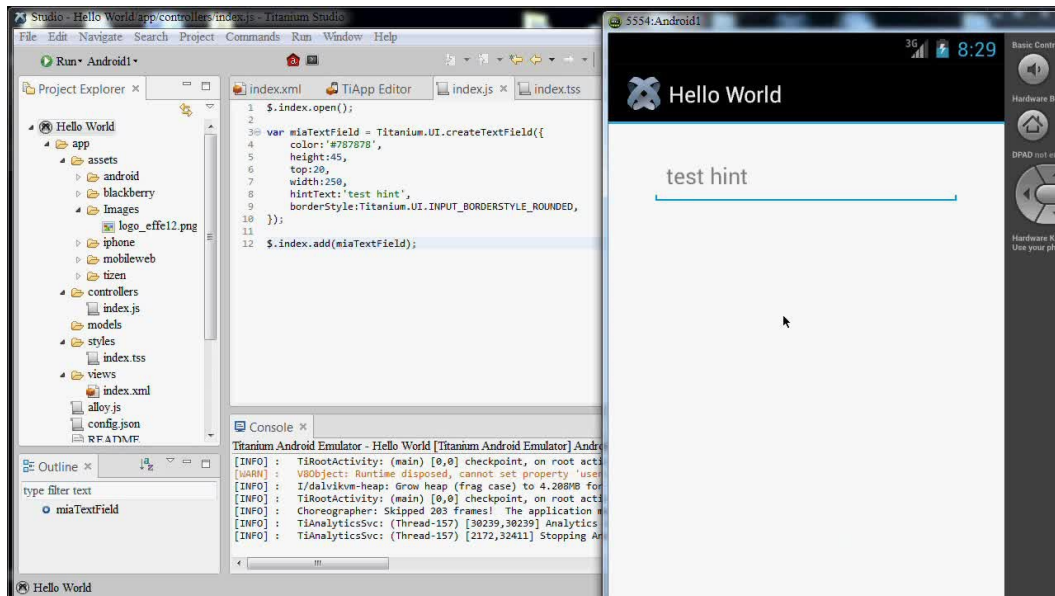
Non è finita qui: mandando in esecuzione l'app, infatti, **non vedremo nulla**, o comunque non la nostra **TextField**, per il semplice motivo che l'abbiamo sì **definita**, associandola ad una variabile, **ma non l'abbiamo inserita** da nessuna parte.

Gli elementi vanno inseriti in aree o contenitori; il contenitore base della nostra **View** è... la **View** stessa, ***index***, per cui scriviamo la seguente istruzione:

```
$.index.add(miaTextField);
```



e adesso sì che possiamo **salvare, compilare ed eseguire** il tutto, trovando a video la nostra **TextField**:

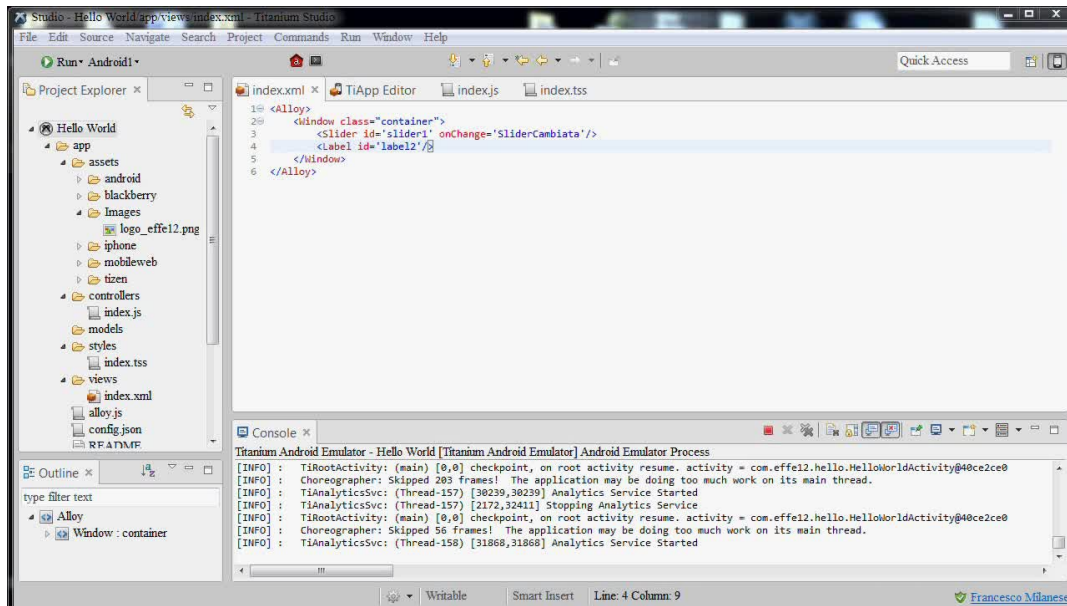


L'accesso a runtime ad un campo o proprietà di un elemento, inclusa una caratteristica del suo aspetto, in realtà è stato trattato indirettamente nei tutorial precedenti, ad esempio **per ricavare il testo di una label** e mostrarlo in un **alert**; anche qui, la logica generale è sempre la stessa, poi cambiano le parole chiave dell'implementazione degli elementi, ma per quello c'è la documentazione online.

Vediamo ad esempio **come mostrare a video**, in una **label**, il valore di una **Slider**.

Nell'**xml** definiamo i due elementi con:

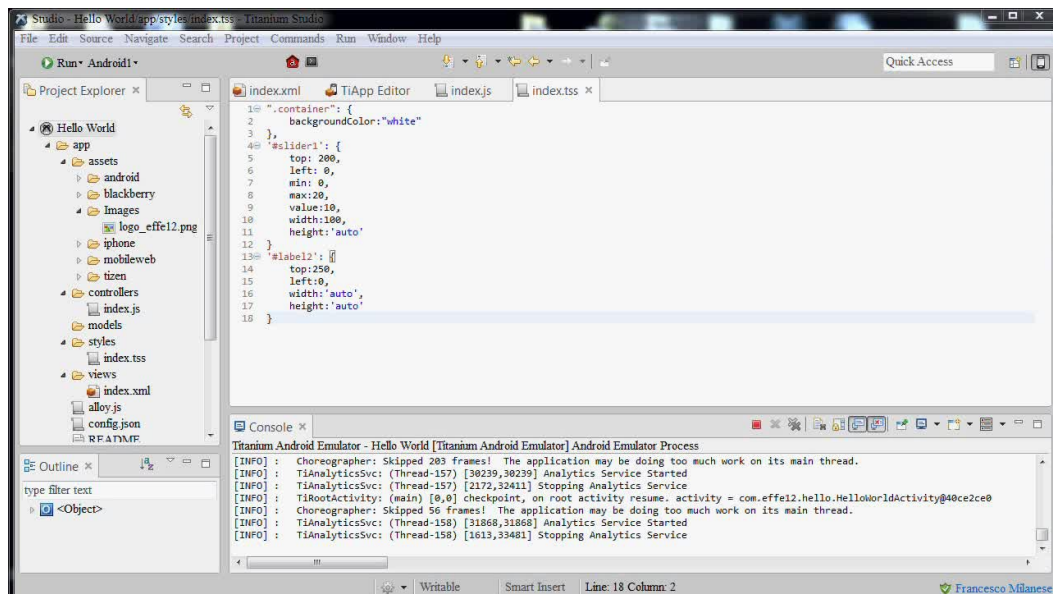
```
<Slider id='slider1' onChange='SliderCambiata' />
<Label id='label2' />
```



quindi nel TSS definiamo degli **stili statici**, per comodità:

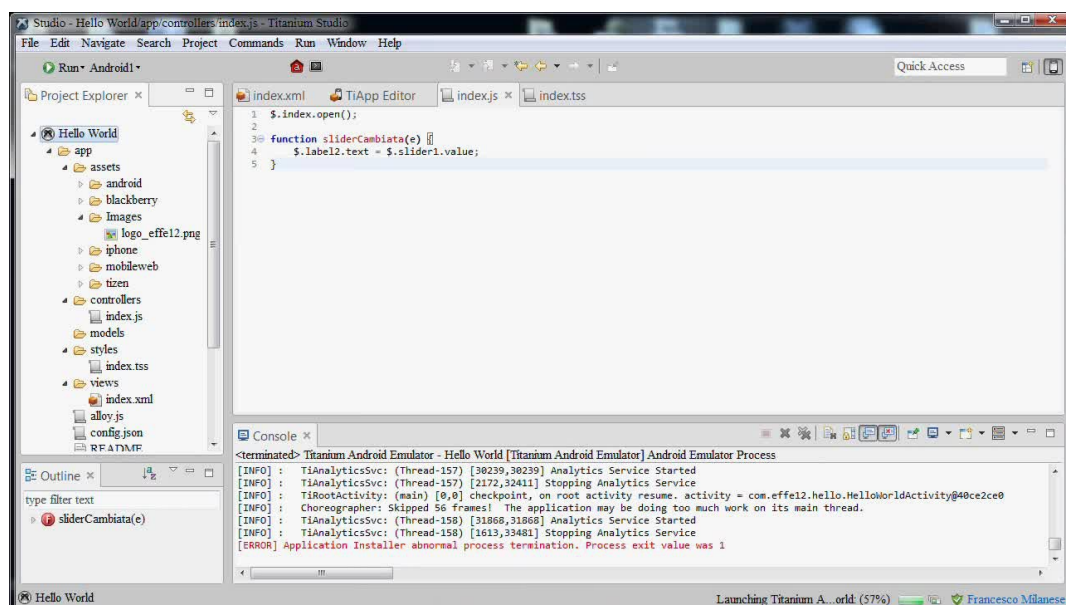
```
'#slider1' : {  
    top : 200,  
    left : 0,  
    min : 0,  
    max : 20,  
    value : 10,  
    width : 100,  
    height : 'auto'  
}
```

```
'#label2' : {  
    top : 250,  
    left : 0,  
    width : 'auto',  
    height : 'auto'  
}
```



Infine nel **Javascript** scriviamo:

```
function sliderCambiata(e) {
    $.label2.text = $.slider1.value;
}
```



Salviamo, compiliamo ed eseguiamo.

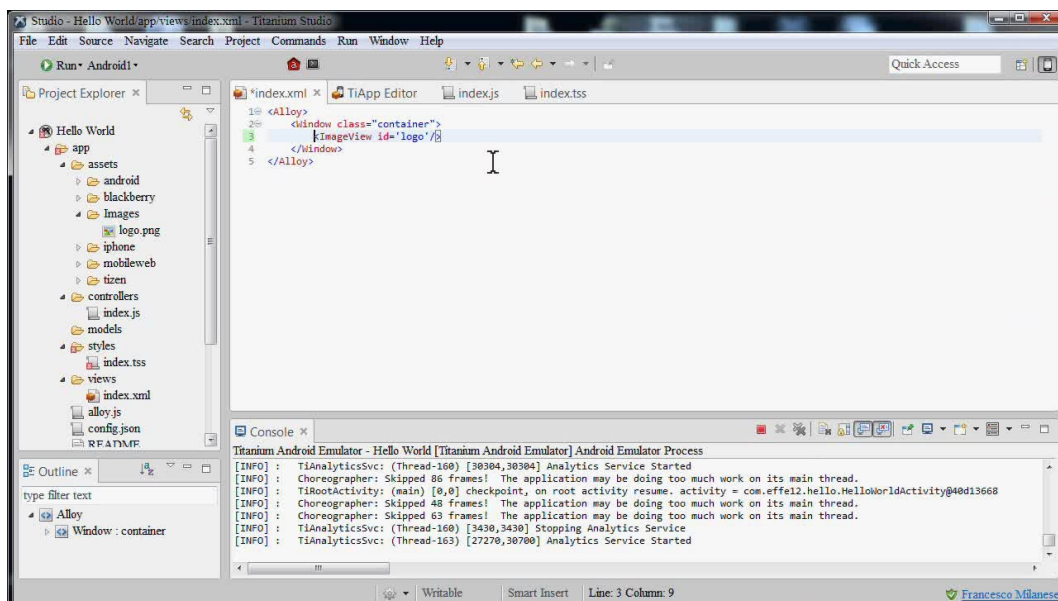
Un'ultima piccola nota, anche se non ce ne dovrebbe essere bisogno (ma per completezza ne parliamo), riguardo **l'accesso e modifica a campi o proprietà TSS e valore di oggetti** inseriti inizialmente in maniera statica, nel TSS di un progetto...

Facciamo un esempio pratico: immaginiamo di volere il logo in basso al centro nella *View*, tuttavia stiamo sviluppando per più dispositivi e non possiamo conoscere a priori le varie risoluzioni.

Tale logo deve essere accessibile da vari elementi e da vari punti dell'applicazione, per cui è anche preferibile metterlo **in maniera statica nell'xml**, definendo il suo *id* per tutta la **View** in un punto facile da trovare.

Procediamo così: specifichiamo la presenza del componente **ImageView** nell'*xml* con:

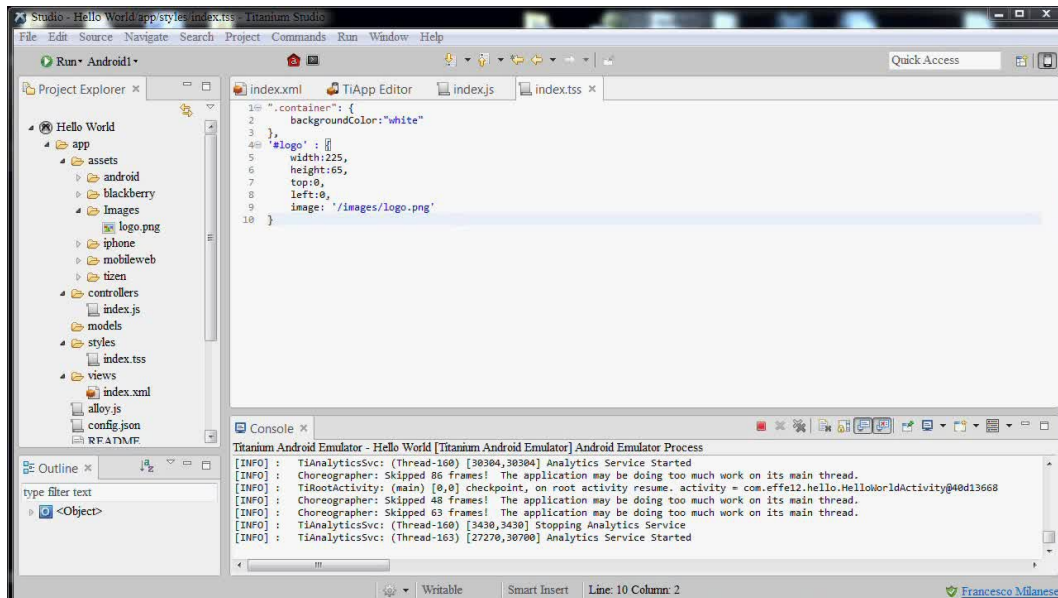
```
<ImageView id='logo' />
```



nel *tss* scriviamo:

```
'#logo' : {
    width : 225,
    height : 65,
```

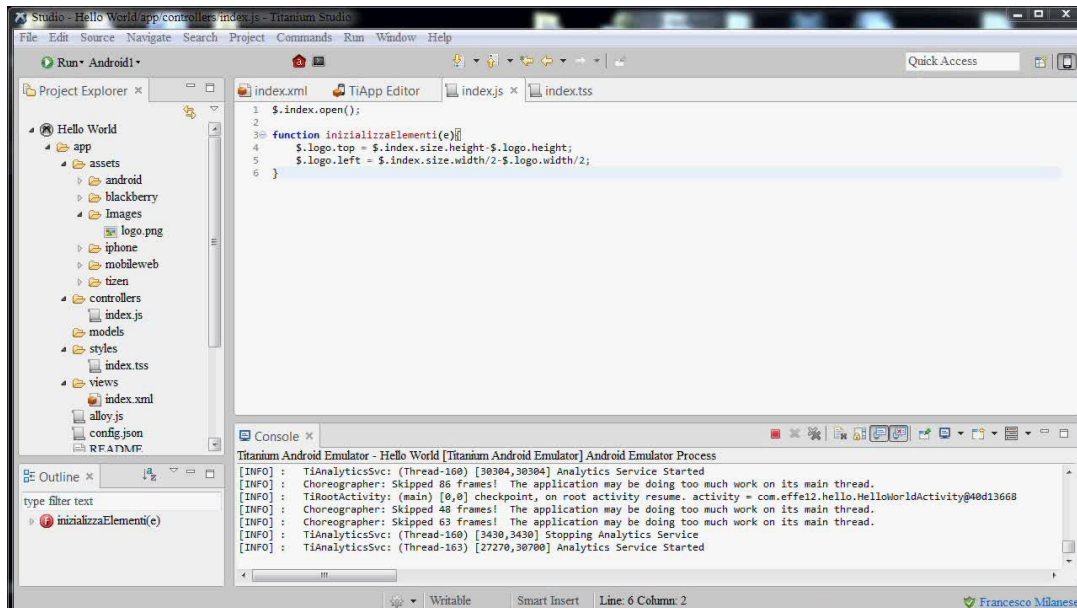
```
top : 0,  
left : 0,  
image : '/images/logo.png'  
}
```



per cui inizializziamo **top** e **left** qui, poi ne cambieremo i valori nel **Javascript**; ovviamente, dovrete mettere un file chiamato **logo.png** nella sottocartella **images** di **Assets**, nel nostro progetto **Titanium** (o un altro file, ma in quel caso dovrete cambiare i nomi dei riferimenti nel codice dell'esempio, ovviamente).

Nel codice **Javascript** di **index.js** scriviamo quindi:

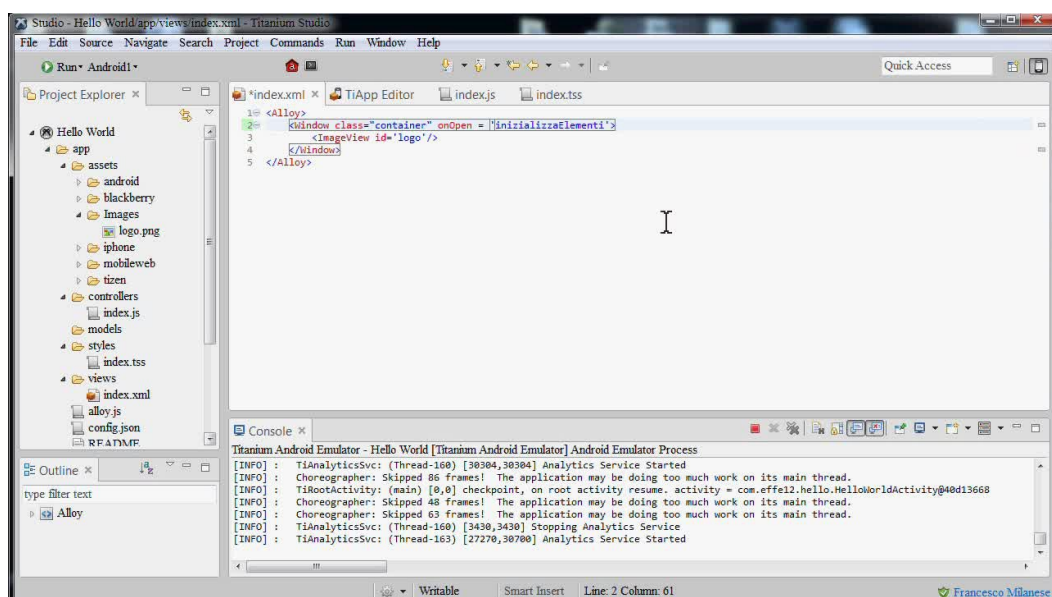
```
function inizializzaElementi(e) {  
    $.logo.top = $.index.size.height - $.logo.height;  
    $.logo.left = $.index.size.width/2 - $.logo.width/2;  
}
```



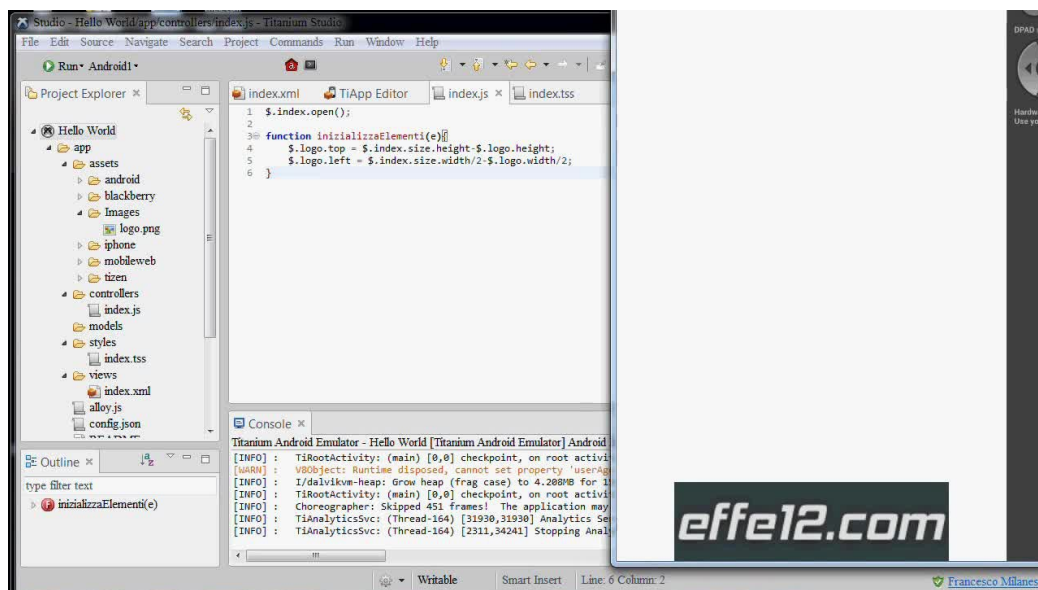
Chi chiama questa funzione ***inizializzaElementi***?

La facciamo invocare **in automatico all'apertura della finestra**, evento che si verifica quando si fa l'***open*** di ***index***, per cui torniamo all'***xml*** della vista e inseriamo, nel **tag di apertura della Window**, la seguente istruzione:

```
<Window class='container' onOpen='inizializzaElementi'>
```



Salviamo, compiliamo ed eseguiamo.



Ricapitolando: è possibile aggiungere elementi di interfaccia mediante le **funzioni di tipo *Ti.UI.create***, ad esempio una *TextField*, specificando poi **tra parentesi tonde e graffe lo stile** del componente; è possibile, inoltre, **accedere a stili e valori dei componenti** via codice *Javascript* e modificarne i valori in base a condizioni o *eventi* (ad esempio la posizione del logo, tenendo conto delle dimensioni della *View* dopo l'*onOpen* della *Window*, la finestra della vista, oppure il testo di una *Label* dopo l'evento *onChange* di una *Slider*).

Prima di chiudere questo capitolo, un esercizio: creare **tre *Slider* per cambiare i canali-colore RGB del canale di sfondo** di una vista (la soluzione all'inizio del prossimo capitolo).

* * *

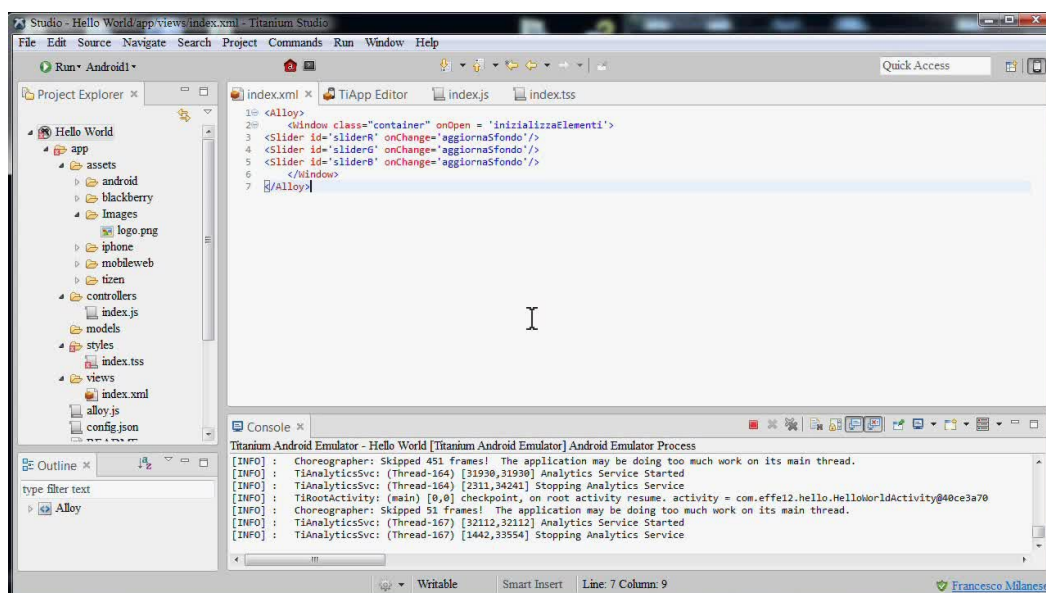
Titanium - Tutorial 5

Memorizzare i dati in un DB SQLite. Le TableView

In questo capitolo vedremo **come memorizzare i dati delle app** su dispositivo mediante le funzioni **Titanium** per accedere ai **DB SQLite**. Analizzeremo inoltre un altro componente di interfaccia: la **TableView**.. prima, però, **risolviamo l'esercizio** della puntata precedente: realizzare tre Slider per cambiare il colore di sfondo.

Partiamo dall'**xml**, dove creiamo i tre *Slider* con la chiamata ad una **funzione “aggiornaSfondo”** in caso di **onChange**:

```
<Slider id='sliderR' onChange='aggiornaSfondo' />
<Slider id='sliderG' onChange='aggiornaSfondo' />
<Slider id='sliderB' onChange='aggiornaSfondo' />
```

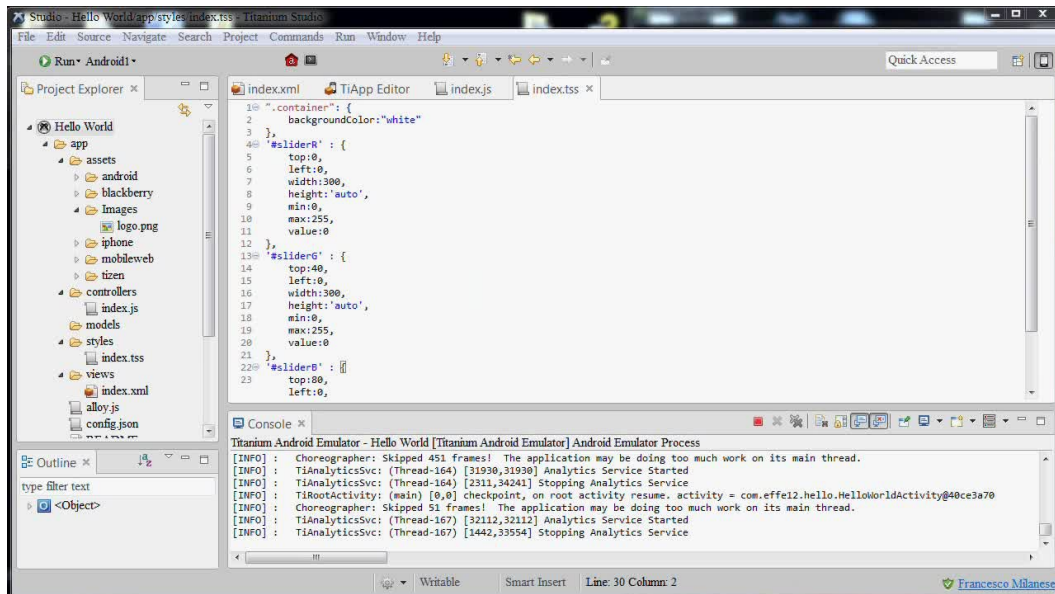


Passiamo quindi al TSS, per specificare l'aspetto di questi componenti:

```
'#sliderR' : {  
  top : 0,  
  left : 0,  
  width : 300,  
  height : 'auto',  
  min : 0,  
  max : 255,  
  value : 0  
},
```

```
'#sliderG' : {  
  top : 40,  
  left : 0,  
  width : 300,  
  height : 'auto',  
  min : 0,  
  max : 255,  
  value : 0  
},
```

```
'#sliderB' : {  
  top : 80,  
  left : 0,  
  width : 300,  
  height : 'auto',  
  min : 0,  
  max : 255,  
  value : 0  
}
```



Infine, il file **Javascript**: creiamo la funzione ***aggiornaSfondo()*** per cambiare il colore di sfondo in caso di tocco su una delle tre **Slider**. Su questo punto bisogna fare un paio di **considerazioni preliminari**.

Innanzitutto, passeremo a ***\$.index.background*** il colore sotto forma di **stringa** di un valore **esadecimale**.

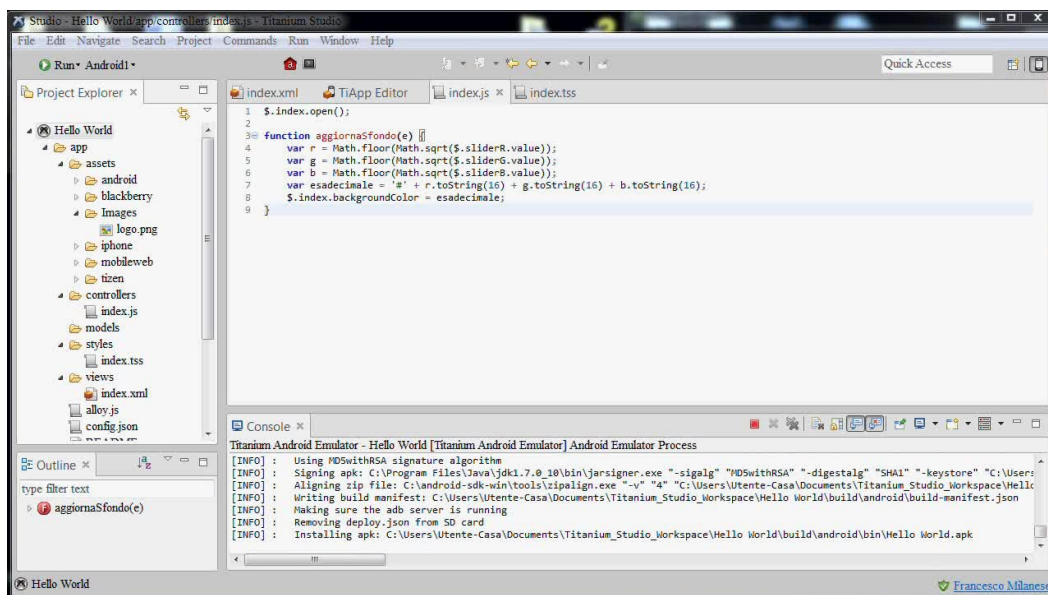
I valori restituiti dagli **Slider** saranno nel range **0-255 con valori decimali**, mentre la stringa esadecimale deve essere nella forma **#RGB**, con un **solo carattere** per esprimere il valore di ciascun canale-colore.

Per ottenere questo risultato, di ciascuno **Slider** dovremo prendere il **valore** (come detto, 0-255 con valori decimali), ottenerne la **radice quadrata** (in modo tale da mappare tutto tra 0, che in esadecimale è 0, e 15, che in esadecimale è F, l'ultimo valore), quindi **arrotondarlo** per difetto: otterremo quindi un intero tra 0 e 15, estremi inclusi.

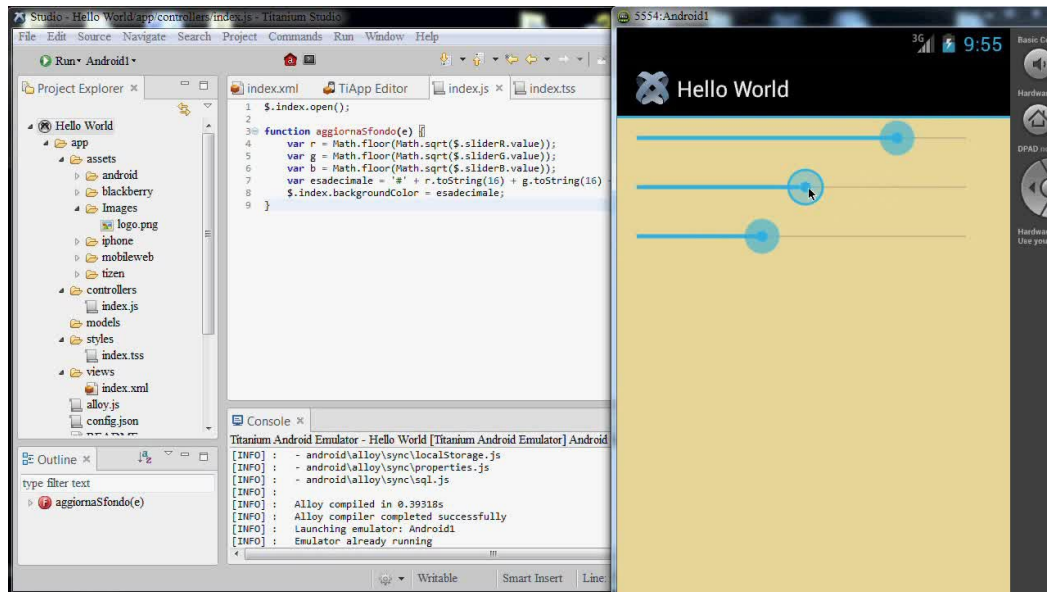
Per realizzare la **conversione in esadecimale** utilizzeremo la funzione nativa ***toString*** e, tra parentesi tonde, **come argomento**, **16**, che effettua appunto una **conversione in formato hex**.

Il codice che segue crea **quattro variabili temporanee**, ma solo per motivi di chiarezza espositiva: in realtà, *tutta la funzione potrebbe essere compattata in una sola riga di codice*, evitando tra l'altro di creare quattro variabili usa-e-getta ad ogni evento ***onChange*** sugli **Slider**, quindi come ulteriore esercizio potete **ottimizzare la seguente funzione**:

```
function aggiornaSfondo(e) {  
    var r = Math.floor(Math.sqrt($.sliderR.value));  
    var g = Math.floor(Math.sqrt($.sliderG.value));  
    var b = Math.floor(Math.sqrt($.sliderB.value));  
    var esadecimale = '#' + r.toString(16) + g.toString(16) +  
        b.toString(16);  
    $.index.backgroundColor = esadecimale;  
}
```



Salviamo, compiliamo ed eseguiamo il tutto.



* * *

Passiamo ora all'argomento vero e proprio di questo capitolo: la **memorizzazione con persistenza** di dati sul dispositivo, inserendoli nel **database SQLite integrato**, e la creazione di una **TableView** nella quale visualizzeremo proprio questi dati.

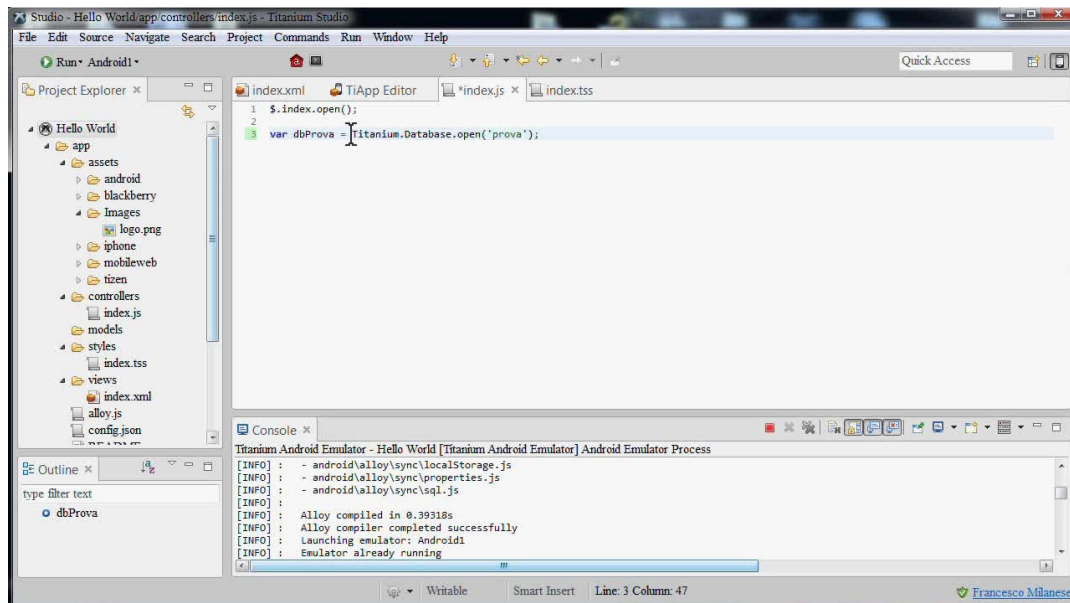
APRIRE UN DATABASE - OPEN

I **database** vengono **aperti o creati nell'app** mediante l'istruzione:

```
Titanium.Database.open(' [nomeDB] ');
```

associando tale elemento ad una **variabile**, per cui ad esempio per creare il **database 'prova'** da associare alla **variabile 'dbProva'** scriveremo:

```
var dbProva = Titanium.Database.open('prova');
```



Per cui: alla prima esecuzione dell'app, quando il **db** ancora non esiste, **“prova”** viene creato, mentre alle esecuzioni successive la **funzione open** avrà l'effetto di recuperare il db **“prova”** con tutto il suo contenuto da disco ed assegnarlo alla **variabile “dbProva”**, tramite la quale potremo poi fare **interrogazioni al db**, inserire nuovi dati o modificare o cancellare quelli esistenti.

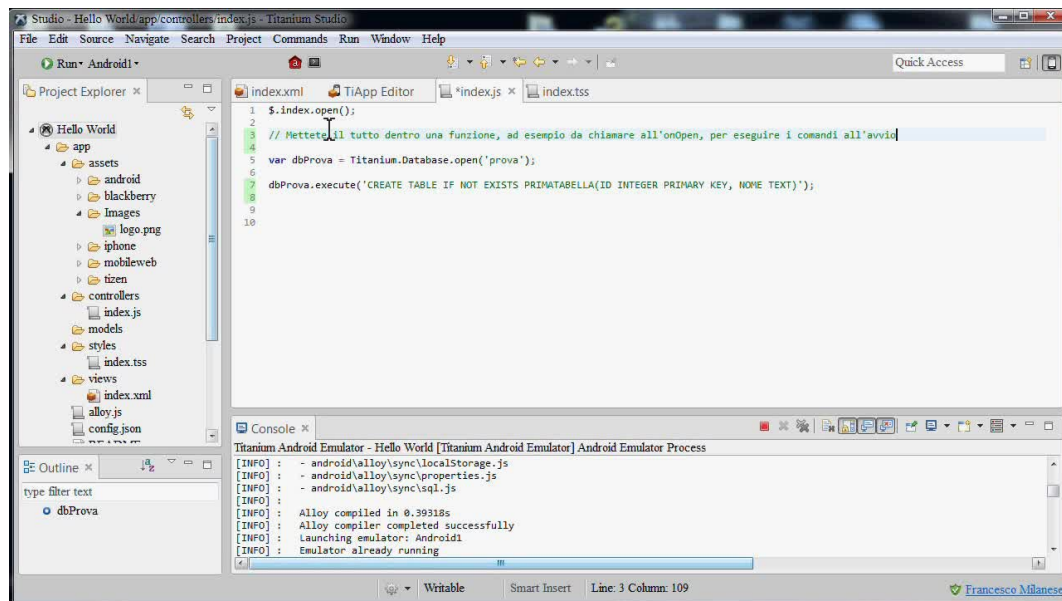
Il linguaggio per creare tabelle, inserire o recuperare dati, è **SQLite**, molto simile a **MySQL** e altri linguaggi di questa famiglia; in effetti, si utilizza la **funzione execute()** sulla variabile del db passandole come parametro **un comando SQL** vero e proprio.

CREARE UNA NUOVA TABELLA – CREATE TABLE

Per esempio, per creare una nuova tabella all'interno del db **“prova”**, avremo:

```
dbProva.execute('CREATE TABLE IF NOT EXISTS PRIMATABELLA(ID INTEGER PRIMARY KEY, NOME TEXT)');
```

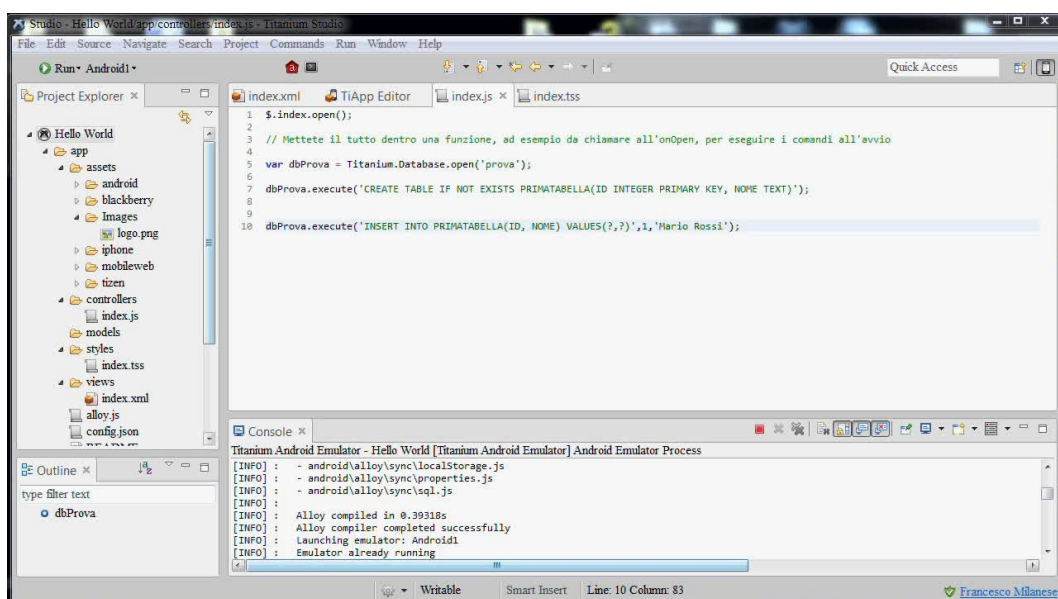
ovvero: crea – se non esiste già – una tabella chiamata **“PRIMATABELLA”**, che avrà **due campi per record (due colonne)**: un campo numerico chiamato **ID** e un campo di tipo testuale chiamato **NOME**.



INSERIMENTO DI UN ELEMENTO: INSERT INTO

Per inserire un elemento nel db useremo quindi il comando *execute* e il comando SQL dell'inserimento, ossia **INSERT INTO** seguito dal nome della tabella e dai “*values*”, ossia i parametri del record da inserire; ad esempio:

```
dbProva.execute('INSERT INTO PRIMATABELLA(ID, NOME) VALUES(?,?)', 1, 'Mario Rossi');
```



Vi spiego il senso di quei “?” al posto dei due **argomenti di values**: per evitare problemi con apici e doppi apici, mettiamo quei “?” come “segnaposto” per i dati, quindi chiudiamo la stringa del comando SQL e, a seguire, sempre dentro la funzione *execute*, mettiamo in ordine le variabili (nel nostro caso, un intero – l’ID numerico, che farà quindi riferimento al primo punto interrogativo – e una stringa specificata tra apici o doppi apici, nel nostro caso un nome che verrà inserito nel secondo campo, quello testuale, in riferimento al secondo punto interrogativo di *values*).

Attenzione, piccola nota a margine: **il campo ID è chiave primaria**, quindi non può contenere valori **duplicati**; per questo motivo, successive esecuzioni di questo script genereranno errore, in quanto si tenterà di inserire altre volte un campo con *ID* pari a 1.

Bisogna quindi provvedere a mettere **blocchi di gestione delle eccezioni a runtime o controlli** sull’esistenza di *ID* nel db, per evitare crash dell’applicazione.

MODIFICARE UN VALORE: UPDATE

Vediamo quindi un esempio di *Update*, ossia come **modificare un valore**.

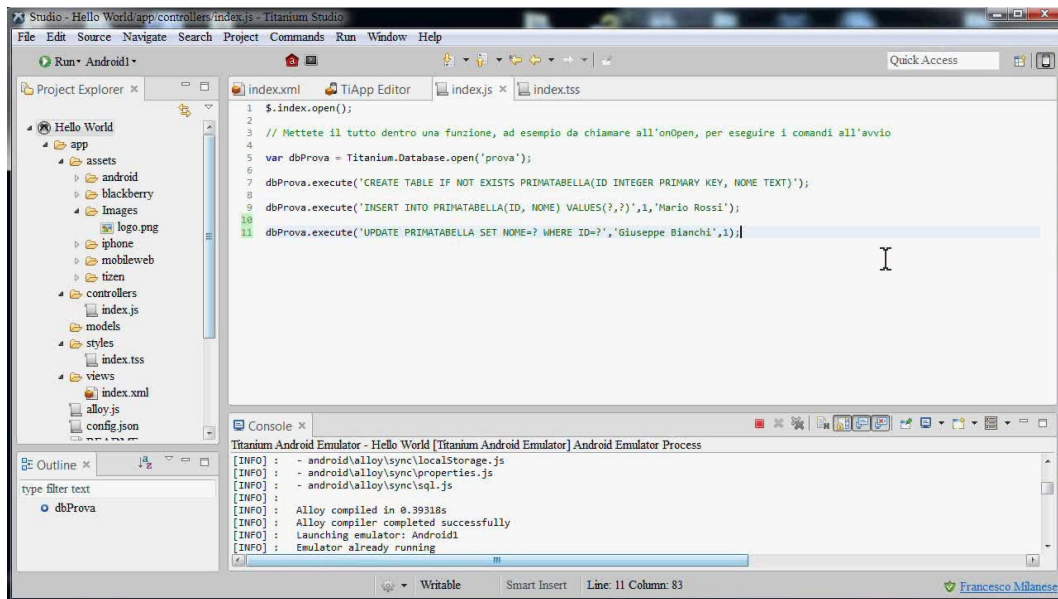
La sintassi è:

```
UPDATE nome_tabella SET nome_campo = nuovo_valore WHERE condizione;
```

ad esempio:

```
dbProva.execute('UPDATE PRIMATABELLA SET NOME=? WHERE ID=?', 'Giuseppe  
Bianchi', 1);
```

quindi come vedete anche qui si mantiene **la logica dei punti interrogativi come segnaposti** per le variabili da specificare in seguito.



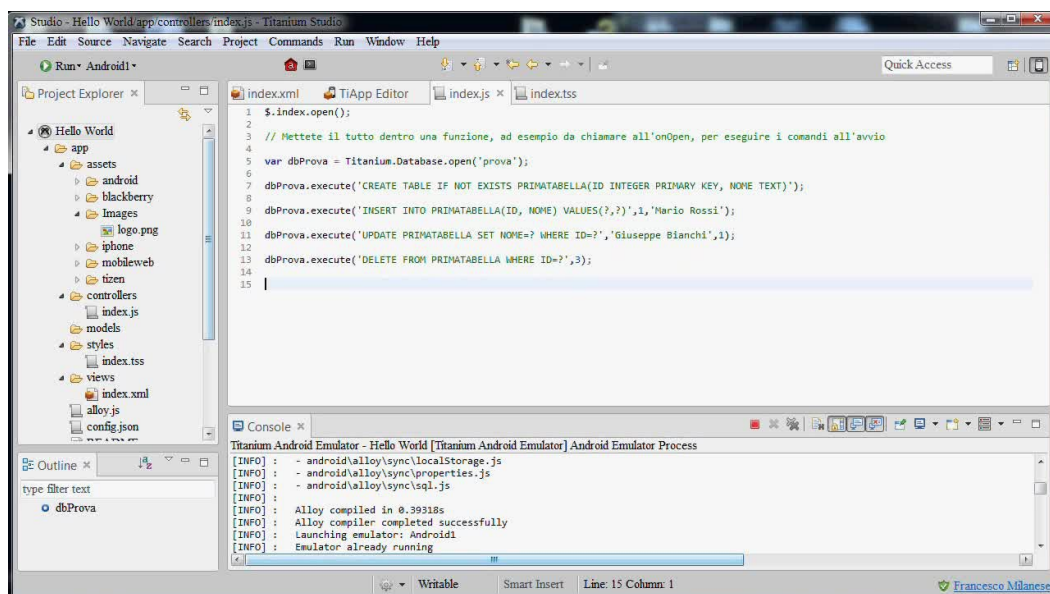
CANCELLAZIONE DI UN RECORD: DELETE FROM

Passiamo alla **cancellazione di un record**, con sintassi:

DELETE FROM nometabella WHERE condizione;

ad esempio:

```
dbProva.execute('DELETE FROM PRIMATABELLA WHERE ID = ?', 3);
```



RECUPERARE I DATI: SELECT

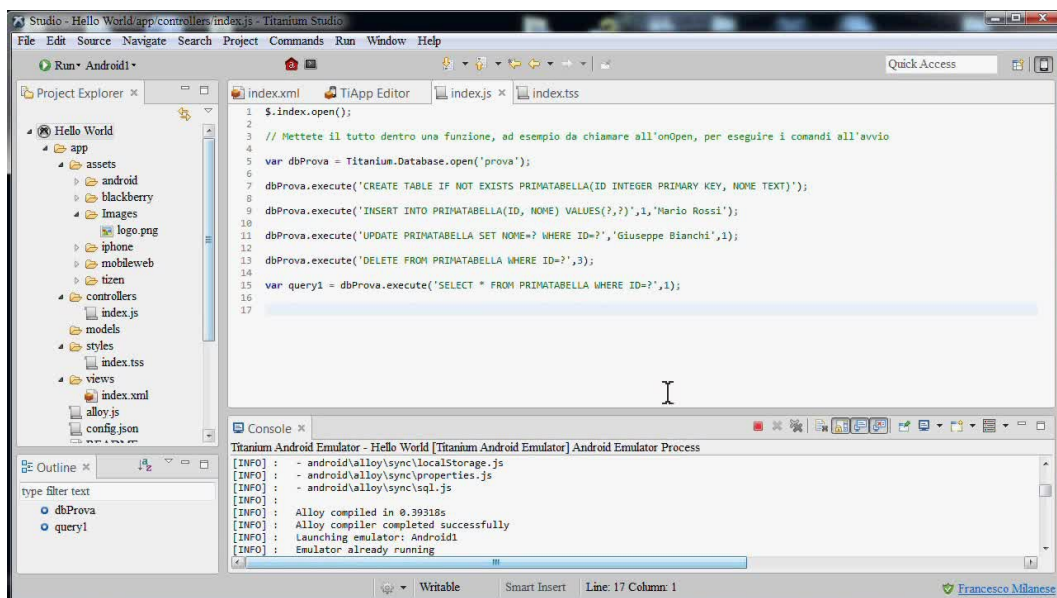
Infine, il recupero dei dati:

```
SELECT campi FROM nometabella WHERE condizione;
```

ad esempio:

```
var query1 = dbProva.execute('SELECT * FROM PRIMATABELLA WHERE ID=?', 1);
```

per cui il risultato dell'interrogazione verrà inserito nella variabile **query1**.



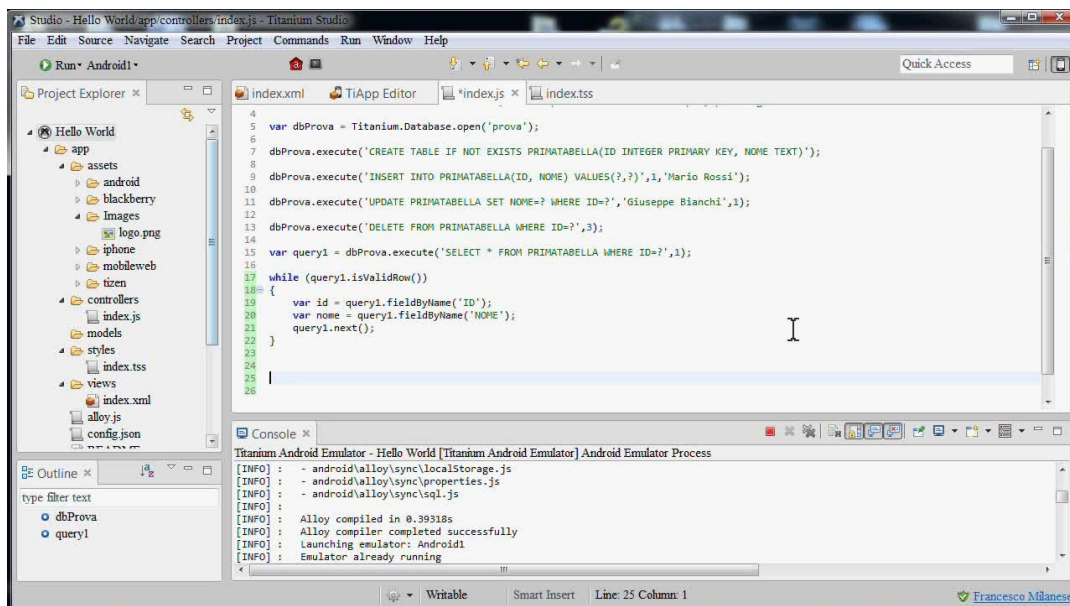
Nel nostro esempio abbiamo richiesto **tutti i campi** (l'asterisco sta per “*ALL*”, quindi “tutti”; in questo caso, ID e NOME).

La **variabile “query1”**, contenente i risultati dell'interrogazione, sarà di tipo **ResultSet**, ossia un **insieme di risultati**; in pratica, un **contenitore** con varie righe e, in ciascuna riga, i vari campi che ci interessano.

Il tutto risulta più semplice con un **esempio pratico**; avremo, ad esempio:

```
var query1 = dbProva.execute('SELECT * FROM PRIMATABELLA WHERE ID=?', 1);
while (query1.isValidRow())
{
    var id = query1.fieldByName('ID');
    var nome = query1.fieldByName('NOME');
    query1.next();
}
```

ossia: fino a quando ci sono **righe valide** (cioè righe di risultati) in **query1**, prendi di ciascuna riga i campi **ID** e **NOME** (**fieldByName**), assegnandoli a delle **variabili temporanee**; poi, vai alla riga successiva (**query1.next()**)... semplice, no?



Quelle appena mostrate sono le **operazioni di base con i database**, le tabelle e i dati; per un elenco completo delle funzioni e delle proprietà **SQLite** rimando alla relativa documentazione online.

CARICARE UN DATABASE SQL PRECOMPILATO

Piccola nota a margine: è anche possibile **recuperare un database SQLite** compatibile preesistente, inserito **in una cartella del progetto**, mediante l'istruzione:

```
var dbDaCaricare = Titanium.Database.install('../dbPrecompilato.db',  
'dbPrecompilato');
```

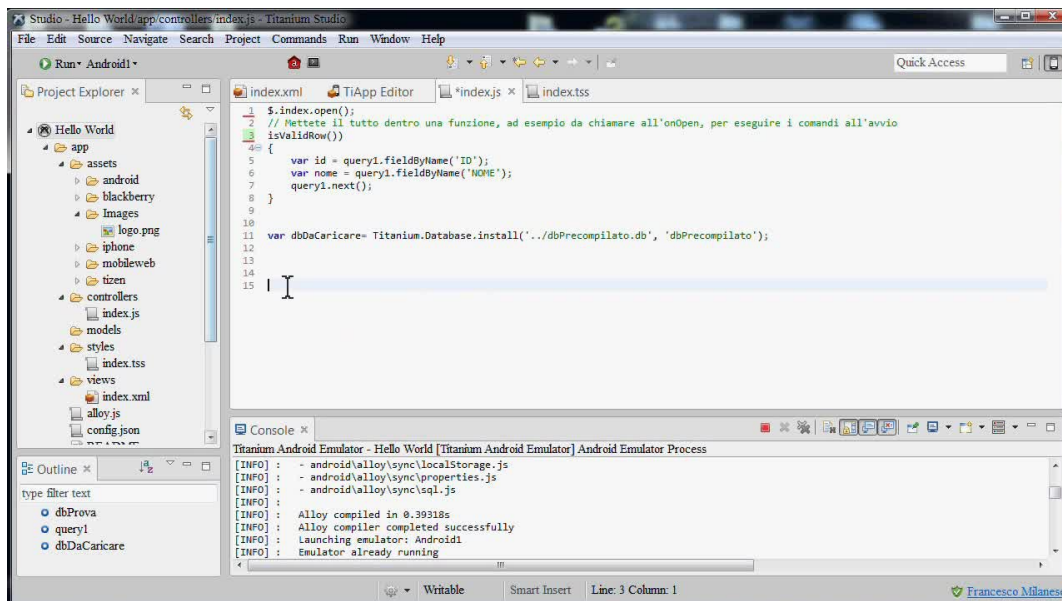


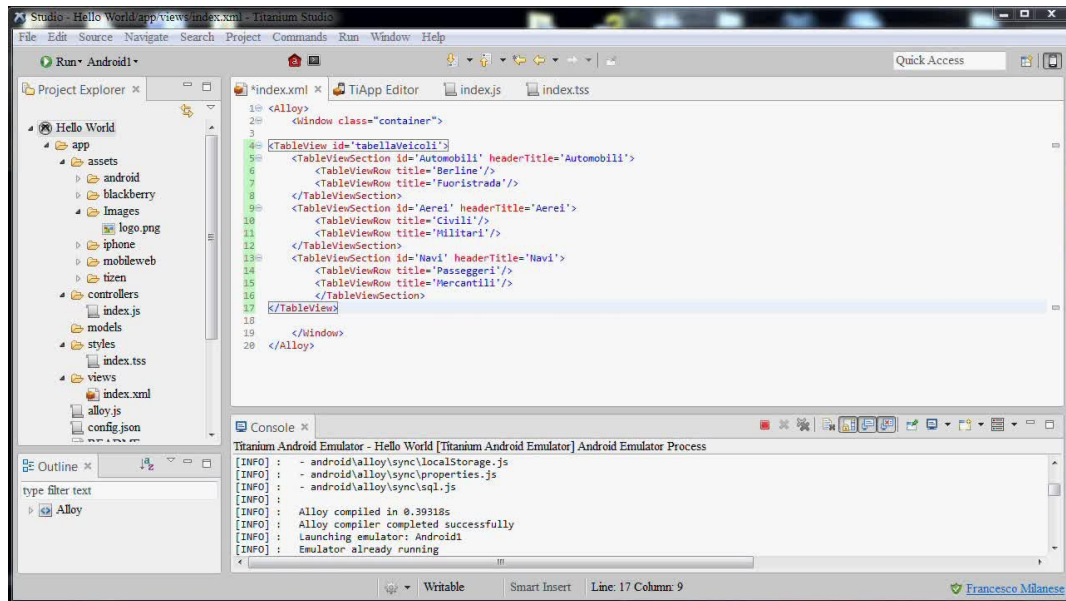
TABELLE STATICHE

La **definizione della struttura in maniera statica** nell'*xml* è indicata quando la tabella deve essere, appunto, statica, cioè **conosciamo già i campi** che dobbiamo contenere e possiamo elencarli nell'*xml*, associando magari ad ogni riga un **evento**, al tocco o click, per eseguire delle azioni.

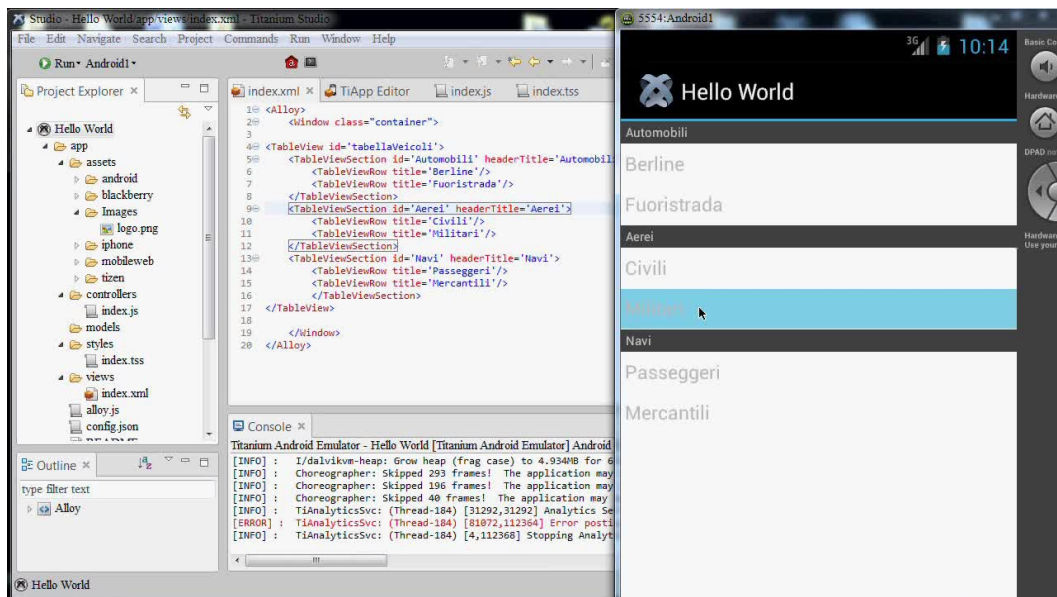
La **sintassi** è semplicissima: i tag che ci interessano sono infatti *TABLE VIEW*, *TABLE VIEW SECTION* e *TABLE VIEW ROW*.

Definiamo ad esempio la seguente **tabella**:

```
<TableView id='tabellaVeicoli'>
  <TableViewSection id='Automobili' headerTitle='Automobili'>
    <TableViewRow title='Berline' />
    <TableViewRow title='Fuoristrada' />
  </TableViewSection>
  <TableViewSection id='Aerei' headerTitle='Aerei'>
    <TableViewRow title='Civili' />
    <TableViewRow title='Militari' />
  </TableViewSection>
  <TableViewSection id='Navi' headerTitle='Navi'>
    <TableViewRow title='Passeggeri' />
    <TableViewRow title='Mercantili' />
  </TableViewSection>
</TableView>
```



Salviamo, compiliamo ed eseguiamo, per vedere il risultato.



Come potete notare, **non sto specificando lo stile TSS** della tabella: **Titanium** gliene darà uno di default, che farà occupare all'elemento lo spazio disponibile nella *View*; ovviamente, non deve essere sempre così, per cui **potete provare a definire degli stili per la TableView**, come esercizio.

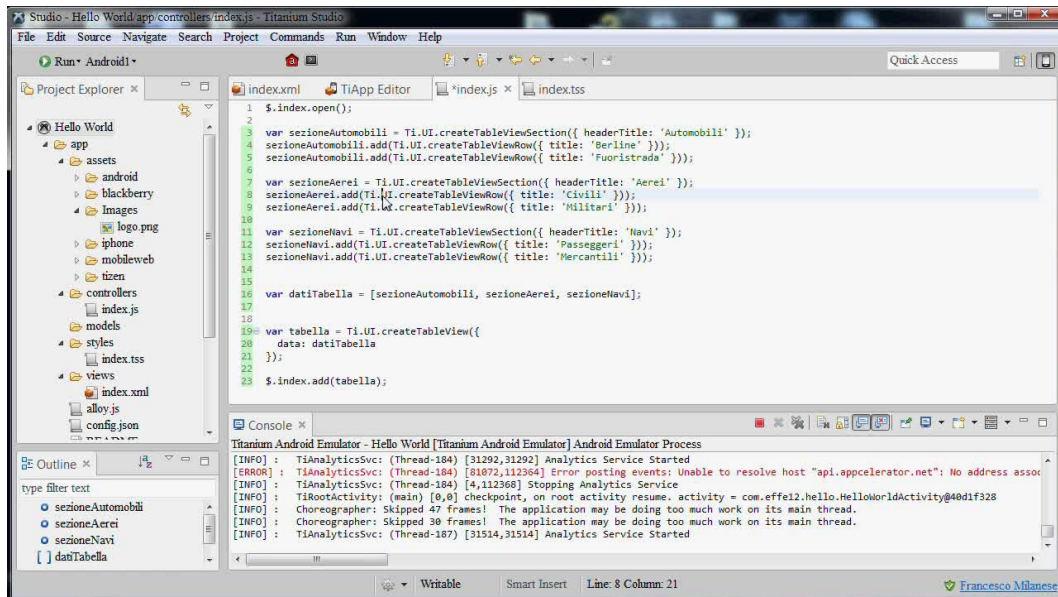
TABELLE DINAMICHE

Con la definizione di una **variabile di dati** possiamo invece creare una tabella **dinamicamente**.

In questo caso si agisce completamente nel **Javascript**, dove bisogna definire la variabile contenente i dati e la tabella (associandole la variabile con i dati), inserendo poi la tabella nella *View*.

La **variabile dei dati** racchiude questi ultimi tra **parentesi graffe**, inserendo tutto l'insieme in **parentesi quadre**... beh, vediamo subito un esempio pratico, riprendendo la tabella statica fatta in xml poco fa:

```
var sezioneAutomobili = Ti.UI.createTableViewSection({ headerTitle :  
  'Automobili' });  
sezioneAutomobili.add(Ti.UI.createTableViewRow({ title : 'Berline' }));  
sezioneAutomobili.add(Ti.UI.createTableViewRow({ title : 'Fuoristrada' }));  
var sezioneAerei = Ti.UI.createTableViewSection({ headerTitle : 'Aerei' });  
sezioneAerei.add(Ti.UI.createTableViewRow({ title : 'Civili' }));  
sezioneAerei.add(Ti.UI.createTableViewRow({ title : 'Militari' }));  
var sezioneNavi = Ti.UI.createTableViewSection({ headerTitle : 'Navi' });  
sezioneNavi.add(Ti.UI.createTableViewRow({ title : 'Passeggeri' }));  
sezioneNavi.add(Ti.UI.createTableViewRow({ title : 'Mercantili' }));  
var datiTabella = [sezioneAutomobili, sezioneAerei, sezioneNavi];  
var tabella = Ti.UI.createTableView({  
  data : datiTabella;  
});  
$.index.add(tabella);
```



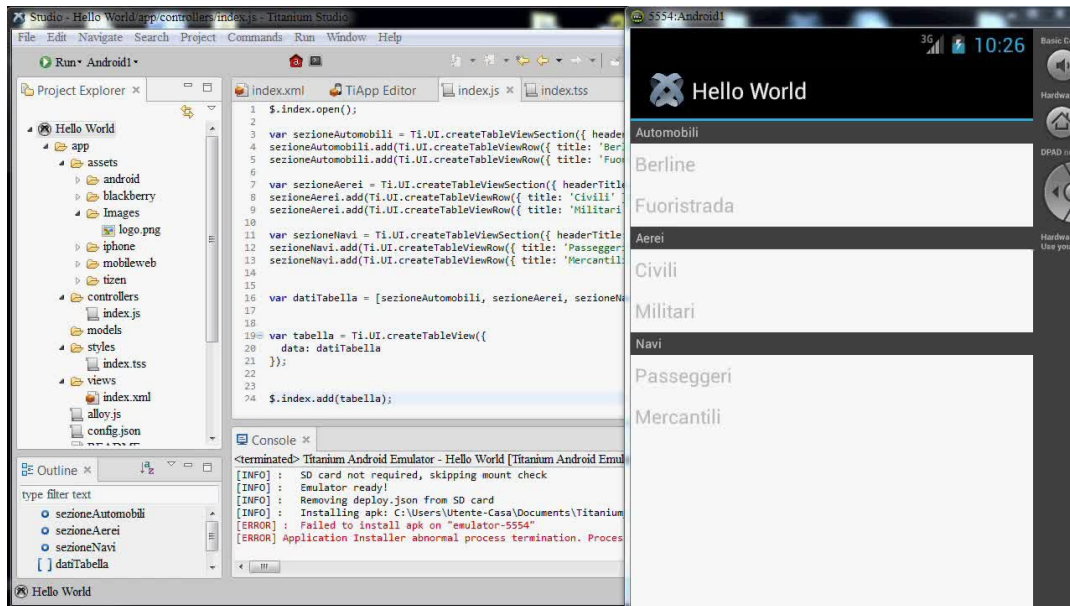
quindi, come è possibile notare analizzando il codice, le **sezioni** (con le relative righe, che verranno incorporate automaticamente grazie all'*add*) vanno **“impacchettate”**, elencandole separate da **virgole tra parentesi quadre**, in una variabile, qui chiamata **“datiTabella”**.

Tale **variabile** verrà poi associata all'**attributo “data”** (i dati, appunto) della **TableView** creata mediante **Ti.UI.createTableView** e associata, a sua volta, ad una **variabile** che la rappresenta, qui chiamata **“tabella”**.

Questa variabile *handler* (gestore) della tabella verrà quindi aggiunta alla schermata con il metodo *add* di *\$.index*.

[**Rivedendo il tutto “al contrario”**: la schermata *index* contiene una *tableView*, creata specificando come “data” un insieme di *TableViewSections* elencate tra *[]*; ciascuna *TableViewSection* contiene delle *TableViewRow*, aggiunte alla sezione con il metodo *add*.]

Salviamo, compiliamo ed eseguiamo il tutto.



Anche per questo capitolo e, anche questa volta, chiudiamo con un **esercizio**: create un semplice db persistente e visualizzatene i dati (da caricare in maniera dinamica, con il ciclo *while*, non con un semplice copia-incolla) in una *TableView*.

* * *

Titanium - Tutorial 6

Windows e Views. Layout TabbedViews. Passare da una Window ad un'altra

Questo capitolo sarà decisamente più leggero rispetto ai precedenti: parleremo infatti della distinzione tra *Windows* e *Views*, del layout *TabbedWindows* e di come passare **da una Window ad un'altra** senza dover ricorrere alle *Tabs*.

Fino a questo momento, infatti, abbiamo esaminato alcuni oggetti di interfaccia, come intercettare alcuni eventi dell'app e anche come creare e interagire con i database, memorizzando i dati su dispositivo; tuttavia, abbiamo esaminato questi aspetti **in una sola schermata**, il che è positivo quando bisogna esaminare solo quelli, per non doversi preoccupare di altri aspetti, ma decisamente riduttivo – se non problematico – quando si deve sviluppare **una vera e propria app...** ma prima facciamo un po' di chiarezza sui termini *View* (vista) e *Window*.

In genere, per indicare una **schermata**, o un ambiente dell'applicazione nell'ambito della programmazione mobile, si usa generalmente il termine **View**, o “**Vista**”, ma in **Titanium** dobbiamo fare **attenzione**: in questo ambiente, infatti, una *View* è sì un contenitore, ma contenuto a sua volta **dentro una Window**, che quindi è il vero e proprio *Top Level container*, che può contenere **una o più Viste**, ma non può essere inserito dentro una di loro e che generalmente occupa tutta la schermata ad eccezione della barra di navigazione, della barra di stato o di una eventuale *Tab Bar*, se si crea un componente *Tab Group*, come vedremo.

Le schermate contenitori sono quindi le **Window**, che possono contenere **una o più View** al loro interno.

Vediamo, per prima cosa, **come definire una View** all'interno di una **Window**, utilizzando ancora una volta **l'applicazione Titanium Alloy di base**.

La nostra **Window** di base è quindi **index** che, come sappiamo, viene aperta automaticamente all'avvio dell'applicazione mediante la **funzione open()** nel file **index.js**, con:

```
$$.index.open();
```

Come per qualsiasi altro tipo di oggetto (e una **View** è un tipo di oggetto, come gli interi o i **ResultSet** dei database), abbiamo una **funzione costruttore nativa di Titanium**, che ci consente di creare una **View** e associarla ad una variabile; tale funzione è:

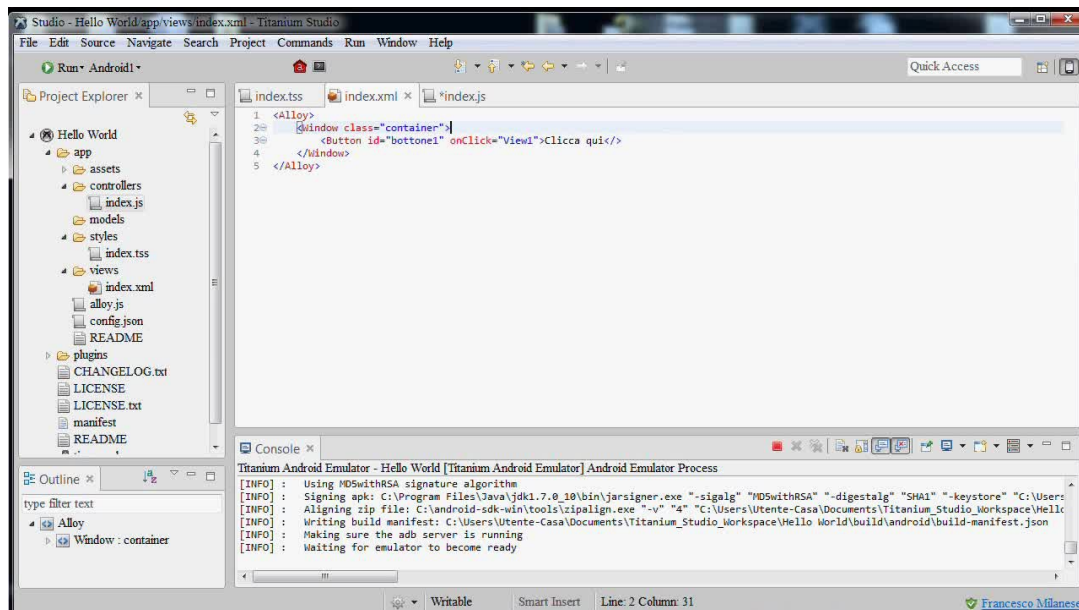
```
Ti.UI.createView
```

e ovviamente non dovrà mancare un **.add(vista)** per la **Window** che la dovrà contenere.

Ecco un esempio pratico: nel file **xml** di **index** scriviamo

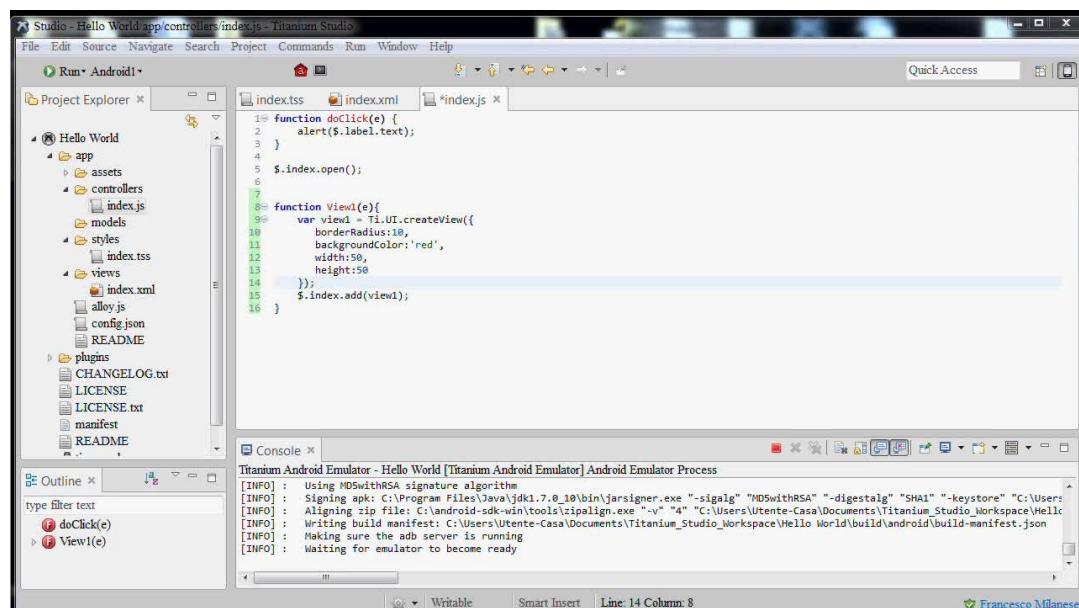
```
<Button id='bottonel' onClick='View1'>Clicca qui</>
```

per creare un **Button** che al **click**, ovvero al tocco, creerà e visualizzerà **una View all'interno della Window** corrente.



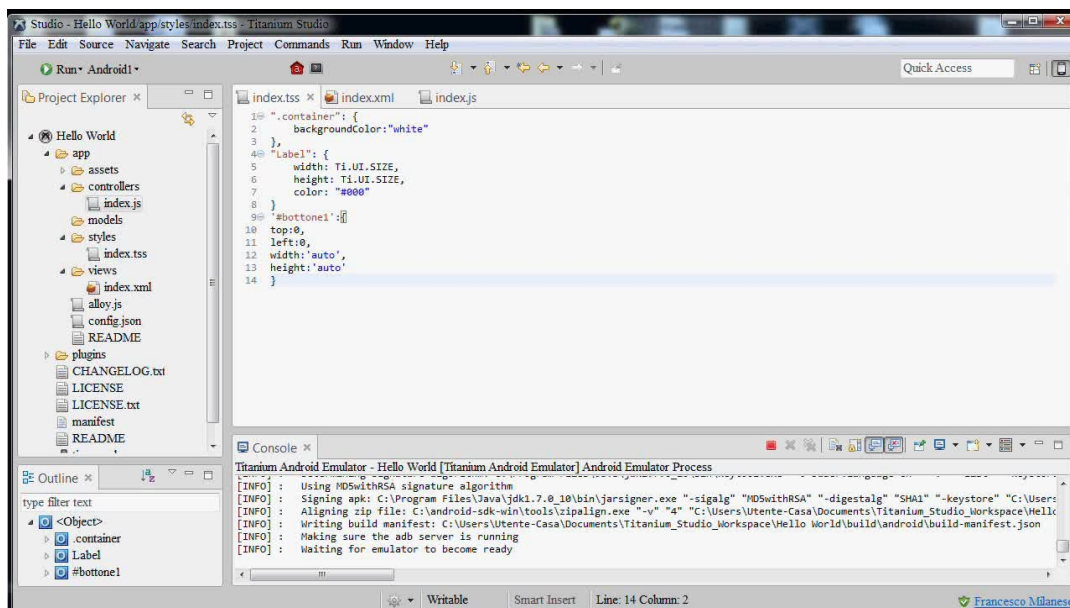
Nel file **Javascript *index.js*** scriviamo quindi:

```
function View1(e) {
    var view1 = Ti.UI.createView({
        borderRadius : 10,
        backgroundColor : 'red',
        width : 50,
        height : 50
    });
    $.index.add(view1);
}
```

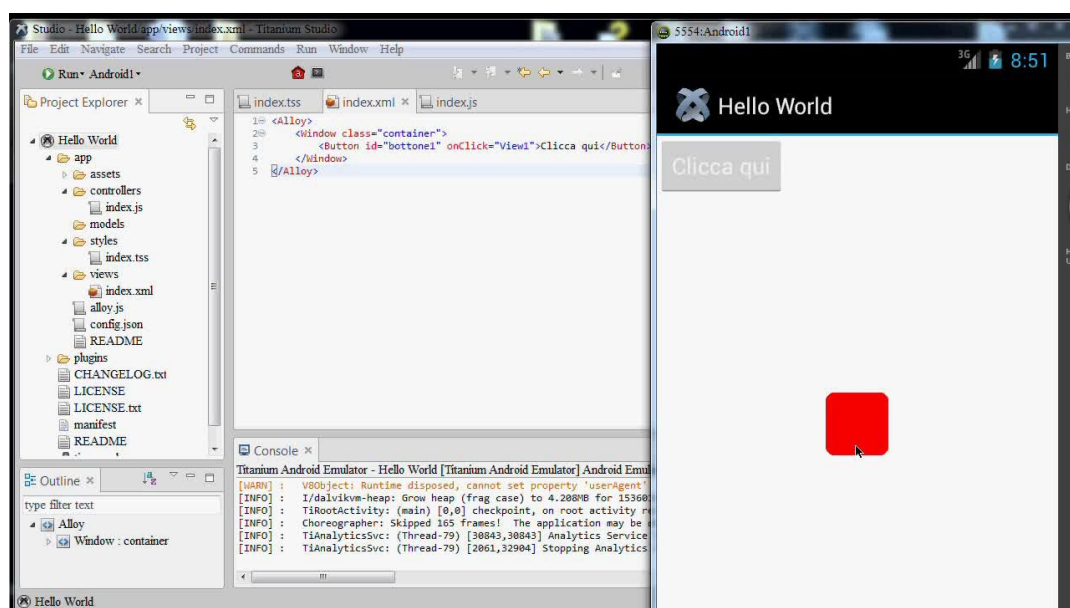


Nel TSS definiamo solo la posizione (in alto a sinistra) per il **bottone**, con:

```
'#bottone1' : {  
    top : 0,  
    left : 0,  
    width : 'auto',  
    height : 'auto'  
}
```



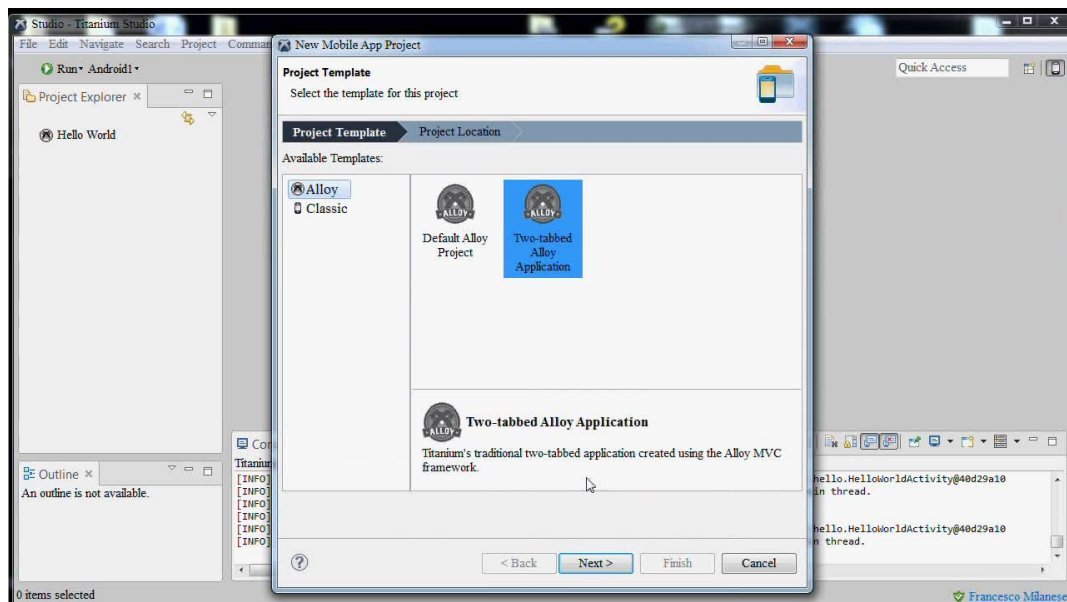
Salviamo, compiliamo ed eseguiamo, per verificare quanto fatto.



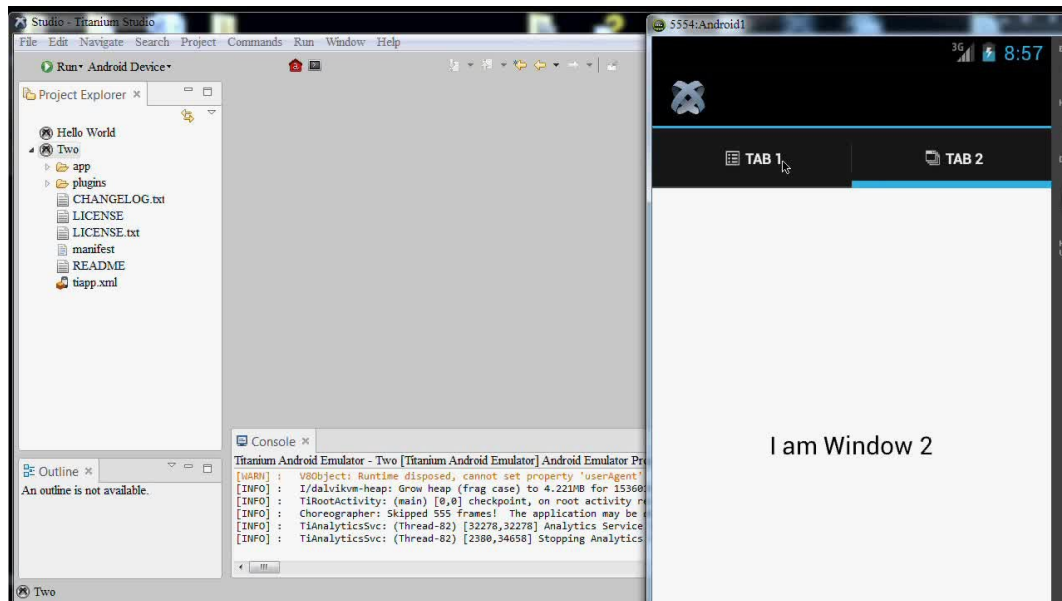
Le **View** di **Titanium** sono quindi come dei **DIV nell'html**: utili per definire aree e per contenere altri **elementi**, con le **coordinate** di questi ultimi da specificare, ad esempio, **relativamente alla View** che li contiene, per comodità e per realizzare interfacce dinamiche.

Vediamo adesso il secondo caso: le **Tabbed Windows**.

Per esaminare questo caso creiamo **un nuovo progetto Titanium con template Alloy**, con *New Mobile App Project*, scegliendo questa volta il tipo **Two Tabbed Alloy Application**, quindi specifichiamo un *nome* e un *id* e confermiamo.



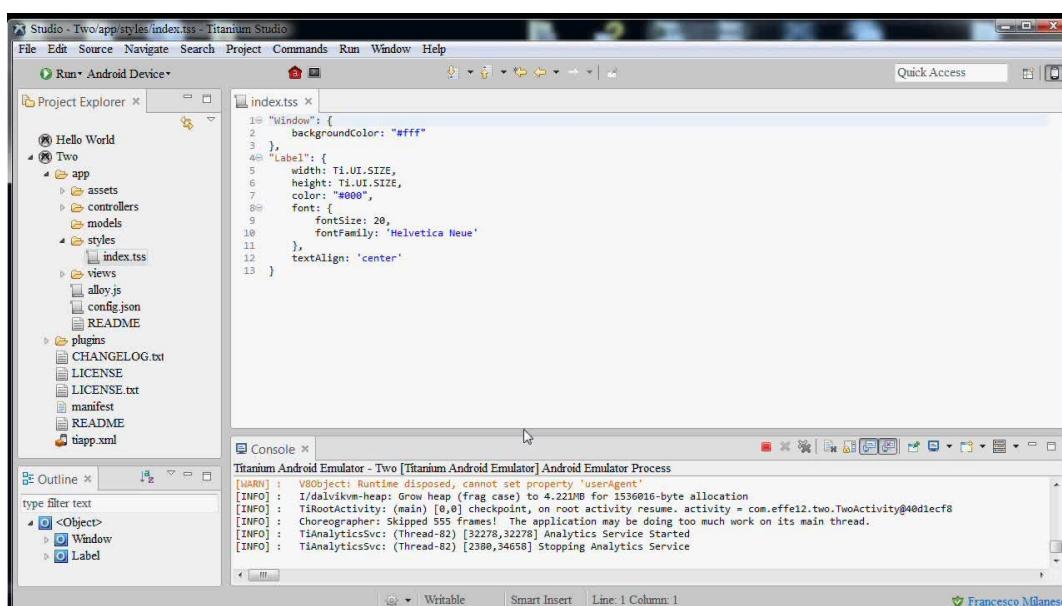
Lanciando l'applicazione vedremo, nell'emulatore, il senso di **“Tabbed”**: le **Window** sono due e sono inserite come schede, delle quali vediamo in alto le etichette.



Cliccando su un'etichetta, la relativa **Tab Window** verrà messa in evidenza rispetto all'altra.

Diamo un'occhiata al **codice**, per vedere la struttura dell'*app* a questo punto.

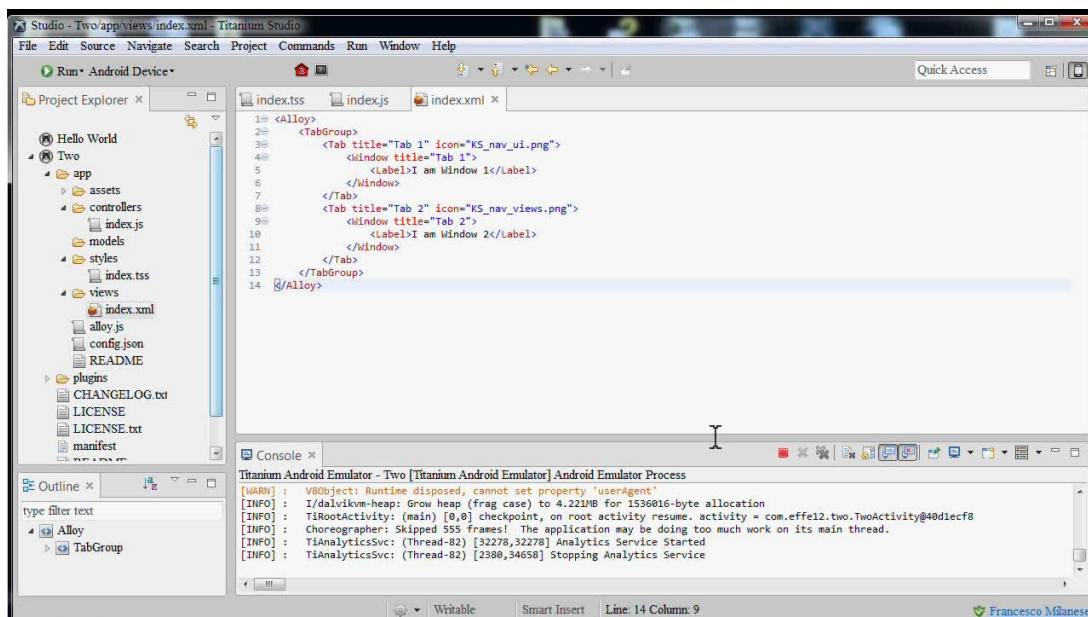
Nel **TSS** notiamo solo la definizione non più di un *container* ma di una **Window**, per la quale stiamo specificando il colore di sfondo.



Abbiamo poi la classica definizione di una **label** (da notare, più che altro, la definizione annidata del font, con **“font :”** e le caratteristiche tra parentesi graffe).

Nel **Javascript**, nulla di nuovo: solo ***\$.index.open()***; ... eppure questa riga di codice ci suggerisce un paio di osservazioni interessanti: la prima è che l'evento del campo **Window**, in seguito a click o tocco sul tab, **non è un evento gestito via codice Javascript**, ma implicitamente dagli **oggetti TAB** nativi di **Titanium**; la seconda è che la “schermata”, per così dire, è sempre una, **“index”** (all'interno della quale possiamo avere più **TABs**, è vero, ma è **index** ad avere il metodo **open()**, invocato automaticamente all'avvio dell'applicazione).

Le novità più interessanti si trovano, invece, nel file **xml**: qui notiamo la presenza di **due nuovi oggetti, TabGroup e Tab**, che arricchiscono la struttura dell'**xml** standard composto, in origine, da **Alloy, Window** e gli **elementi grafici**.



Adesso infatti abbiamo **Alloy, TabGroup** e, all'interno di questo “gruppo”, **più TABs** (in questo caso due, ciascuna delle quali contiene finalmente gli elementi che conosciamo, ossia **Window** e gli **elementi grafici** a seguire)... nulla di particolarmente complicato, quindi: solo l'aver **racchiuso più Window in più TABs dentro un TabGroup**.

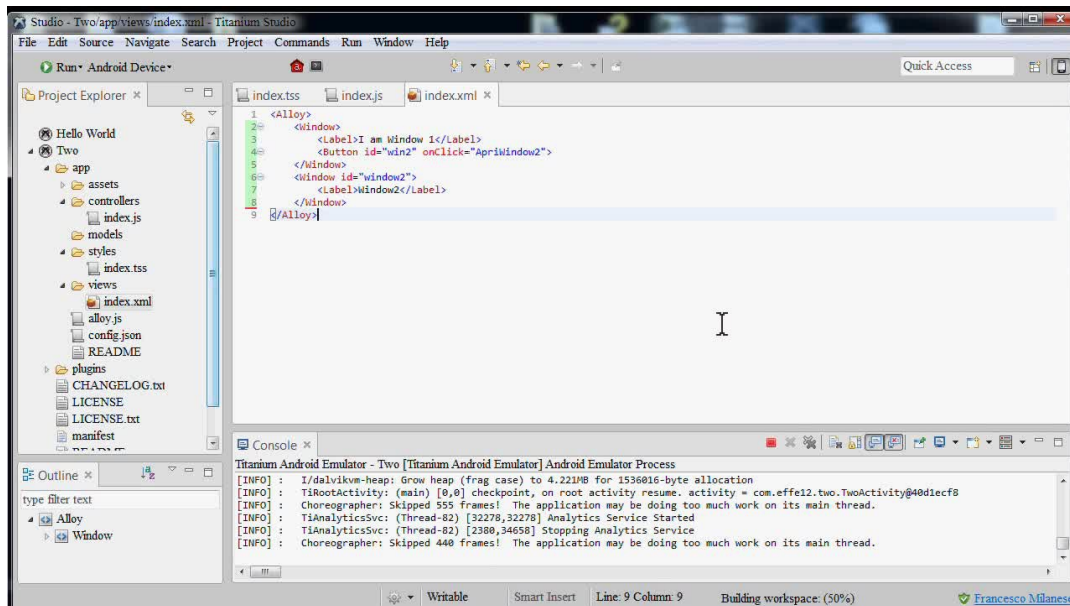
Gli oggetti **TAB** possono avere solo un elemento **figlio**, che deve essere di tipo **Window**, quindi **non possiamo definire più oggetti Window dentro un Tab**.

A questo punto però potremmo chiederci: ma non c'è proprio **un metodo per definire più Window**, in un'app, in modo tale da separare le schermate, **senza dover ricorrere alle TABs**, anche per una questione di praticità di sviluppo, nel senso che così l'*xml* e il *tss* delle varie schermate potranno prepararli vari sviluppatori in parallelo, nonché per una migliore comprensione del codice, ad esempio se si deve sviluppare un codice con molte schermate di tipo diverso?

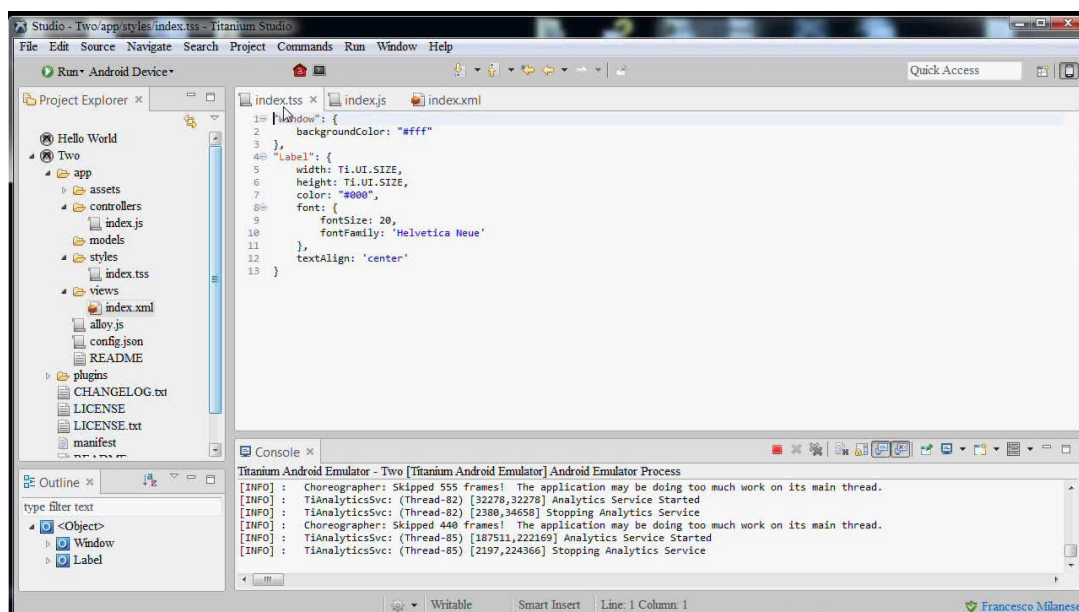
Sì che c'è: è sufficiente **definire nell'*xml* più Windows**, avendo cura di lasciare la prima, quella iniziale, **senza *id***, in modo tale da farla riconoscere come “*index*” da **Titanium**, per aprirla all'avvio dell'app, mettendo **gli *id* nelle altre**, inserendo queste *Window* direttamente in **Alloy**, senza *TabGroup* e *Tab*, e definendo poi dei metodi **Javascript** per definire l'*open* delle varie *Window*, in base a eventi o azioni.

Ridefiniamo quindi il **contenuto di *index.xml*** come segue:

```
<Alloy>
  <Window>
    <Label>I am Window 1</Label>
    <Button id='win2' onClick='ApriWindow2'>
  </Window>
  <Window id='window2'>
    <Label>Window2</Label>
  </Window>
</Alloy>
```



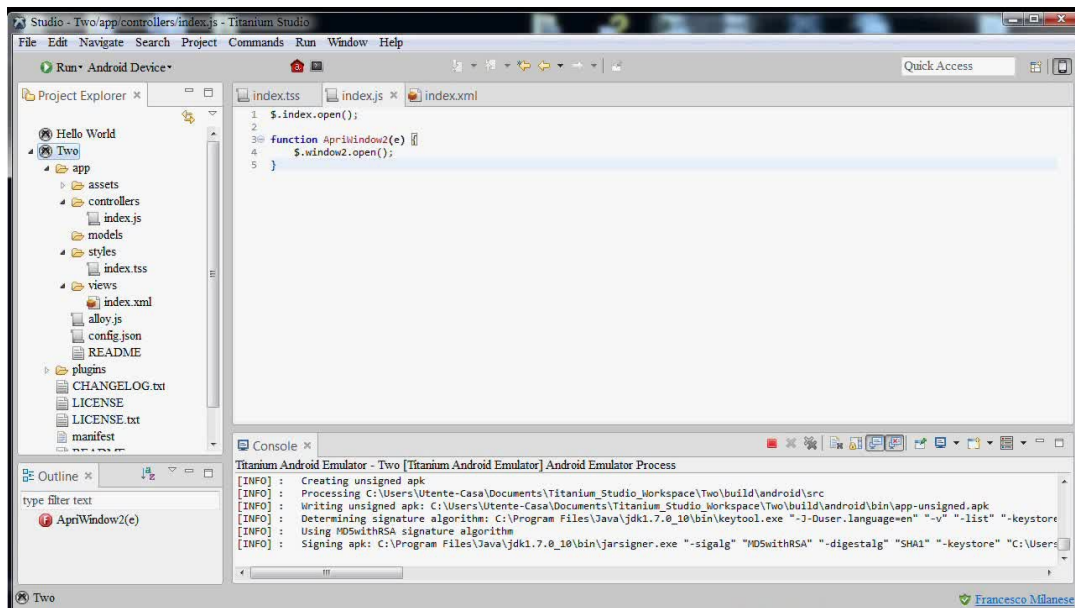
Lasciamo **inalterato** il **TSS** originale del template...



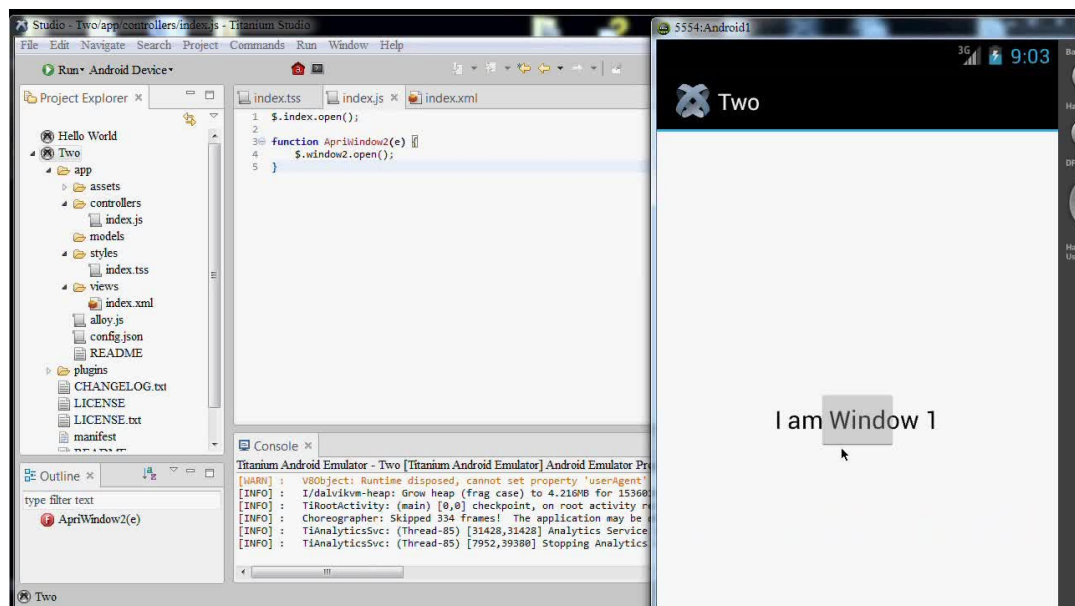
... e ridefiniamo invece il contenuto di **index.js** come segue:

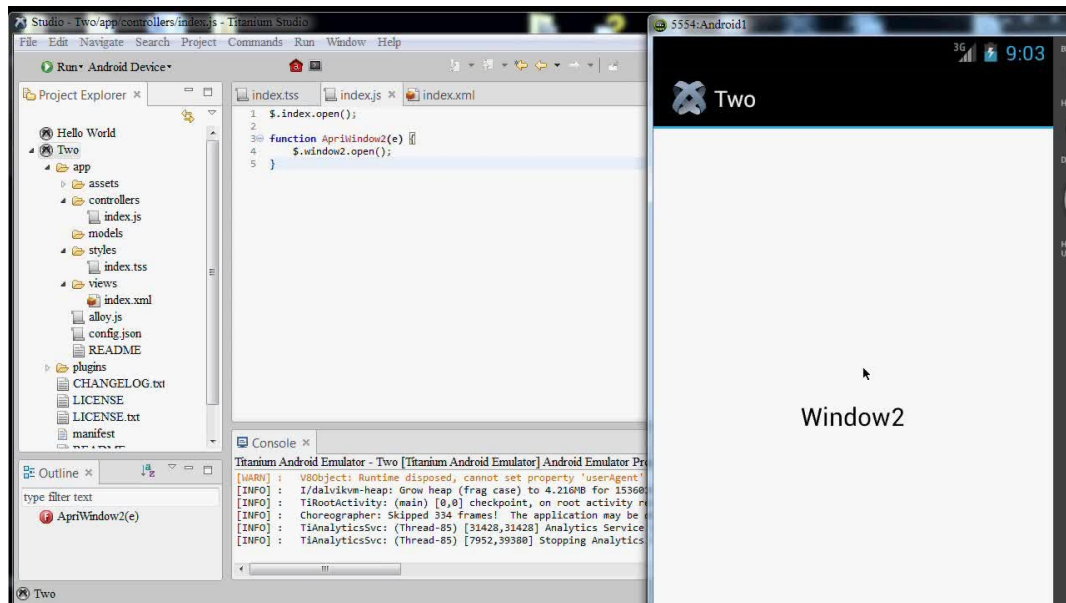
```
$.index.open();

function ApriWindow2(e) {
  $.window2.open();
}
```



Salviamo, compiliamo ed eseguiamo il tutto.





Per mostrarvi, quindi, **come definire più Windows**, ho seguito una strada un po' “strana”, partendo da un template nativo più complesso e “spogliandolo” di alcuni elementi per arrivare al caso base, ma l'ho fatto proprio per sfruttare da subito un servizio messo a disposizione da **Titanium** e semplificandolo, anziché costruirlo arrivandoci passo dopo passo, con un percorso più difficile.

* * *

Titanium - Tutorial 7

Rilevare le Gestures (Shake, Swipe, Touch, Pinch) e i dati dell'accelerometro

In questo capitolo vedremo **come rilevare le *Gestures* (*Shake, Swipe, Touch, Pinch*, pressione prolungata e dati in arrivo dall'*accelerometro*) in **Titanium**.**

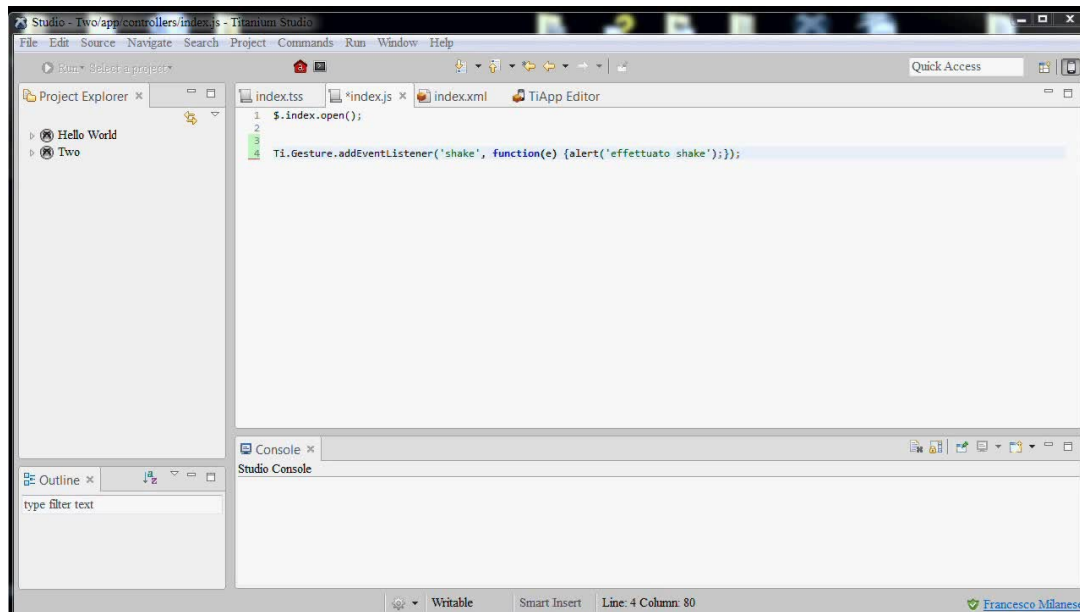
In generale, **il rilevamento avviene aggiungendo l'evento** che ci interessa rilevare ad un ***EventListener***, cioè letteralmente un ascoltatore di eventi, un oggetto di **Titanium** che rileva l'elemento specificato ed esegue del codice, con ***Ti.Gesture.EventListener*** e, come argomenti, il nome del gesto tra apici e la funzione da chiamare per effettuare le operazioni volute.

SHAKE

Ad esempio, per rilevare lo ***Shake*** (il movimento che si ha agitando – *shaking*, appunto – il dispositivo) scriveremo:

```
Ti.Gesture.addEventListener('shake', function(e) {alert('effettuato  
shake');});
```

quindi come potete vedere ho definito il **secondo argomento**, ovvero la **funzione da invocare**, “in linea”, all'interno di ***addEventListener***, in quanto tale funzione riguardava solo l'esecuzione di un *alert*, quindi ho definito tutto lì, ma naturalmente per progetti più complessi potrete invocare funzioni definite esternamente.



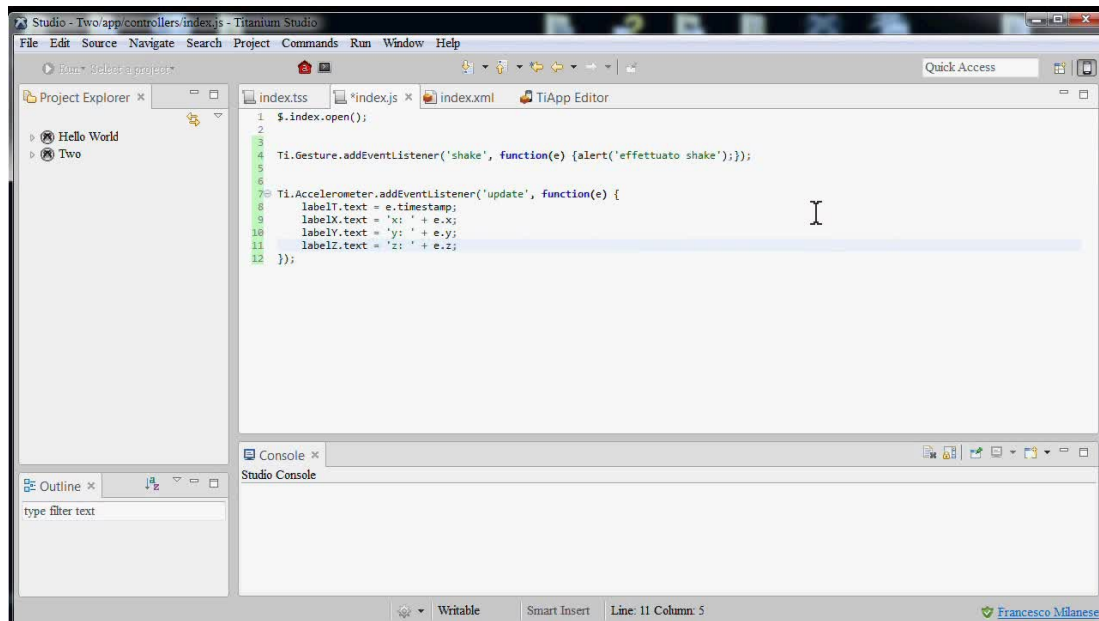
Purtroppo, l'emulatore **Android** non implementa l'emulazione dello *Shake*.

ACCELEROMETRO

Per rilevare invece movimento del dispositivo mediante l'**accelerometro** (anche questo non simulato, purtroppo, dall'emulatore Android) si deve ricorrere al **modulo Accelerometer**, anziché **Gesture**, dopodiché sarà possibile rilevare informazioni sul momento in cui è stato lanciato l'evento e sul movimento sui tre assi XYZ mediante i **campi timestamp, x, y e z** di “*e*”, rispettivamente, con “*e*” che è la variabile che passiamo come parametro alla funzione da invocare; ad esempio:

```
Ti.Accelerometer.addEventListener('update', function(e) {
    labelT.text = e.timestamp;
    labelX.text = 'x: ' + e.x;
    labelY.text = 'y: ' + e.y;
    labelZ.text = 'z: ' + e.z;
});
```

con *labelT*, *labelX*, *labelY* e *labelZ* che sono, come suggeriscono i nomi, delle *label*, che quindi dovrete definire nell'*xml* o anche via codice, per visualizzare in qualche modo i dati, se volete appunto vederli per farvi un'idea di quali valori restituiranno, per poterli poi utilizzare confrontandoli con altri parametri o inserendoli in operazioni nel resto del codice.



Dico di usare le *label* perché non ha senso utilizzare gli *Alert*, come con lo *Shake*, perché verrebbero **lanciati di continuo**; la *label* verrà semplicemente **aggiornata**, mostrando il valore in quel momento.

Attenzione a separare, nella *function*, *labelT* e le altre istruzioni con **punto e virgola**, non con virgole, in quanto non si tratta di parametri ma del vero e proprio **blocco di istruzioni** che compongono la funzione, qui definita “in linea” all'interno di ***addEventListener***.

GESTURES

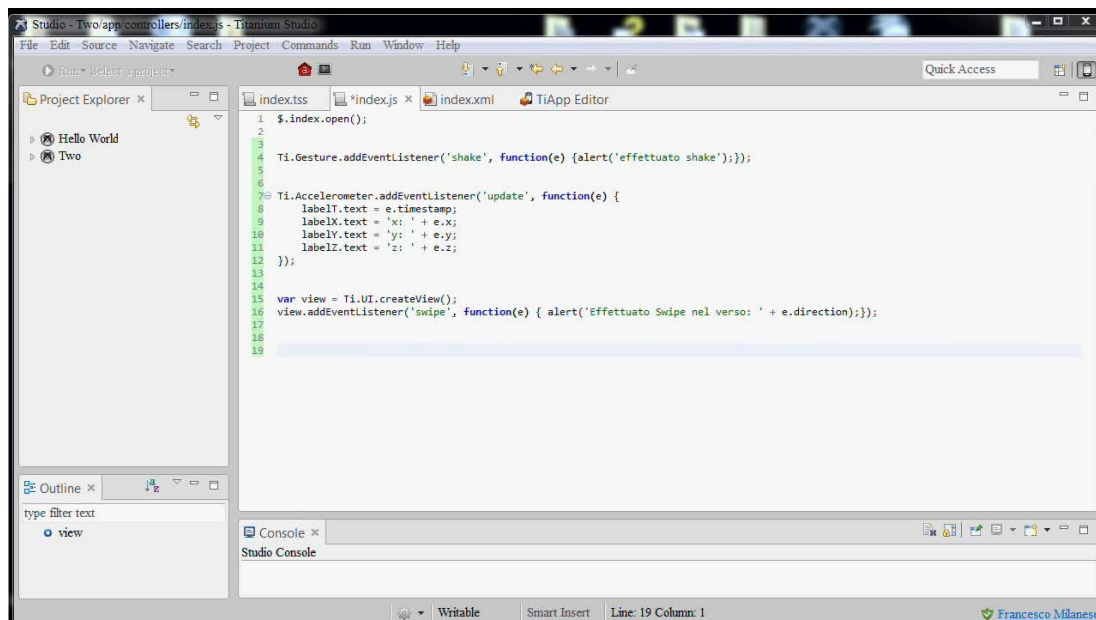
Lo *Shake* e l'accelerometro riguardano più che altro movimenti che hanno a che fare con l'intero dispositivo (nel senso hardware), mentre le **Gestures**, che stiamo per esaminare, si riferiscono ad **interazioni con il display del dispositivo**, quindi mediante tocchi e trascinamenti delle dita.

SWIPE (PAN)

Lo ***Swipe***, ossia “tocco e strisciata” del dito (detto anche ***Pan***), può essere rilevato sulla maggior parte dei **componenti di interfaccia** di **Titanium**, nel senso che potete fare uno **Swipe** su un'immagine, un *Button* o una generica *View*, come quella vista nella puntata precedente...

... anzi, vediamo proprio questo caso, scrivendo nel file **Javascript**:

```
var view = Ti.UI.createView();
view.addEventListener('swipe', function(e) { alert('Effettuato Swipe
nel verso: ' + e.direction);});
```



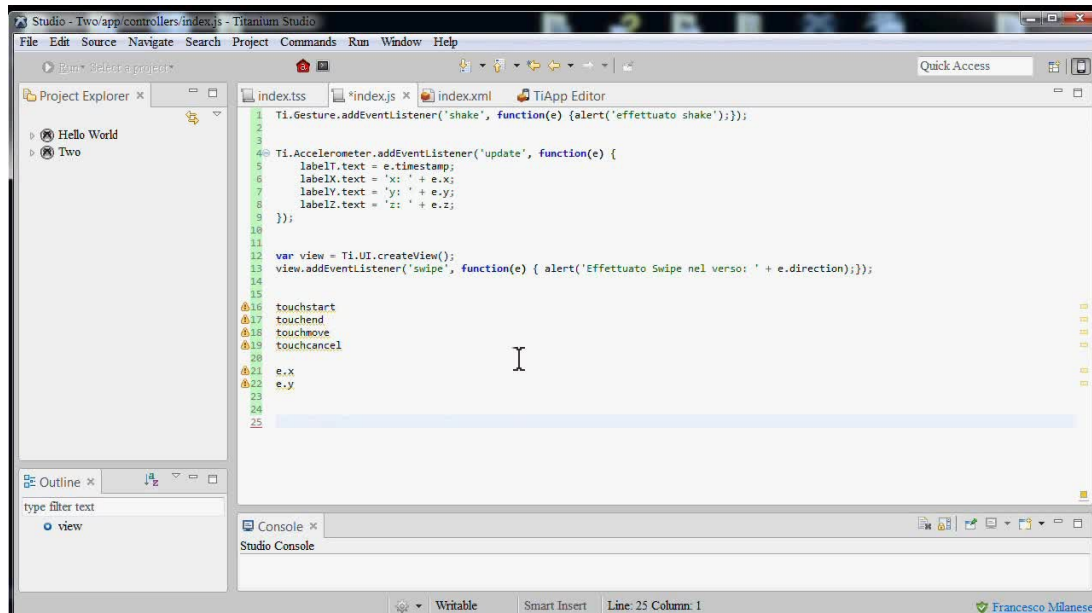
EVENTI E COORDINATE DEL TOCCO

Parlare del **Tocco** può sembrare superfluo, visto che abbiamo definito diverse volte gli eventi **onClick** sugli elementi di interfaccia delle app, ma non è così: gli **eventi Touch** sono infatti ben **quattro** e per ciascuno di essi è possibile rilevare le **coordinate X e Y** in riferimento all'interfaccia grafica dell'applicazione, per capire dove l'utente ha effettuato il tocco.

Questi quattro eventi sono: **TouchStart** (inizio del contatto tra dito e display), **TouchEnd** (che avviene quando l'utente solleva il dito dal display), **TouchMove** (si ha se l'utente muove il dito tenendolo poggiato sul display) e **TouchCancel** (lanciato dal sistema se l'evento del tocco deve essere annullato, cosa che avviene quando ad esempio qualcuno vi telefona – e vi arriva cioè una chiamata – proprio quando state facendo un tocco su un punto dello schermo).

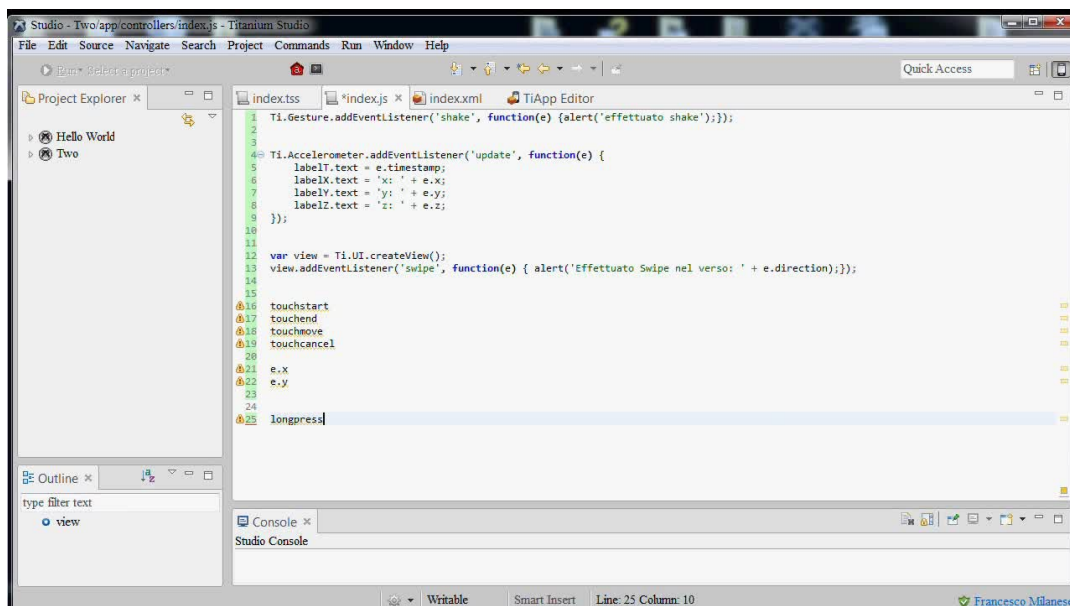
Gli eventi da far rilevare sono quindi questi quattro, così come li ho nominati, mentre **per ricavare i valori x e y dell'evento** dovreste accedere a questi campi “posizione” dell'evento, ossia della

variabile **“e”** presente tra le parentesi tonde della funzione, con **e.x** e **e.y**, così come abbiamo fatto ad esempio un attimo fa con *Swipe*, per rilevare la **direzione del movimento** con **e.direction**.



LONGPRESS

Molto simile agli eventi *Touch* è il **LongPress**, la pressione prolungata, effettuata in genere quando, toccando a lungo un'icona o un altro oggetto di interfaccia, viene mostrato a video un *Alert*, un menù di opzioni o, in generale, qualche altro elemento di interfaccia, per consentirci di scegliere opzioni o effettuare azioni specifiche sull'elemento toccato.



In questo caso, l'evento da far registrare all'*Event Listener* è appunto **“longpress”**, tra apici, tutto in minuscolo e attaccato. Anche per questo evento, come per i tocchi, è possibile ricavare le **coordinate x e y** dell'evento, con *e.x* e *e.y*.

PINCH

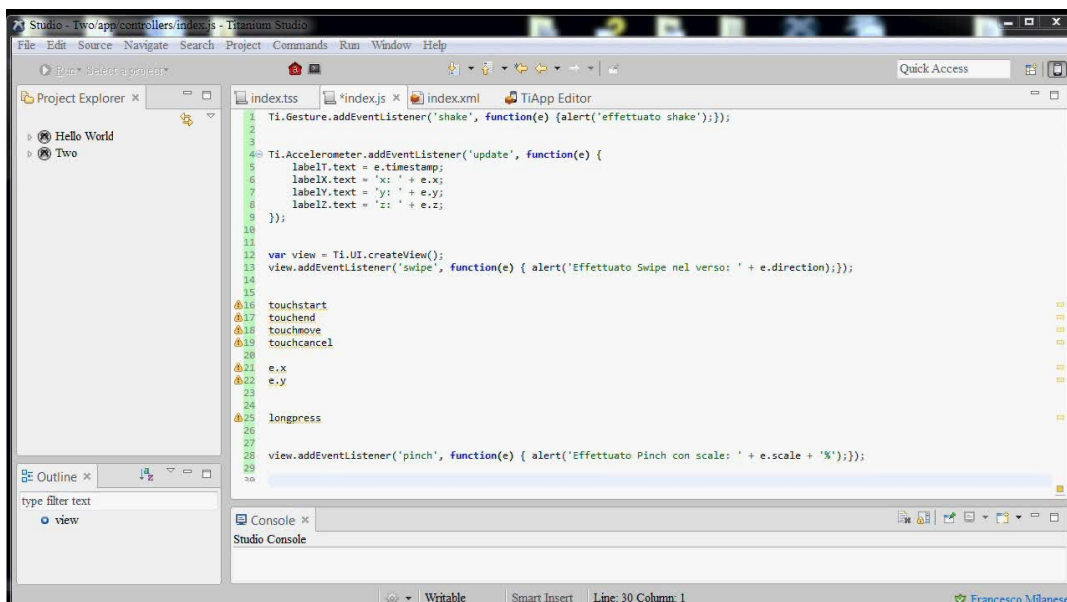
Il **Pinch** è l'evento che si ha quando vengono poggiate contemporaneamente **due dita sul display** e le si allontana o le si avvicina, tenendole premute sul dispositivo; in pratica, quello che succede quando si fa uno **zoom in avanti o indietro** su un'immagine.

In questo caso l'evento da rilevare si chiama appunto **Pinch**, mentre il parametro che ci interessa, nell'evento “e”, si chiama **“scale”**.

Avremo ad esempio (sempre utilizzando la **var view** definita poco fa per lo *Swipe*):

```
view.addEventListener('pinch', function(e) { alert('Effettuato Pinch con scale: ' + e.scale + '%');});
```

mettendo una **% nel testo** perché il valore di **scale** è espresso, appunto, **in percentuale**.



* * *

Titanium - Tutorial 8

Aprire una pagina web dall'app. Utilizzare i Moduli. Inserire delle Mappe nell'app

Nella prima parte di questo capitolo vedremo **come aprire una pagina web** dalla nostra app e, nella seconda parte, come aggiungere un **modulo** (o “*add-on*”) di **Titanium** per inserire le **mappe** nella nostra app.

Vedremo, in particolare, due approcci diversi.

Il primo, quello che useremo **per aprire il web browser**, è quello utilizzato con gli altri elementi, ossia **aggiungendo un elemento *Ti.UI***, nativo di **Titanium** (e anche per questo vi invito ad esplorare la documentazione online, per scoprire gli **altri servizi nativi** messi a disposizione da **Titanium**).

Il secondo approccio consiste nell'**importare un modulo** (o *plugin*, o *add-on*) aggiuntivo che, nella **versione 3.2 di Titanium** (quella con la quale ho realizzato questi tutorial) è fornito **di serie** col programma, per creare un **oggetto *Map*** e invocarne i metodi.

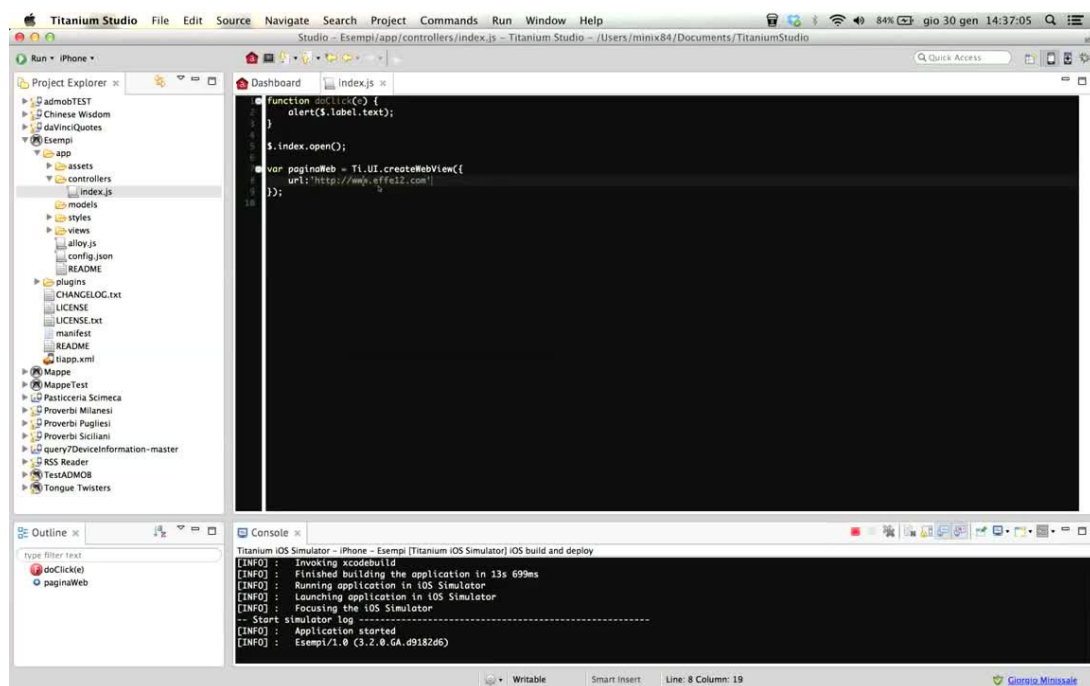
Iniziamo con il primo caso: **la chiamata al web browser**.

L'apertura di una pagina web può essere utile, ad esempio, per aprire la nostra home page se l'utente, utilizzando l'app, clicca (o fa un **Touch**, in effetti) su un **link** o un **Button** che rimanda al nostro sito, tipo un tasto **“Chi siamo”** o **“Contatti”**.

In **Titanium** una pagina web browser è una **View**, quindi un elemento di interfaccia utente, al pari di una **ImageView** o di una **TableView**, per cui va creata con **create** seguito, questa volta, da **WebView**.

Avremo, ad esempio:

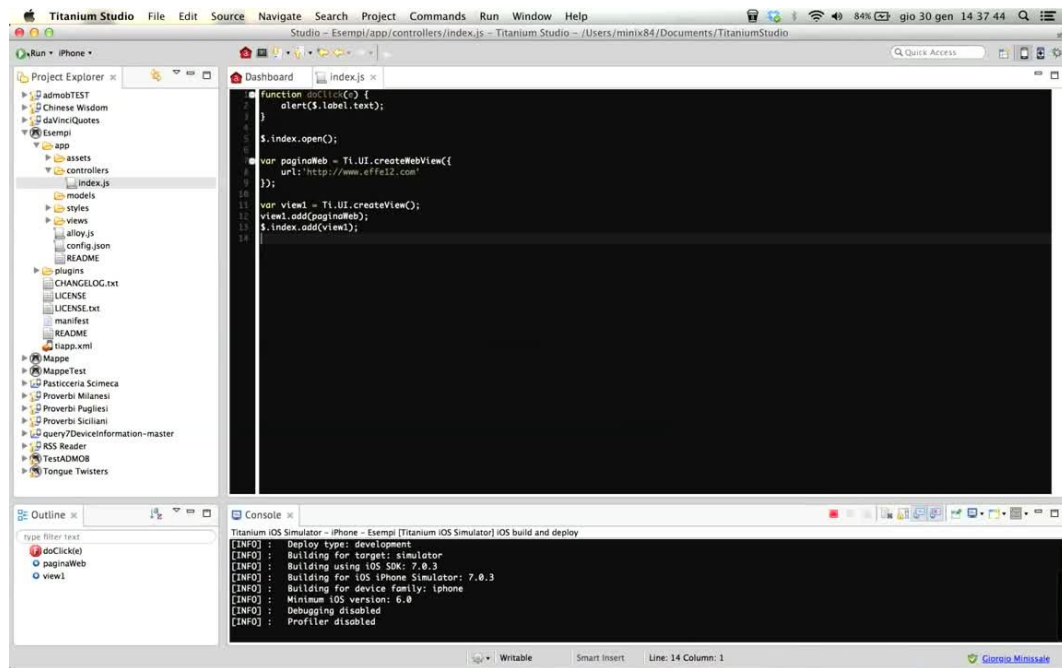
```
var paginaWeb = Ti.UI.createWebView({url: 'http://www.effe12.com'});
```



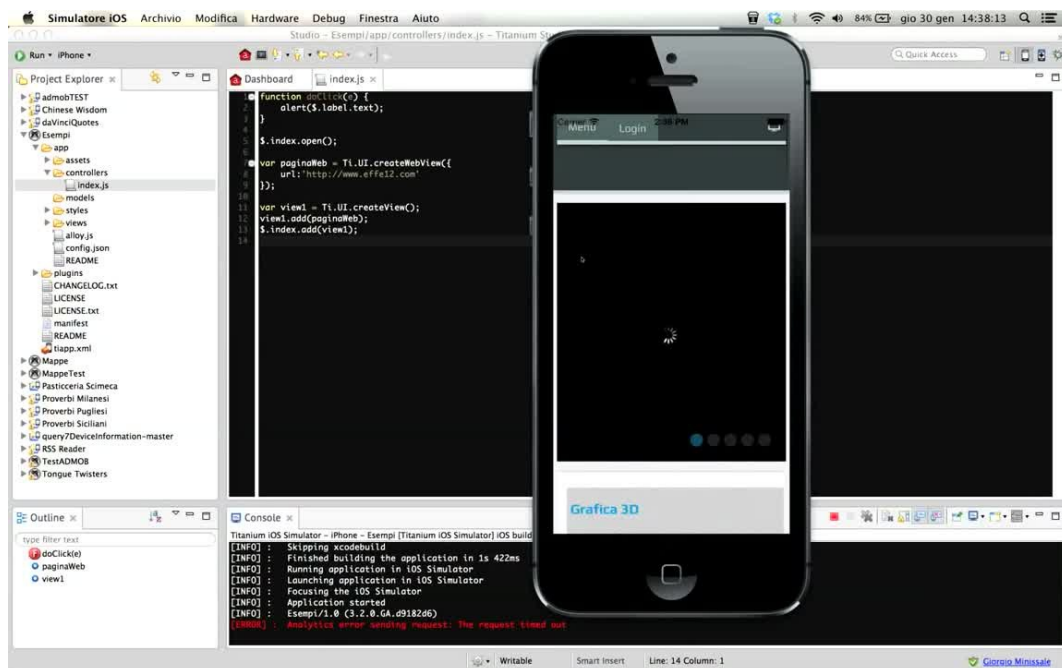
Questo oggetto andrà quindi aggiunto alla **Window** nella quale vogliamo visualizzarla o ad un'altra **View**, ad esempio con:

```
var view1 = Ti.UI.createView();  
view1.add(paginaWeb);  
$.index.add(view1);
```

e non dimentichiamo ovviamente l'**open()** sulla **Window index** da visualizzare.



Salviamo, compiliamo ed eseguiamo, per verificare quanto fatto.



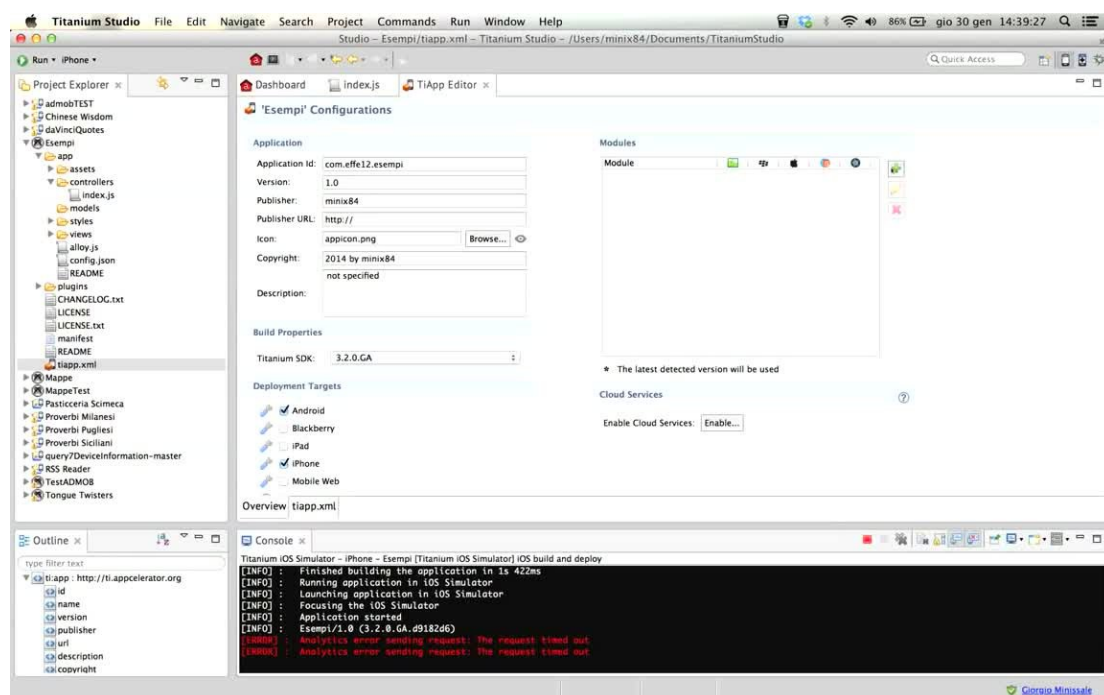
* * *

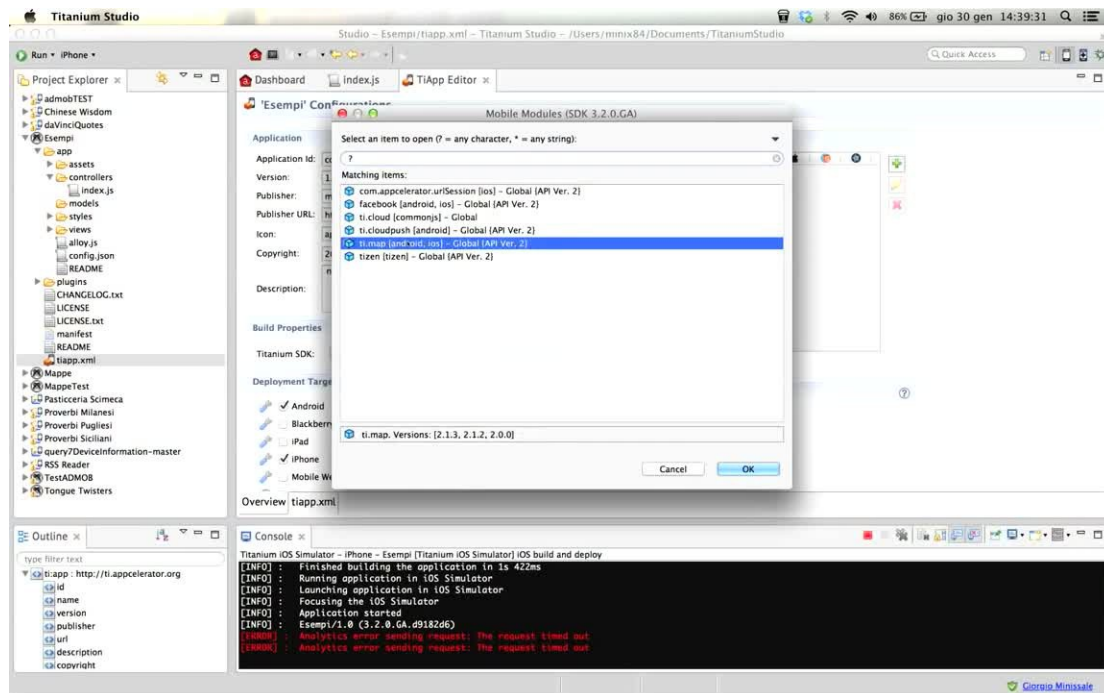
Passiamo quindi alla seconda parte del tutorial, dedicato al **modulo Map** per aggiungere, appunto, una **mappa** alla nostra app... prima, però, un paio di **precisazioni**.

Al momento attuale il modulo **Map** richiede, per poter essere eseguito su dispositivi **iOS**, solo **xCode 5** o superiore, mentre per poter essere eseguito su **Android** sono necessarie le **Google Map API Keys**, più l'installazione del *Google Play Service SDK* mediante l'*Android SDK Manager* e, in ogni caso, funziona solo sui **dispositivi fisici Android**, non nell'emulatore, per cui qui tecnicamente non potrei mostrarvelo (ed ecco perché, in questa tutorial, sto lavorando su un Mac e sto utilizzando il **simulatore iOS** per provare le app).

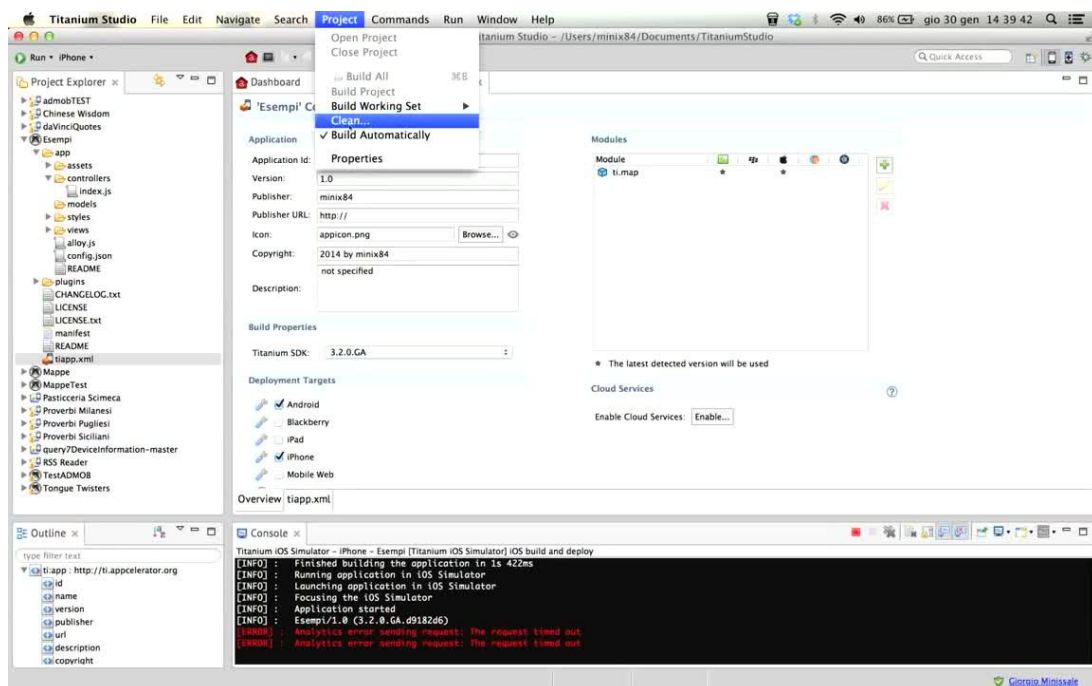
Come detto all'inizio del tutorial, da **Titanium 3.2** (quello che sto utilizzando) in poi, per poter aggiungere le **mappe** bisogna ricorrere ad un **modulo esterno**, per cui ne approfittiamo anche per vedere **come aggiungere moduli e add-ons** per implementare funzionalità extra.

Andiamo nella **finestra TiApp.xml**, clicchiamo sul + accanto all'elenco dei moduli a destra, quindi scegliamo il **modulo Ti.Map** dall'elenco che apparirà.



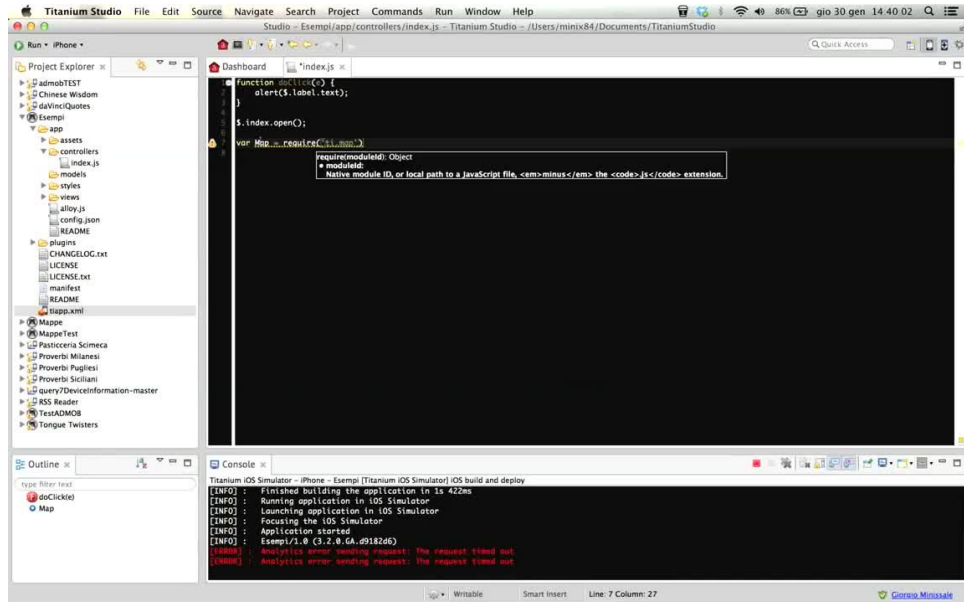


A questo punto può essere utile, dopo aver salvato tutto, cliccare sul **menù Project** e scegliere **Clean Project**, per ripulire il progetto corrente e fargli includere il nuovo **modulo**.



A questo punto nel codice **Javascript** dovremo **istanziare un oggetto *Map***, con:

```
var Map = require('ti.map');
```



Utilizzeremo quindi questo **nuovo oggetto *Map*** per accedere a **campi e metodi del modulo *Map***.

In particolare, ci interessa **recuperare la View della mappa**, specificando alcuni parametri, per poi inserire questa **View** nella nostra **Window**.

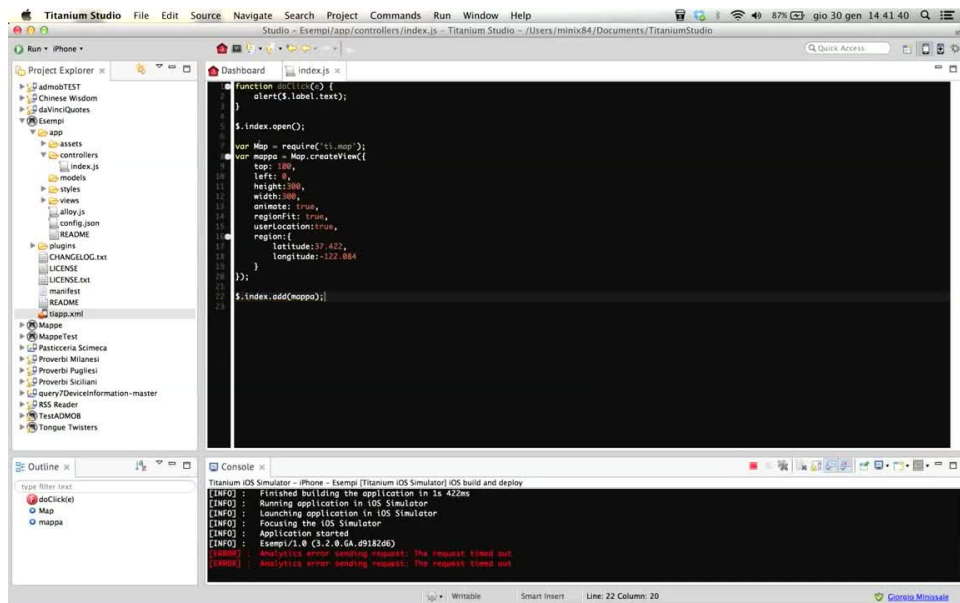
Per ottenere ad esempio una **View** chiamata **“mappa”** (dall'oggetto ***Map***, creato un attimo fa) che rappresenta il **modulo *Ti.Map* importato**, scriveremo:

```
var mappa = Map.createView({
  top : 100,
  left : 0,
  height : 300,
  width : 300,
  animate : true,
  regionFit : true,
  userLocation : true,
  region : {
    latitude : 37.422,
    longitude : -122.084
  }
});
```

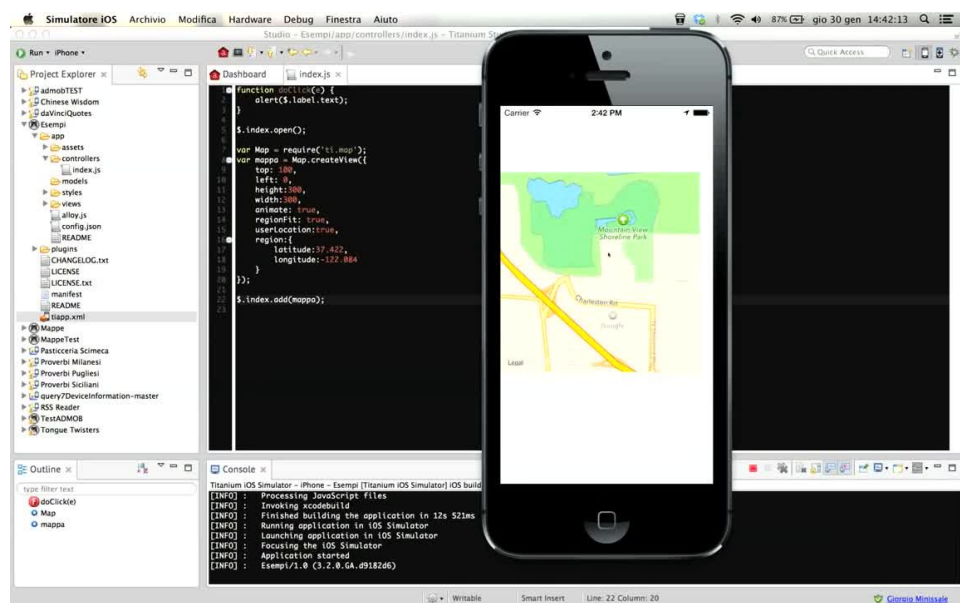
specificando quindi il **TSS** in linea, visto che l'oggetto è comunque una **View**, e quale coordinata visualizzare in apertura, mediante il **gruppo “region”** (ovviamente qui ho messo delle **coordinate di esempio**, scegliendo in particolare quelle del *Googleplex* a Mountain View).

Aggiungiamo ovviamente questo elemento alla **Window**, per la visualizzazione, con ad esempio:

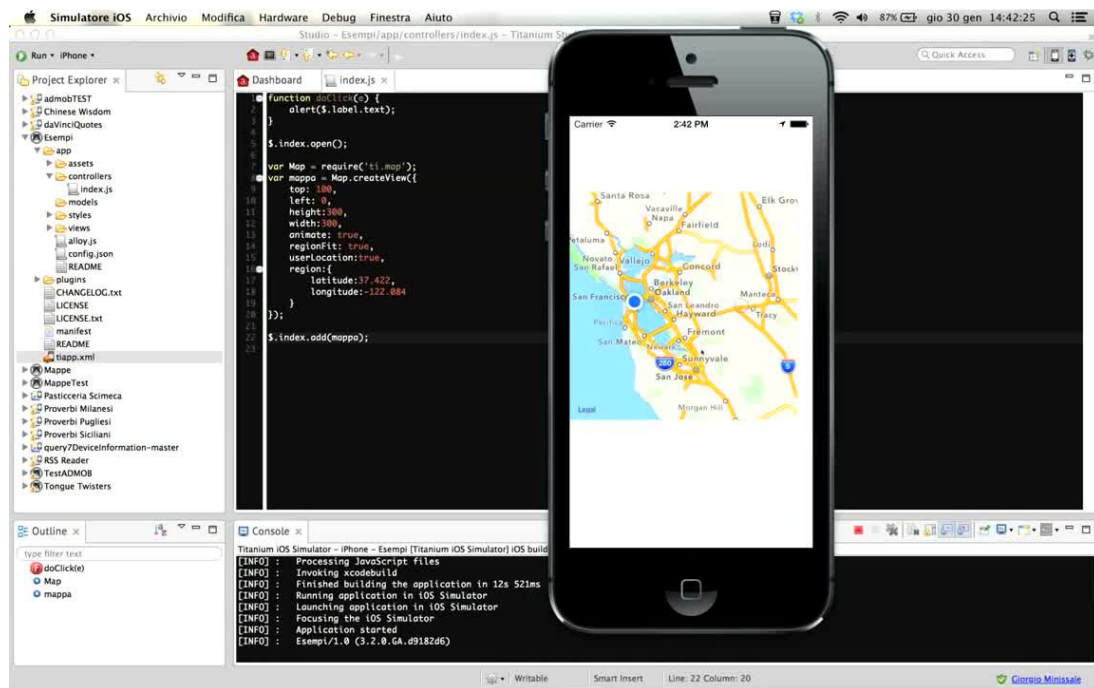
```
$.index.add(mappa);
```



Salvare, compilare ed eseguire.



[NOTA: un altro parametro è MapType, di default Ti.Map.STANDARD_TYPE; “di default” nel senso che, non specificandolo (come nel nostro caso), l'app utilizzerà questo tipo, mostrando la mappa classica (tipo cartina stradale), mentre ad esempio un altro tipo è Ti.Map.SATELLITE_TYPE, per la visualizzazione fotografica dell'area; un altro ancora è TERRAIN_TYPE, per visualizzare alcune strade con i relativi nomi e, infine, il tipo HYBRID_TYPE, per la visualizzazione mista.]



Per questo tutorial – e per questo **ebook** – è tutto!

* * *

* * *