

Integrazione di componenti esterni

Indice generale

Introduzione	2
Integrazione di componenti JavaScript	2
Architettura degli elementi del DOM virtuale	2
Esempio: il componente tagcloud	3
Definizione dell'interfaccia del componente	4
Definizione delle risorse necessarie al componente	5
La classe di interfaccia	6
Inizializzazione del componente	7
Aggiornamento delle proprietà	9
Inizializzazione degli eventi	10
Implementazione dei metodi di interfaccia	11
Gestire parametri e restituire risultati	12
Altri metodi sovrascrivibili nella classe di interfaccia	13
Client.<elemento>.prototype.appendChildObject = function (child, domObj)	13
Client.<elemento>.prototype.close = function (firstLevel, triggerAnimation)	13
Client.<elemento>.prototype.onResize = function (ev)	14
Client.<elemento>.prototype.getRootObject = function ()	14
Client.<elemento>.prototype.onRemoveChildObject = function (child)	14
Client.<elemento>.prototype.onPositionChildObject = function (position)	14
Estensioni di elementi	14
Pubblicazione di elementi tramite package	15
Integrazione di librerie di back-end	16
Integrazione di una libreria JavaScript	17
Integrazione di un pacchetto Node.js	18
Integrazione di un plugin Cordova	19
Definizione della classe di interfaccia	20
Inizializzazione del plugin	21
Implementazione dei metodi	21
Deallocazione delle risorse	22
Test del plugin	23
Pubblicazione del plugin	23

Integrazione di componenti esterni

Scopri come utilizzare componenti JavaScript, pacchetti Node e plugin Cordova nelle tue applicazioni.

Introduzione

Il framework di Instant Developer Cloud contiene un set completo di funzionalità per costruire sistemi omnichannel nel cloud senza normalmente richiedere integrazioni o componenti esterni.

Tuttavia oggi sono sempre più i componenti disponibili in modalità open source che permettono di integrare servizi particolari o di ottenere funzionalità di interfaccia utente desiderabili. Sarebbe impossibile poter disporre di qualunque componente in una forma già integrata con Instant Developer Cloud.

Proprio per questa ragione, questo libro descrive le tecniche di integrazione di tre diverse classi di componenti: quelli di interfaccia utente, i componenti Node.js lato server ed infine i plugin Cordova che consentono l'integrazione con la parte nativa dei dispositivi.

Fra gli esempi di Instant Developer Cloud è disponibile un progetto che mostra questi tre tipi di integrazioni: [Extensibility Design Patterns](#).

Nota: per integrare con successo componenti esterni è necessaria la conoscenza delle API che essi espongono e dell'ambiente operativo (JavaScript, DOM, Node o Cordova) necessario per controllarle.

Integrazione di componenti JavaScript

Architettura degli elementi del DOM virtuale

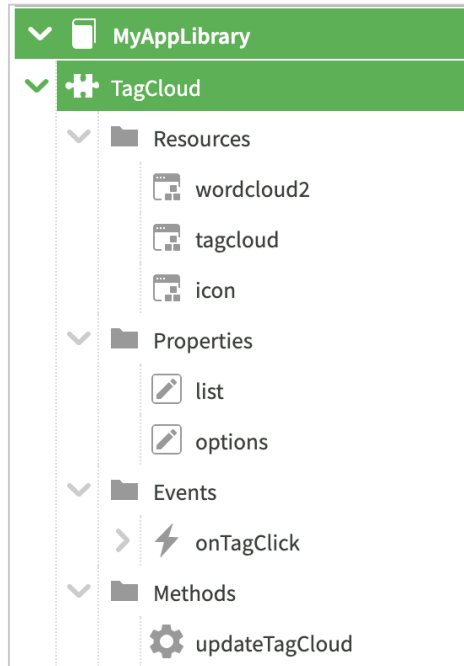
Per poter integrare un nuovo componente JavaScript, è necessario conoscere il funzionamento del DOM virtuale di Instant Developer Cloud.

Come indicato nel paragrafo [Gli elementi visuali](#) del primo manuale della documentazione, il DOM virtuale di Instant Developer Cloud è basato su una serie di elementi visuali che vengono composti in una struttura ad albero che rappresenta l'interfaccia utente dell'applicazione.

Ogni elemento visuale è composto da due classi di codice. Una chiamata *RemElem*, che viene istanziata nel codice di front-end dell'applicazione e un'altra, che è specifica dell'elemento in questione, che viene istanziata nella webview che mostra l'interfaccia utente. Le due istanze comunicano fra di loro, scambiandosi cambiamenti di valore delle proprietà, chiamate a metodi e notifiche di eventi. Questa comunicazione può avvenire

- 3) Definire proprietà, metodi ed eventi dell'elemento.
- 4) Specificare le risorse necessarie al funzionamento del componente.

Vediamo nel dettaglio ognuno di questi passaggi. Nell'immagine seguente è mostrata l'intera definizione dell'elemento visuale *TagCloud*, che, come abbiamo detto, è contenuto nella libreria *MyAppLibrary* di tipo *Applicazione*.



Per aiutare lo sviluppo dell'applicazione è possibile inserire una descrizione della classe e dei vari metodi che la compongono. In questo modo viene creata la documentazione in linea del componente, che entra a far parte dell'insieme della documentazione del sistema. Per maggiori informazioni sulla definizione della documentazione, si veda l'esempio *TagCloud*.

Definizione dell'interfaccia del componente

Le proprietà del componente vengono aggiunte tramite il menu *+Proprietà* e, oltre a specificare i dati relativi al tipo, è possibile attivare i seguenti flag:

- 1) *Design Time*: la proprietà potrà essere impostata anche a design time, tramite la barra delle proprietà dell'IDE.
- 2) *Traducibile*: la proprietà contiene informazioni che dipendono dalla lingua, quindi deve essere gestita dal sistema di traduzioni.
- 3) *Binding predefinito*: se attivato, specifica quale proprietà impostare per default quando si aggiunge una *data source* all'elemento.

Si ricorda che, per definire il valore di design time delle proprietà di tipo oggetto, è necessario specificare il codice JSON utilizzando le doppie virgolette per i nomi di proprietà oltre che per i valori stringa, come nell'esempio seguente: `{ "width": "300px", "height": "150px" }`.

I metodi vengono aggiunti tramite il menu **+Metodo**. Se il metodo deve restituire un valore, cioè se è di tipo *Funzione*, deve essere attivato il flag *Asincrono* e l'ultimo parametro deve essere sempre la callback, cioè un parametro di nome *cb*, avente come tipo dati *Function*. Per aggiungere un evento si deve usare il menu **+Metodo**, cambiando poi il tipo in *Evento*. I metodi degli elementi non sono mai asincroni. Per gli eventi è possibile attivare il flag *Design Time* che lo aggiunge alla barra degli eventi dell'elemento; si consiglia di attivare sempre questo flag.

L'elemento eredita i metodi della classe base, in questo caso *ApplicationLibrary.Element*. In funzione del tipo di elemento, si potranno selezionare altre classi base, come ad esempio:

- *ApplicationLibrary.Container*: per elementi destinati a contenere altri elementi.
- *ApplicationLibrary.Input*: per elementi che permettono di inserire un dato tramite la tastiera.

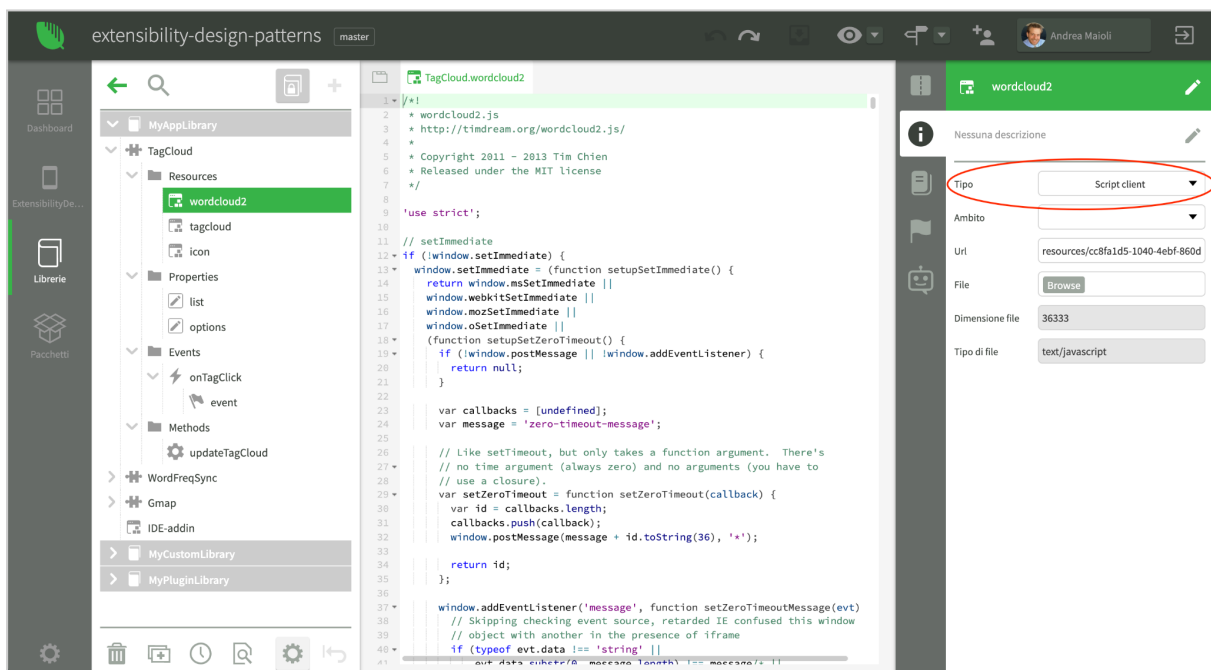
Si consiglia di utilizzare come classi base solo quelle contenute nella cartella *HTML Tags* della libreria *ApplicationLibrary*, a meno che non si stia definendo un'estensione di un altro elemento di cui vedremo un esempio nei paragrafi successivi.

Nel caso in cui l'elemento possa contenere altri elementi, a livello della classe deve essere attivato il flag *Contenitore*. In questo modo l'editor di videate consentirà di aggiungere altri elementi al suo interno.

Definizione delle risorse necessarie al componente

Solitamente gli elementi visuali vengono integrati a partire da un repository GitHub che contiene il codice sorgente: in questo caso si tratta di [wordcloud2](#). Per integrare uno di questi componenti occorre identificare e scaricare tutti i file JavaScript necessari al loro funzionamento e poi caricarli all'interno della classe come risorsa di tipo *Script client*.

Nell'immagine seguente viene mostrato l'unico file JavaScript necessario per l'elemento *TagCloud*.



Le classi JavaScript del componente possono essere anche riferite direttamente da una URL esterna. In questo caso è sufficiente aggiungere la risorsa alla classe tramite il menu *+Risorsa* e poi specificare nome, *Script client* come tipo e l'URL della risorsa. In questo caso, però, se l'applicazione dovesse funzionare offline, il componente potrebbe non essere disponibile.

Oltre alle risorse del componente, è necessario inserire una classe JavaScript che funge da interfaccia fra il framework di Instant Developer Cloud e il componente vero e proprio. Infine è possibile inserire una risorsa di nome *icon*, tipo *File* e specificare il nome di una icona nella *Url*, ad esempio *icon-font*. Questa icona sarà utilizzata per mostrare l'elemento *TagCloud* nella barra degli elementi dell'IDE. I nomi delle icone utilizzabili sono specificati nel file: <https://ide2-developer.instantdevelopercloud.com/svgdefz.svg>.

Si ricorda che è possibile caricare ogni tipo di risorse, non solo quelle JavaScript, come ad esempio i file CSS necessari al componente. La posizione delle varie risorse nell'elenco non è indifferente: le risorse vengono caricate ed eseguite in ordine, attendendo quelle precedenti, ed in questo processo entrano in gioco anche quelle delle classi base, che vengono caricate per prime. Si consiglia quindi di specificare il numero minimo possibile di risorse per rendere più veloce il caricamento a runtime del componente.

Notiamo infine che, sebbene sia possibile aggiungere qualunque tipo di componenti, quelli che si basano su particolari framework come React o Angular sono più complessi da integrare. Se possibile si consiglia di utilizzare componenti che non dipendono da altre librerie che possono interagire negativamente con il framework client di Instant Developer Cloud.

La classe di interfaccia

In questo paragrafo vediamo come comporre la classe JavaScript che funge da interfaccia verso il componente integrato. Lo scopo di questa classe è quello di istanziare il componente perché appaia nella videata al punto giusto, impostare le proprietà iniziali, gestirne i cambiamenti di valore, eseguire sul componente i metodi chiamati dal codice del front-end ed eventualmente restituire dei risultati, infine inoltrare al front-end gli eventi notificati dal componente.

Per creare questa classe, il metodo più semplice è quello di creare un nuovo file JavaScript nella propria workstation con le parti essenziali dell'interfaccia, poi caricarlo nell'IDE come risorsa di tipo *Script client*. A questo punto sarà possibile effettuare modifiche anche direttamente dall'IDE.

A questo proposito si segnala che il codice della classe di interfaccia viene modificato come semplice file di testo, quindi se si sta lavorando con Teamworks, al momento dell'unione delle modifiche nella copia master, il file viene portato così com'è, senza gestire eventuali conflitti. Si consiglia quindi di effettuare modifiche alle risorse di tipo testuale come gli *Script client* solo da un singolo fork del progetto.

Vediamo ora come sono stati implementati questi compiti nell'esempio dell'elemento *TagCloud*.

Inizializzazione del componente

Il codice di inizializzazione della classe di interfaccia dell'elemento *TagCloud* è il seguente:

```
Client.TagCloud = function (element, parent, view)
{
  // Call base class
  Client.Element.call(this, element, parent, view);
  //
  // Every element must have a dom element stored in this property
  this.domObj = document.createElement("div");
  //
  // Default options
  this.options = {
    gridSize: 18,
    weightFactor: 3,
    fontFamily: 'Finger Paint, cursive, sans-serif',
    color: '#f0f0c0',
    backgroundColor: '#001f00',
  };
  this.list = [
    ["Web Technologies", 26],
    ["HTML", 20],
    ["<canvas>", 20],
    ["CSS", 15],
    ["JavaScript", 15],
    ...
    ["flexbox", 5],
    ["viewpoint", 5]
  ];
  //
  // Copy property from design time values
  this.updateElement(element);
  //
  // Attach server events
  this.attachEvents(element.events);
  //
  // Put the DOM object in the document
  parent.appendChildObject(this, this.domObj);
  //
  // Show the first cloud
  this.updateTagCloud();
};
```

Vediamo come è composto il costruttore della classe. Innanzitutto possiamo notare che il costruttore viene assegnato ad una proprietà chiamata *Client.TagCloud*, che rappresenta la classe di questi elementi. È necessario che la proprietà di *Client* che contiene il costruttore abbia lo stesso nome della classe dell'elemento definita nella libreria del progetto.

A questo punto occorre chiamare la classe base degli elementi nella webview, e questo viene fatto dalla prima riga di codice del metodo:

```
Client.Element.call(this, element, parent, view);
```

Il compito successivo della parte di inizializzazione è quello di creare un oggetto DOM che conterrà tutti gli altri oggetti del componente. In questo caso viene creato un DIV, che poi viene memorizzato nella proprietà *this.domObj*. Questo passo è essenziale perché questa proprietà viene poi utilizzata da diversi componenti del framework client. Il tipo di oggetto da creare come contenitore per gli oggetti DOM creati dal componente può variare in funzione delle specifiche del componente stesso.

A questo punto è possibile utilizzare il costruttore per inizializzare proprietà specifiche del componente, fino ad arrivare all'ultima parte in cui viene effettuata l'inizializzazione delle proprietà e degli eventi. Nel caso specifico di *TagCloud* vengono richiamati i seguenti metodi:

- 1) `this.updateElement(element);`
- 2) `this.attachEvents(element.events);`
- 3) `parent.appendChildObject(this, this.domObj);`
- 4) `this.updateTagCloud();`

this.updateElement è il metodo che viene chiamato anche dal framework durante i cambiamenti di valore delle proprietà. L'oggetto passato contiene le proprietà ed i relativi valori da impostare. Si noti che in questo caso *updateElement* viene chiamato prima che *this.domObj* sia stato attaccato al DOM della pagina. Occorre tenerne conto perché in alcuni casi è meglio anteporre il punto 3) al punto 1).

this.attachEvents serve per attaccare gli eventi di interesse al componente in modo da poterli notificare al front-end.

parent.appendChildObject è il metodo che attacca il DOM del componente a quello dell'elemento padre. Si noti che non sempre l'elemento padre è già attaccato al DOM della pagina. Anche se normalmente è così, in alcuni casi esso viene attaccato dopo che tutti i figli sono già stati inizializzati; questo può richiedere per alcuni componenti l'utilizzo di una *setTimeout* per ritardare alcune inizializzazioni.

this.updateTagCloud: è un metodo specifico dell'elemento *TagCloud* che mostra i dati attuali a video.

Per completare la fase di inizializzazione dell'elemento occorre estendere la classe base con la seguente riga di codice, esterna al costruttore. In questo caso occorre estendere la classe base impostata nella classe di interfaccia, come ad esempio *Element*, *Container*, *Input* e così via.

```
Client.TagCloud.prototype = new Client.Element();
```


Aggiornamento delle proprietà

Vediamo adesso come viene eseguito l'aggiornamento delle proprietà, sia al momento dell'inizializzazione che al momento della variazione delle stesse. Il metodo che si occupa di questo è *updateElement*, come vediamo nel codice seguente:

```
Client.TagCloud.prototype.updateElement = function (el)
{
  // Remove bounced properties (for telecollaboration)
  this.purgeMyProp(el);
  //
  // update specific properties
  if (el.list !== undefined) {
    if (typeof el.list === "string") {
      try {
        el.list = JSON.parse(el.list);
      }
      catch (ex) {
        el.list = [{"error parsing the list property", 1}];
      }
    }
    this.list = el.list;
    this.updateTagCloud();
    delete el.list;
  }
  //
  if (el.options !== undefined) {
    if (typeof el.options === "string") {
      try {
        el.options = JSON.parse(el.options);
      }
      catch (ex) {
        el.options = {};
      }
    }
    this.options = el.options;
    this.updateTagCloud();
    delete el.options;
  }
  //
  // Call base class
  Client.Element.prototype.updateElement.call(this, el);
};
```

Il parametro passato è un oggetto che contiene le proprietà da impostare sul componente. La classe di interfaccia può intercettare le proprietà che desidera, rimandando le altre al trattamento di default che avviene nella classe base *Element*.

Per prima cosa viene chiamato il metodo `this.purgeMyProp(el)`; che si occupa di far funzionare correttamente il componente durante le sessioni di telecollaborazione.

A questo punto vengono intercettate le proprietà impostate. Si può usare il seguente schema:

```
// Se la proprietà è variata
if (el.<proprietà> !== undefined) {
  // la memorizzo localmente
  this.<proprietà> = el.<proprietà>;
  // aggiorno la parte visuale
  this.updateUI( ... );
  // elimino la proprietà perché già gestita
  delete el.<proprietà>;
}
```

Al termine del metodo viene richiamata la classe base con la riga di codice:

```
Client.Element.prototype.updateElement.call(this, el);
```

Anche in questo caso si consiglia di richiamare la classe base specifica dell'elemento.

Inizializzazione degli eventi

Il metodo che inizializza gli eventi si chiama *attachEvents* e nell'esempio *TagCloud* ha il seguente codice:

```
Client.TagCloud.prototype.attachEvents = function (events)
{
  if (!events)
    return;
  //
  // Attach specific events
  var pos = events.indexOf("onTagClick");
  if (pos >= 0) {
    events.splice(pos, 1);
    this.sendTagClick = true;
  }
  //
  // Call base class
  Client.Element.prototype.attachEvents.call(this, events);
};
```

Il parametro *events* è un array di stringhe che contiene gli eventi a cui il front-end è interessato, cioè per i quali è stato definito uno script che lo gestisce. Per ogni evento disponibile occorre quindi vedere se esso è presente nella lista passata come parametro, e, in caso affermativo, occorre toglierlo dalla lista e attaccare al componente il relativo *event handler*. Nel caso di *TagCloud*, viene impostata a *true* la proprietà *this.sendTagClick* con la quale si potrà decidere se inviare al server i clic che avvengono sul componente.

È poi importante rimuovere dall'array *events* quelli già gestiti, in modo che le classi base non lo facciano di nuovo. Al termine del metodo, infatti occorre richiamare la classe base.

Implementazione dei metodi di interfaccia

Per ogni metodo definito nell'interfaccia dell'elemento occorre definirne uno corrispondente nella classe di interfaccia. Nell'esempio *tagCloud* è presente il metodo *updateTagCloud*, il cui codice è il seguente:

```
Client.TagCloud.prototype.updateTagCloud = function ()
{
  if (!this.updateTimer) {
    this.updateTimer = setTimeout(function () {
      this.updateTimer = undefined;
      //
      // Prepare options
      var options = this.options || {};
      options.list = this.list || [];
      //
      if (this.sendTagClick) {
        options.click = function (item, dimension, event) {
          this.onTagClick(item, dimension, event);
        }.bind(this);
      }
      //
      // Use the wordcloud2 library to generate the tag cloud
      WordCloud(this.domObj, options);
    }.bind(this), 10);
  }
};
```

Questo metodo può essere chiamato dal codice del front-end essendo parte dell'interfaccia dell'elemento, oppure dal codice dell'interfaccia quando cambiano i valori delle proprietà. Lo scopo del metodo è quello di inizializzare la word cloud usando il codice JavaScript del componente vero e proprio, importato nel progetto e scaricato da GitHub. Questo viene fatto, in particolare dalla riga di codice:

```
WordCloud(this.domObj, options);
```

che utilizza l'oggetto globale *WordCloud*, creato dal codice del componente quando viene inizializzato. La modalità di interazione con il componente è specifica e va studiata componente per componente.

Ad esempio, tra le opzioni del componente è possibile passare una funzione che viene richiamata quando viene eseguito un clic sulla word cloud; per ottenere questo evento, viene passata la funzione *onTagClick*, che presenta il seguente codice:

```
Client.TagCloud.prototype.onTagClick = function (item, dimension, event)
{
  // Disable standard click event handling
  Client.mainFrame.preventClick();
  //
  // Prepare event data
  var cnt = this.saveEvent(event);
```

```

    cnt.item = item;
    cnt.dimension = dimension;
    //
    // Prepare an event list
    var ee = [];
    ee.push({obj: this.id, id: "onTagClick", content: cnt});
    //
    // Send an event list to the server
    Client.mainFrame.sendEvents(ee);
};

```

Questa funzione è interessante perché permette di vedere come notificare al codice di front-end gli eventi che avvengono nel componente. Vediamo i vari passaggi. Innanzitutto, essendo un evento di clic, occorre comunicare al framework che è stato gestito. I clic, infatti, hanno una loro specifica gestione per uniformare il funzionamento mouse e touch. Questo può essere fatto chiamando il metodo *Client.mainFrame.preventClick()*.

A questo punto occorre preparare i parametri da passare all'evento di front-end. Il metodo *this.saveEvent(event)* restituisce un oggetto a cui è possibile aggiungere le proprietà specifiche dell'evento da notificare.

Poi occorre preparare un array di oggetti, ognuno dei quali rappresenta un evento da notificare, con le seguenti proprietà:

- 1) *obj*: è l'id dell'elemento che notifica l'evento. Impostare a: *this.id*.
- 2) *id*: è il nome dell'evento come definito nell'interfaccia dell'elemento. In questo caso "onTagClick".
- 3) *content*: sono i parametri da passare all'evento, in questo caso: *cnt*.

Infine per notificare gli eventi al front-end è possibile passare l'array al metodo: *Client.mainFrame.sendEvents(ee)*.

Gestire parametri e restituire risultati

I parametri dei metodi possono essere definiti nello stesso modo in cui compaiono nell'interfaccia dell'elemento nel progetto. Se il metodo deve restituire un risultato, occorre sempre far corrispondere all'ultimo parametro (*callback*) dell'interfaccia, un parametro di nome *cbId*, che poi dovrà essere utilizzato nel punto in cui vengono restituiti i risultati al codice di front-end.

Ad esempio vediamo il metodo *getPlacePrediction* dell'elemento *Gmap*, che utilizza il seguente codice:

```

Client.Gmap.prototype.getPlacePredictions = function (request, cbId)
{
    try {
        var pthis = this;
        //
        if (request.location)
            request.location = this.toGmapPosition(request.location);
        //

```

```

if (!this.AutoCompleteService)
    this.AutoCompleteService = new
        google.maps.places.AutoCompleteService ();
//
this.AutoCompleteService.getPlacePredictions(request,
    function (results, status) {
        var e = [{obj: pthis.id, id: "cb", content:
            {res: results, cbId: cbId}}];
        Client.mainFrame.sendEvents(e);
    });
}
catch (ex) {
    pthis.notifyError("getPlacePredictions", ex);
}
};

```

Come si può vedere, dopo aver chiamato *getPlacePrediction* delle API client di Google Maps, la callback che intercetta il risultato compone un messaggio e poi utilizza ancora una volta il metodo *sendEvents* per comunicarlo al front-end.

In questo caso le proprietà del messaggio da inviare sono le seguenti:

- *obj*: l'id dell'elemento, in questo caso *pthis.id*.
- *id*: il metodo da chiamare nell'istanza di *remElem* che rappresenta questo elemento nel codice di front-end, in questo caso "cb".
- *content*: un oggetto che contiene i risultati veri e propri nella proprietà *res* e l'id della callback nella proprietà *cbId*. In questo modo il framework può sapere per quale chiamata si stanno restituendo i risultati.

Altri metodi sovrascrivibili nella classe di interfaccia

Oltre ai metodi già descritti, le classi di interfaccia possono sovrascrivere diversi altri metodi della classe base di gestione degli elementi visuali nella webview. Vediamo adesso un elenco di questi metodi, anche se non sono stati usati per il componente *TagCloud*.

```
Client.<elemento>.prototype.appendChildObject = function (child, domObj)
```

Questo metodo dovrebbe essere sovrascritto solo se un elemento contenitore gestisce l'inserimento a video di un elemento figlio in un modo specifico. Il metodo standard consiste nell'aggiungere l'oggetto radice dell'elemento figlio al *domObj* di questo elemento.

Il parametro *child* è l'elemento che deve essere inserito. Il parametro *domObj* è il nodo HTML dell'elemento *child* che deve essere aggiunto a quello attuale.

```
Client.<elemento>.prototype.close = function (firstLevel, triggerAnimation)
```

Questo metodo viene chiamato quando l'elemento viene rimosso dal DOM. Un elemento dovrebbe sovrascrivere questo metodo solo se ha bisogno di aggiornare il DOM in un modo specifico diverso dalla rimozione dell'oggetto radice.

```
Client.<elemento>.prototype.onResize = function (ev)
```

Questo metodo viene chiamato quando il browser viene ridimensionato. Può essere utile per aggiornare la visualizzazione del componente se essa non lo fa automaticamente.

```
Client.<elemento>.prototype.getRootObject = function ()
```

Questo metodo viene chiamato per sapere qual è il nodo HTML radice dell'elemento. Normalmente il metodo restituisce *this.domObj*.

```
Client.<elemento>.prototype.onRemoveChildObject = function (child)
```

Questo metodo viene chiamato su un elemento padre quando un elemento figlio viene rimosso. Va sovrascritto solo se si deve aggiornare l'elemento padre dopo la cancellazione del figlio.

```
Client.<elemento>.prototype.onPositionChildObject = function (position)
```

Questo metodo viene chiamato su un elemento padre quando un elemento figlio viene inserito e poi posizionato in mezzo agli altri. Va sovrascritto solo se si deve aggiornare l'elemento padre dopo l'inserimento posizionato del figlio.

Estensioni di elementi

In alcuni casi può essere interessante estendere elementi esistenti o anche semplicemente adattarne il comportamento alle proprie esigenze. Il framework client di Instant Developer Cloud ha una soluzione molto semplice a queste necessità.

Come esempio immaginiamo di voler estendere l'elemento *Gmap* già presente nel framework base, aggiungendo i metodi *addCircle*, *removeCircle* e *deleteCircles*. Anche questo esempio è disponibile nel progetto [Extensibility Design Patterns](#).

Per ottenere questo risultato si deve creare una nuova classe di tipo *Elemento* in una proprietà libreria di tipo *Applicazione*, e questa classe deve avere lo stesso nome dell'elemento da estendere, in questo caso *Gmap*. Come classe base bisogna scegliere proprio l'elemento da estendere, quindi *ApplicationLibrary.Gmap*.

A questo punto nella classe si devono descrivere i nuovi metodi, con la stessa logica dell'esempio precedente, ed infine si deve aggiungere una nuova risorsa di tipo *Script client* che andrà ad aggiungersi a quella base.

Come nel caso precedente, si crea sul proprio desktop un file JavaScript che poi viene associato alla risorsa. A questo punto sarà possibile modificarla anche direttamente dall'IDE. Vediamo come vengono aggiunti i tre metodi citati.

```
Client.Gmap.prototype.addCircle = function (id, pos, options)
{
    ... il testo completo è nell'esempio Extensibility Design Pattern
};
```

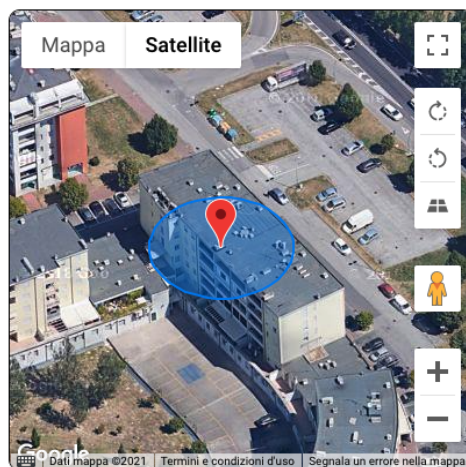
```

Client.Gmap.prototype.deleteCircles = function ()
{
    ... il testo completo è nell'esempio Extensibility Design Pattern
};

Client.Gmap.prototype.removeCircle = function (id)
{
    ... il testo completo è nell'esempio Extensibility Design Pattern
};

```

A questo punto è necessario inserire nelle videate il nuovo componente, oppure sostituire la classe di quelli presenti nelle videate esistenti con quella nuova. Nell'esempio bisognerà sostituire *ApplicationLibrary.Gmap* con *MyAppLib.Gmap*. Questo è il risultato dell'esempio in funzione: ora è possibile disegnare sulla mappa anche dei cerchi.



Publicazione di elementi tramite package

Dopo aver creato i propri elementi visuali, è possibile esportarli come package per riutilizzarli in altri progetti, condividerli all'interno della propria organizzazione oppure con tutti gli utenti di Instant Developer Cloud.

Le operazioni da compiere sono le seguenti:

- 1) Dalla sezione Package del progetto, premere il pulsante + nella lista in alto per aggiungere un nuovo package.
- 2) Premere il pulsante con l'icona a matita per modificare le proprietà del package. In particolare il nome, la descrizione e la versione del componente servono per identificarlo.

La versione IDE contiene la versione minima e massima che possono utilizzarlo, ad esempio 19.0 - 22.5. È possibile indicare una sola versione o lasciarlo vuoto per poterlo usare con qualunque versione dell'IDE.

I prerequisiti identificano eventuali altri package che devono essere presenti nel progetto prima di poter importare questo package, indicati per nome e separati da virgola.

Si noti che il flag *Aggiornamento automatico* attualmente non è utilizzato.

The image shows a web form titled "Modifica pacchetto" (Edit package). It contains the following fields and controls:

- Nome:** Text input containing "MieMappe".
- Descrizione:** Empty text input.
- Versione:** Empty text input.
- Versione IDE:** Empty text input.
- Prerequisiti:** Empty text input.
- Lingua:** Text input containing "en", with download and upload icons to its right.
- Aggiornamento automatico:** An unchecked checkbox.
- Buttons:** A grey "Annulla" (Cancel) button and a green "Salva" (Save) button.

Pagina delle proprietà di un package

- 3) A questo punto occorre selezionare nel progetto gli oggetti che devono far parte del package, ad esempio la libreria che contiene i nuovi elementi visuali. Dopo averla selezionata nell'albero, utilizzare la voce di menu personalizzato *A <nome componente>* per aggiungere la libreria nel componente. È possibile aggiungere anche solo una classe invece di una libreria, se si desidera.
- 4) Infine si può esportare il package ritornando nella sezione corrispondente e premendo il tasto con l'icona a freccia in alto.
- 5) Per decidere se il componente deve essere visibile ad altri utenti della propria organizzazione o dell'intera piattaforma è possibile accedere alla pagina *Componenti* nella console di Instant Developer Cloud e, dopo aver espanso la sezione relativa al proprio componente, modificare i criteri di visibilità.

Per aggiornare un componente è possibile semplicemente ripetere il punto 4) dopo aver modificato il codice dell'elemento nel progetto originario.

Integrazione di librerie di back-end

I componenti di back-end sono la seconda tipologia di integrazione di cui vogliamo illustrare il funzionamento.

Un componente di back-end è costituito da una libreria di codice JavaScript in grado di interagire con l'ambiente operativo di back-end. Questa libreria può essere integrata come elemento a sé stante, sotto forma di un insieme di file JavaScript, ed in questo caso si tratta solitamente di librerie di calcolo, oppure come pacchetto Node.js, codice che può utilizzare l'intero ambiente Node.js in cui sono in funzione le applicazioni.

Vediamo adesso due esempi di integrazione, partendo dalla libreria [word-freq](#), una libreria di codice JavaScript che calcola la frequenza di parole in un testo, che viene integrata come

libreria JavaScript a sé stante. Un esempio funzionante di questa libreria è contenuto nel progetto [Extensibility Design Patterns](#).

Integrazione di una libreria JavaScript

Per integrare una libreria JavaScript, il primo passo è quello di ottenere un file JavaScript che la contiene. Solitamente si ottiene scaricandolo dal repository GitHub della libreria.

A questo punto occorre inserire la definizione degli oggetti della libreria all'interno di una libreria del progetto di tipo Applicazione. Nel caso di esempio useremo ancora *MyAppLibrary*.

Per ogni oggetto che si desidera utilizzare dal codice di back-end di Instant Developer Cloud, è necessario creare una classe in cui vengono definiti i metodi e le eventuali proprietà dell'oggetto. Nel caso di *WordFreqSync*, viene definito il solo metodo *process*, che, a fronte di un testo, restituisce l'array delle parole marcate con la frequenza relativa.

Infine è necessario caricare il codice JavaScript della libreria all'interno della classe come risorsa di tipo *Script server*. Il nome di questa risorsa solitamente deve coincidere con il nome della classe, perché quando l'applicazione è in funzione in un server, il framework esegue la seguente riga di codice per ogni risorsa di tipo *Script server*.

```
App.[nome-risorsa] = require("url script server");
```

In questo modo il codice della risorsa viene caricato nella sessione e viene assegnato ad una classe che può essere istanziata. Nel caso dell'esempio, la classe *WordFreqSync* viene usata come segue:

```
let vv = App.WordFreqSync();  
let wordList = vv.process(txt);
```

Si noti che non esiste un unico metodo per utilizzare gli oggetti derivati dalle librerie JavaScript di back-end, dipende infatti da come la libreria espone i metodi interni all'ambiente esterno. È quindi richiesta la conoscenza dell'ambiente operativo JavaScript e Node.js per avere successo nell'operazione di integrazione.

Quando l'applicazione è in esecuzione locale (modalità "offline"), non è possibile effettuare il *require*, quindi le risorse di tipo *Server script* vengono semplicemente caricate nella webview quando l'applicazione è in fase di inizializzazione. Per ottenere una libreria equivalente può essere necessario adattare l'interfaccia. Questo avviene nell'evento *onStart* tramite il seguente codice:

```
if (app.runsLocally()) {  
  App.WordFreqSync = window.WordFreqSync;  
  delete window.WordFreqSync;  
}
```

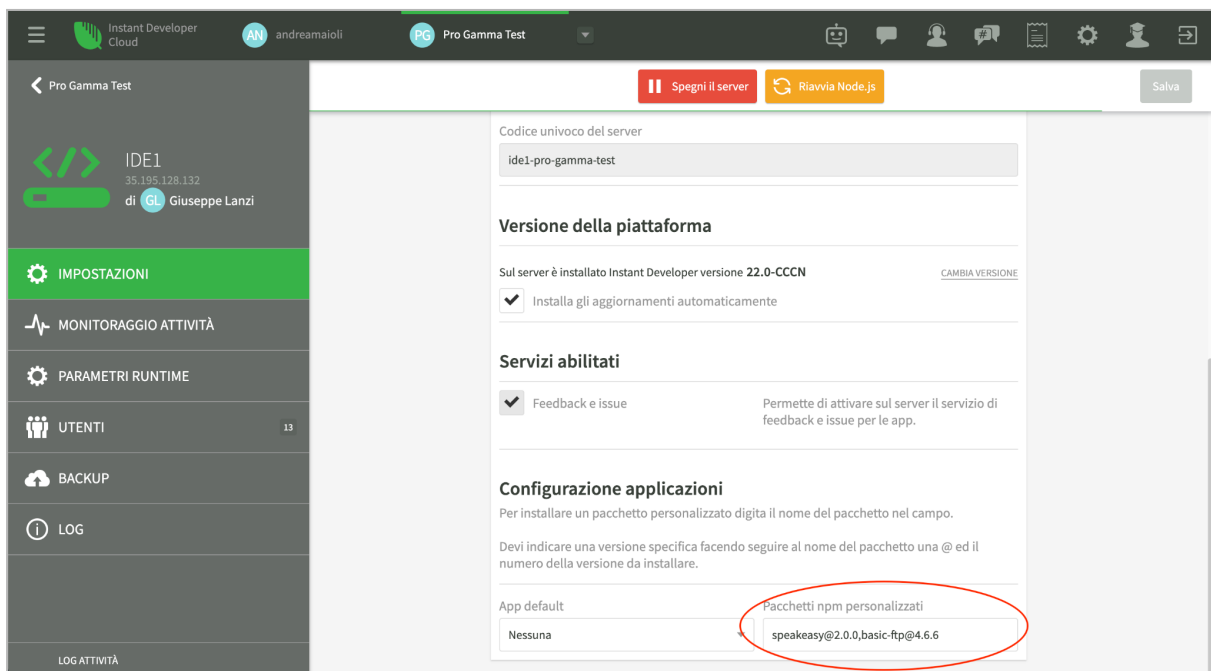
Integrazione di un pacchetto Node.js

L'integrazione di un pacchetto Node.js è la soluzione migliore per aggiungere librerie di back-end che devono essere utilizzate solo nell'ambiente Node.js, cioè quando l'applicazione è in esecuzione in un server.

Il primo passo è identificare il pacchetto che implementa la funzionalità desiderata, solitamente tramite il sito [npm](https://www.npmjs.com/). Nel nostro caso utilizzeremo [basic-ftp](https://www.npmjs.com/package/basic-ftp), un client ftp che può servire alla nostra applicazione per interagire con un server ftp di esempio, posto all'indirizzo: <https://www.wftpserver.com/onlinedemo.htm>.

A questo punto è necessario inserire il pacchetto Node.js nel proprio server, prima di tutto nel server IDE e poi, prima di installare l'applicazione, anche nei server di produzione.

La definizione dei pacchetti da aggiungere ai propri server viene effettuata tramite la console di Instant Developer Cloud, nella pagina *Impostazioni* del proprio server nella console, come si vede nell'immagine seguente:



Nel campo “Pacchetti npm personalizzati” è necessario aggiungere nome e versione dei pacchetti da caricare nel server, separati da virgole e senza spazi. Siccome il valore precedente del campo era “*speakeasy@2.0.0*”, per aggiungere il pacchetto *basic-ftp* è stato modificato in “*speakeasy@2.0.0,basic-ftp@4.6.6*”.

A questo punto è possibile utilizzare le funzionalità aggiuntive, descritte nella documentazione del pacchetto. È opportuno considerare che le funzioni del pacchetto non possono essere sincronizzate con *yield*, quindi bisognerà scrivere il codice tramite *callback* o *promise*. Per questa ragione è necessaria la conoscenza dell'ambiente operativo Node.js e delle relative tecniche di programmazione asincrona.

Nell'esempio seguente effettuiamo la connessione al server, cambiamo la directory di lavoro in download e poi stampiamo a video la lista dei file.

```
App.FirstForm.prototype.onLoad = function (options)
{
  const ftp = require("basic-ftp");
  const client = new ftp.Client();
  client.ftp.verbose = false;
  //
  client.access({
    host : "demo.wftpserver.com",
    user : "demo",
    password : "demo",
    secure : false
  }).catch (e=>console.error(e))
  .then(s=>client.cd("download"))
  .then(s=>client.list())
  .then(function (s) {
    let msg = "";
    for (let i = 0; i < s.length; i++) {
      msg += s[i].name + "\n";
    }
    $helloTxt.innerText = msg;
    //
    client.close();
  });
};
```

Integrazione di un plugin Cordova

In questo paragrafo vediamo come aggiungere nuovi plugin Cordova alla propria applicazione.

Innanzitutto è necessario trovare il plugin Cordova giusto e, anche per questa ricerca, è possibile utilizzare il sito [npm](#). Immaginiamo di voler controllare la “torcia” del dispositivo: ricercando “cordova flashlight” su npm, otteniamo questo risultato: [cordova-plugin-flashlight](#).

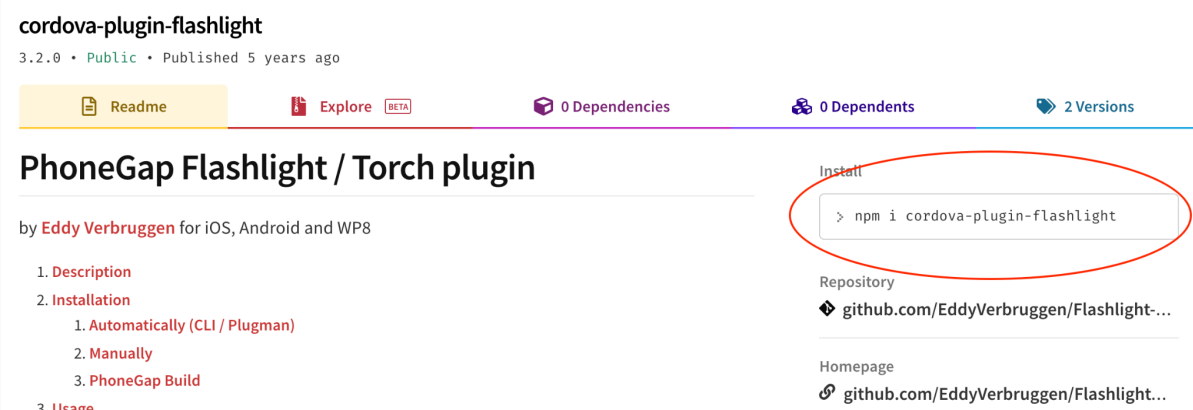
Se non esiste un plugin Cordova che integra le funzioni native richieste, è possibile crearne uno seguendo la [guida allo sviluppo dei plugin \(in lingua inglese\)](#).

Vediamo adesso come integrare il [cordova-plugin-flashlight](#). È possibile vedere il risultato dell'integrazione nel progetto [Extensibility Design Patterns](#).

Dopo aver selezionato il plugin, è necessario definire la sua interfaccia all'interno del progetto Instant Developer Cloud. Questo avviene in modo simile a quanto abbiamo già fatto per gli elementi visuali. I passaggi sono i seguenti:

- 1) Creare una nuova libreria di tipo *Applicazione*, oppure usarne una già esistente. Le librerie di tipo *Applicazione* definiscono le interfacce verso i vari componenti del framework, anche quelli aggiunti, e non contengono codice Instant Developer Cloud.

- 2) All'interno della libreria, creare una nuova classe senza tipo e senza estensione. In questo esempio viene chiamata *Flashlight*.
- 3) Definire proprietà, metodi ed eventi dell'elemento in modo corrispondente alle funzioni messe a disposizione dal plugin. Tutte le funzioni dovranno essere definite come asincrone e quindi avere la callback come ultimo parametro.
- 4) Aggiungere alla classe una risorsa di tipo *Plugin*, in cui verrà scritto il codice JavaScript che implementa la classe di interfaccia del plugin. Il nome della risorsa deve essere uguale a quello della classe. A differenza degli elementi visuali, per i plugin la classe di interfaccia non deve essere caricata da un file esterno, ma scritta direttamente nell'editor di codice della risorsa di tipo Plugin.
- 5) Infine, nella cartella *App Objects* dell'applicazione in cui si vuole utilizzare il plugin, si espande la proprietà *device*, e al suo interno si identifica la proprietà *flashlight* che viene creata dal sistema proprio per accedere al plugin. Nel campo *parametri* della proprietà occorre scrivere *cordova-plugin-flashlight*, cioè il parametro del comando *cordova plugin add* che serve per aggiungere il plugin al progetto Cordova. Il valore corretto è presente nel campo *Install* della pagina del plugin di npm, come mostrato nell'immagine seguente. Se devono essere specificati dei parametri di installazione, possono essere aggiunti dopo il nome del pacchetto, sempre nel campo *parametri*.



cordova-plugin-flashlight
3.2.0 • Public • Published 5 years ago

Readme Explore BETA 0 Dependencies 0 Dependents 2 Versions

PhoneGap Flashlight / Torch plugin

by Eddy Verbruggen for iOS, Android and WP8

- Description
- Installation
 - Automatically (CLI / Plugman)
 - Manually
 - PhoneGap Build
- Usage

Install

```
> npm i cordova-plugin-flashlight
```

Repository
github.com/EddyVerbruggen/Flashlight-...

Homepage
github.com/EddyVerbruggen/Flashlight...

A questo punto è possibile utilizzare i metodi del plugin nella propria applicazione. Tuttavia il plugin funzionerà solo se l'applicazione è in funzione in un launcher personalizzato.

Definizione della classe di interfaccia

La classe di interfaccia è un file JavaScript che mette in comunicazione il plugin Cordova con il framework di Instant Developer Cloud. Vediamo ora le varie parti che la compongono, utilizzando sempre l'esempio della torcia. Per verificare come sono stati integrati i plugin standard, è possibile vedere il [repository pubblico](#) di InstaLauncher.

Si ricorda che la classe di interfaccia deve essere caricata come risorsa di tipo *Plugin* nella classe che definisce il plugin.

Inizializzazione del plugin

L'inizializzazione del plugin avviene tramite le seguenti righe di codice:

```
var Plugin = Plugin || {};  
var PlugMan = PlugMan || {};  
  
Plugin.Flashlight = {};  
  
Plugin.Flashlight.init = function ()  
{  
  // inserire qui il codice di inizializzazione  
};
```

Implementazione dei metodi

Per ogni metodo della libreria, deve essere presente una corrispondente funzione del plugin. Nel caso di esempio, sono stati implementati i seguenti metodi:

```
Plugin.Flashlight.available = function (req)  
{  
  window.plugins.flashlight.available(function(isAvailable) {  
    req.setResult(isAvailable);  
  });  
}  
  
Plugin.Flashlight.switchOn = function (req)  
{  
  var opt = {};  
  if (req.params.intensity)  
    opt.intensity = req.params.intensity;  
  //  
  window.plugins.flashlight.switchOn(  
    function() {}, // optional success callback  
    function() {}, // optional error callback  
    opt // optional as well  
  );  
}  
  
Plugin.Flashlight.toggle = function (req)  
{  
  var opt = {};  
  if (req.params.intensity)  
    opt.intensity = req.params.intensity;  
  //  
  window.plugins.flashlight.toggle(  
    function() {}, // optional success callback  
    function() {}, // optional error callback  
    opt // optional as well  
  );  
}
```

```

Plugin.Flashlight.switchOff = function (req)
{
    window.plugins.flashlight.switchOff();
}

```

È importante notare che ogni metodo riceve un parametro *req*, che rappresenta la chiamata da parte dell'applicazione. I parametri della chiamata sono presenti nella proprietà *req.params*, come si vede nel metodo *switchOn*.

L'implementazione vera e propria dipende dal plugin Cordova utilizzato. Per restituire un risultato all'applicazione è necessario chiamare:

- *req.setResult(result)*: per restituire un valore.
- *req.setError(error)*: per generare un'eccezione.

Per notificare eventi all'applicazione, è necessario chiamare il metodo *PlugMan.sendEvent* nell'*event handler* che la classe di integrazione registra presso il plugin. Vediamo come esempio il metodo *watchAcceleration* del plugin *Accelerometer*. Questo metodo attiva la notifica dell'accelerometro all'applicazione chiamando il relativo metodo del plugin e passando la callback evidenziata in giallo. La riga di codice *PlugMan.sendEvent* genera la notifica dell'evento *onAcceleration* all'applicazione.

```

Plugin.Accelerometer.watchAcceleration = function (req)
{
    // Clean, then set.
    this.clearWatch(req);
    //
    var watchID = navigator.accelerometer.watchAcceleration(
        function (acceleration) {
            req.result = acceleration;
            PlugMan.sendEvent(req, "Acceleration");
        },
        function () {
            req.result = {error: "error"};
            PlugMan.sendEvent(req, "Acceleration");
        },
        req.params);
    //
    // Remember which app requests this watch
    this.watchList.push({id: watchID, app: req.app});
};

```

Deallocazione delle risorse

Al momento della chiusura dell'applicazione viene chiamato il metodo *stopApp* del plugin, che si deve occupare di liberare eventuali risorse acquisite durante il funzionamento.

```

Plugin.Flashlight.stopApp = function (app)
{
    window.plugins.flashlight.switchOff();
};

```

Test del plugin

È possibile lanciare l'applicazione in anteprima in un browser, tuttavia:

- Chiamando un metodo che non restituisce risultati, esso non avrà effetto.
- Chiamando un metodo che restituisce risultati, viene generata l'eccezione: *Plugin not found*.
- Nessun evento verrà notificato.

Per testare il plugin in anteprima, è consigliabile creare e configurare un *Developer Launcher* tramite la console di Instant Developer Cloud, come indicato nel manuale [Launcher: funzioni native e pubblicazione sugli store](#).

Un *Developer Launcher* è un'app che funziona come InstaLauncher, ma contiene anche i plugin nativi definiti nell'applicazione che viene installata nel launcher. A questo punto sarà possibile eseguire i test completi dell'applicazione, sia collegandola all'IDE che usando l'applicazione installata.

Per installare un *Developer Launcher* sui propri dispositivi è possibile operare come segue:

- Per iOS è possibile scaricare il progetto Cordova dalla console di Instant Developer Cloud e poi usare XCode per caricarlo direttamente sul proprio iPhone o iPad. In questo modo si ha la possibilità di effettuare un debug di basso livello. In alternativa è possibile configurare l'intera applicazione nel portale developer di Apple, caricare l'applicazione tramite il build server e poi attivare l'alpha test o il beta test tramite TestFlight.
- Per Android è conveniente scaricare l'applicazione in formato APK sui propri device e poi usare direttamente quella. È possibile anche scaricare il progetto Cordova nel caso in cui sia necessario un debug di basso livello; tuttavia questo richiede la configurazione completa dell'ambiente Cordova per Android.

Nota bene: il Developer Launcher contiene i plugin aggiuntivi solo se essi sono stati direttamente referenziati almeno una volta nel progetto. Nel caso di esempio, il plugin *flashlight* viene aggiunto solo se nel codice dell'applicazione viene fatto riferimento diretto a *app.device.flashlight*. Per verificare se una riga di codice contiene un riferimento diretto, è sufficiente aprire il menù contestuale nell'editor di codice sulla proprietà *flashlight* e verificare che sia presente la voce di menù *Vai a flashlight*.

Pubblicazione del plugin

Come già visto nel paragrafo [Pubblicazione di elementi tramite package](#), anche per i plugin è possibile creare pacchetti Instant Developer che ne consentono il riutilizzo in altri progetti, oppure la condivisione con altri utenti.

La procedura per la creazione del pacchetto è identica a quella per gli elementi. Tuttavia dopo aver importato il pacchetto in un altro progetto, sarà necessario indicare il comando di importazione nel parametro della proprietà relativa al plugin nell'applicazione, come indicato al punto 5 della [lista precedente](#).