Attività Progettuale Sistemi Mobili M – Anno Accademico 2013/2014

APPLICAZIONI IBRIDE E NATIVE : PREGI E DIFETTI DEI DUE APPROCCI

Daniele Campogiani

Indice generale

1 Introduzione

Lo scopo di questa attività progettuale è studiare le possibilità offerte dalla realizzazione di applicazioni mobili chiamate *Ibride*, ossia scritte usando strumenti pensati per il web, ma avendo anche a disposizione API per l'accesso al dispositivo.

L'argomento è stato scelto per capire se attualmente un approccio di questo tipo può essere utilizzato per la realizzazione di una applicazione completa, permettendo allo sviluppatore di scrivere una sola applicazione e poterla poi distribuire su diverse piattaforme, evitando così di dover scriver un'applicazione per ogni specifica piattaforma.

Per poter utilizzare API native utilizzando tecnologie web è stato utilizzato Apache Cordova, e per strutturare l'applicazione è stato utilizzato il framework javascript scritto da Google chiamato AngularJS.

L'applicazione realizzata permette all'utente di scattare delle fotografie, aggiungere una descrizione per ognuna di esse e poi visualizzarle in una timeline oppure in una mappa, posizionando ogni foto nelle coordinate in cui è stata scattata. L'utente può anche decidere di condividere ogni foto utilizzando altre applicazioni installate nel dispositivo.

Al fine di valutare le performances ottenute con una applicazione ibrida, è stata poi realizzata un'applicazione *nativa,* utilizzando Java su piattaforma Android, con le stesse funzionalità.

E' disponibile online il codice sorgente sia per l'applicazione ibrida, al seguente indirizzo:

https://github.com/dcampogiani/My-Photo-Diary

sia per l'applicazione nativa, all'indirizzo:

https://github.com/dcampogiani/My-Native-Photo-Diary

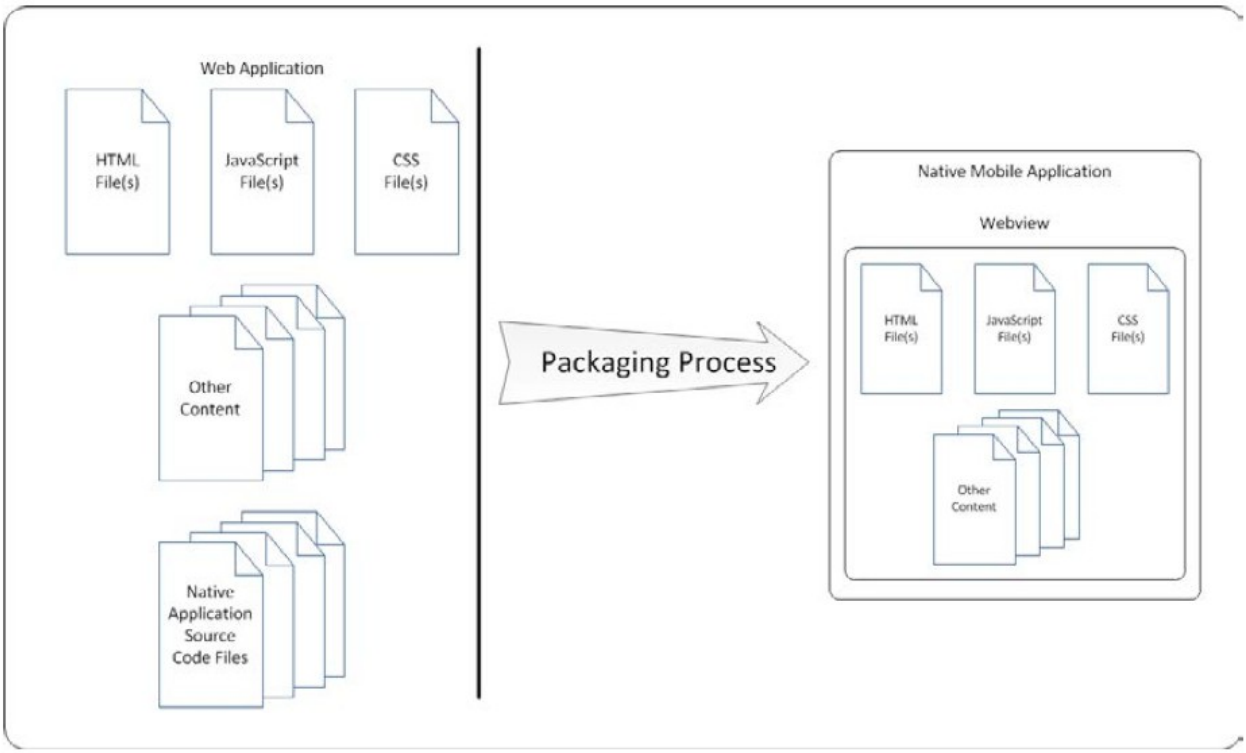
2 Apache Cordova

2.1 Introduzione

Apache Cordova è un framework open source per la realizzazione di applicazioni cross-platform usando HTML5. L'approccio usato da Cordova è però unico nel suo genere, perché fonde le capacità native di un dispositivo con la possibilità di realizzare applicazioni usando gli standard web, dando così vita alle applicazioni chiamate *ibride*.

Il punto di debolezza delle applicazioni web su dispositivi mobili è sempre stato la mancanza della possibilità di usare funzionalità fornite dal produttore per interfacciarsi con i vari componenti hardware, e mentre si spera che in un futuro questa limitatezza possa essere superata attraverso sforzi di standardizzazione, Cordova fornisce una serie di API alle applicazioni che vengono eseguite all'interno del suo container per poter accedere a tali funzionalità, ad esempio sarà possibile utilizzare la fotocamera, l'accelerometro ed il gps.

Cordova si occupa anche di realizzare il pacchetto applicativo (ad esempio file .apk per Android ed .ipa per iOS) per ogni piattaforma supportata a partire dai file della web application.



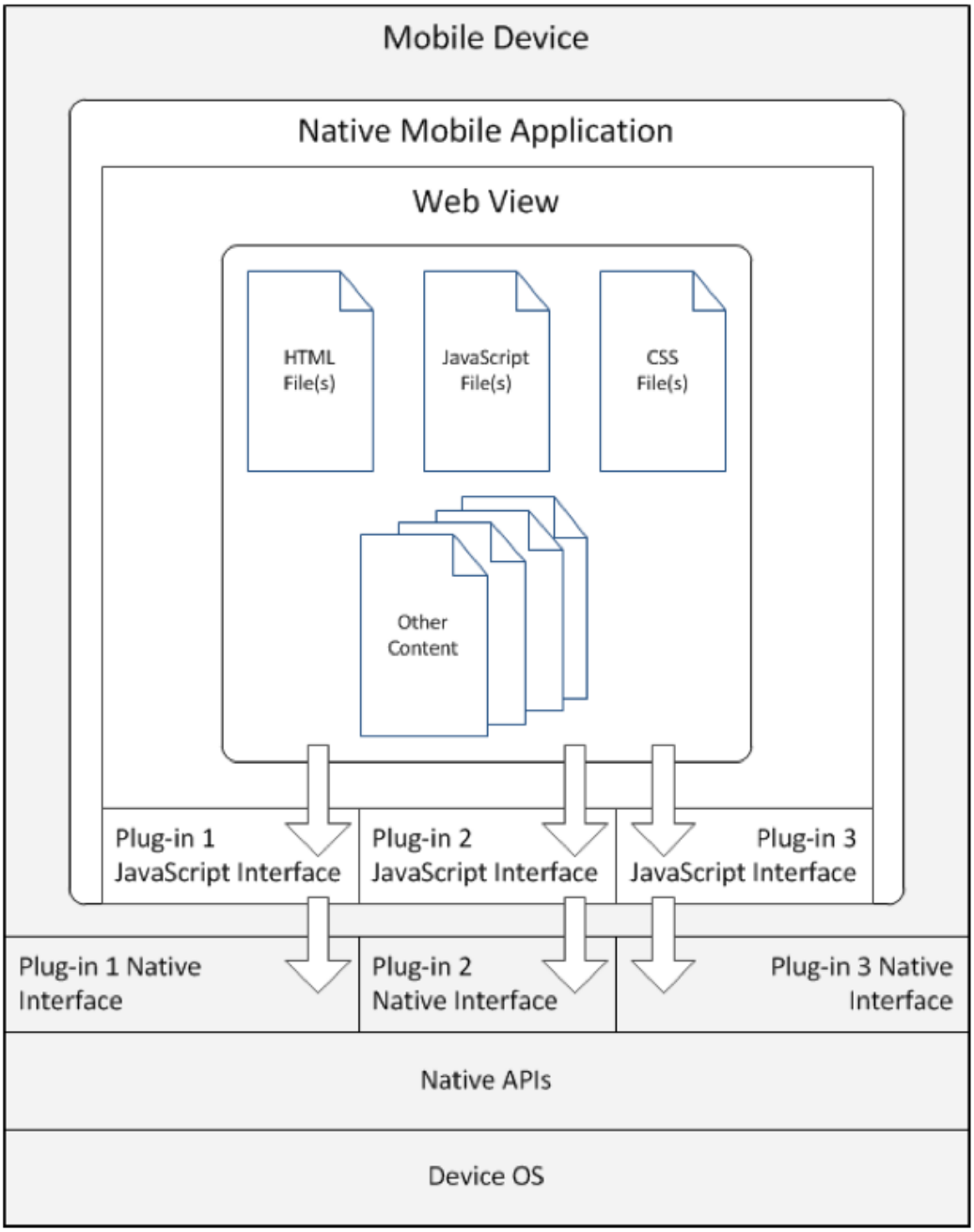
L'applicazione nativa consisterà in una webview (un browser integrato) che mostrerà il contenuto utilizzando tutto lo schermo disponibile. Questo browser embedded potrà utilizzare l'interprete di Javascript fornito dal sistema operativo che ospita l'applicazione ed in più le API esposte da Cordova.

Attualmente Cordova supporta le seguenti piattaforme [1] :

	amazon-fireos	android	blackberry10	Firefox OS	ios	Ubuntu	wp7 (Windows Phone 7)	wp8 (Windows Phone 8)	win8 (Windows 8)	tizen
cordova CLI	✔ Mac, Windows, Linux	✔ Mac, Windows, Linux	✔ Mac, Windows	✔ Mac, Windows, Linux	✔ Mac	✔ Ubuntu	✔ Windows	✔ Windows	✔	✘
Embedded WebView	✔ (see details)	✔ (see details)	✘	✘	✔ (see details)	✔	✘	✘	✘	✘
Plug-in Interface	✔ (see details)	✔ (see details)	✔ (see details)	✘	✔ (see details)	✔	✔ (see details)	✔ (see details)	✔	✘
Platform APIs										
Accelerometer	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
Camera	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
Capture	✔	✔	✔	✘	✔	✔	✔	✔	✘	✘
Compass	✔	✔	✔	✘	✔ (3GS+)	✔	✔	✔	✔	✔
Connection	✔	✔	✔	✘	✔	✔	✔	✔	✔	✔
Contacts	✔	✔	✔	✔	✔	✔	✔	✔	✘	✘
Device	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
Events	✔	✔	✔	✘	✔	✔	✔	✔	✔	✔
File	✔	✔	✔	✘	✔	✔	✔	✔	✔	✘
Geolocation	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
Globalization	✔	✔	✘	✘	✔	✔	✔	✔	✘	✘
InAppBrowser	✔	✔	✔	✘	✔	✔	✔	✔	uses iframe	✘
Media	✔	✔	✔	✘	✔	✔	✔	✔	✔	✔
Notification	✔	✔	✔	✘	✔	✔	✔	✔	✔	✔
Splashscreen	✔	✔	✔	✘	✔	✔	✔	✔	✔	✘
Storage	✔	✔	✔	✘	✔	✔	✔ localStorage & indexedDB	✔ localStorage & indexedDB	✔ localStorage & indexedDB	✔

2.2 Architettura

A partire dalla versione 3.0 (al momento della scrittura siamo alla versione 3.5.0-0.24) le API sono state separate in diversi plugin, per poter includere nel progetto solamente quelli necessari, e per avere un'architettura più facilmente estensibile anche con plugin di terze parti.



Dalla figura [2] si può notare come ogni singolo plugin sia composto da una parte in Javascript e una parte invece scritta nel linguaggio supportato dalla specifica piattaforma.

Attualmente Cordova fornisce questi plugin:

- Battery Status
- Camera
- Console
- Contacts

- Device
- Device Motion
- Device Orientation
- Dialogs
- File
- File Transfer

2.3 Command Line Interface

Cordova fornisce un tool, chiamato *Command Line Interface* (CLI) per aiutare lo sviluppatore nella creazione, gestione e deploy di un progetto. Il CLI utilizza i vari sdk delle singole piattaforme che vanno installati manualmente.

2.3.1 Installazione

Per installare cli di cordova è prima necessario scaricare ed installare *node.js* e poi eseguire:

```
sudo npm install -g cordova
```

npm è il package manager di node, l'opzione -g specifica di installare il pacchetto globalmente e cordova è il nome del pacchetto da installare.

2.3.2 Creazione di una applicazione

Con il comando

```
cordova create hello com.example.hello HelloWorld
```

cordova creerà la cartella *hello* contenente il progetto per l'applicazione con l'identificatore *com.example.hello*. La sottocartella www contiene i file *html/css/javascript* scritti dallo sviluppatore, e il file *config.xml* specifica i metadati necessari per la generazione e la distribuzione dell'applicazione. Il terzo parametro sarà il nome user-friendly dell'applicazione, se omesso viene utilizzato il nome *HelloCordova*.

2.3.3 Gestione delle Piattaforme

Una volta nella cartella dell'applicazione è possibile gestire le piattaforme per le quali si intende fare il deploy attraverso il comando

```
cordova platform
```

seguito da un'opzione (*add/rm*) ed il nome della piattaforma. Per ottenere una lista delle piattaforme correntemente attive per il progetto basterà utilizzare l'opzione *ls* dopo il comando platform.

Per ogni piattaforma attiva verrà utilizzata una cartella nel path *nomeProgetto/platforms/nomePiattaforma*.

2.3.4 Build

Attraverso il comando

```
cordova build (nomePiattaforma)
```

verrà generato il codice specifico per ogni piattaforma attualmente attiva. Si può limitare il build ad una singola piattaforma, specificandone il nome dopo il comando build.

2.3.5 Test e Run

Laddove il produttore della piattaforma fornisca anche un emulatore/simulatore per desktop quest'ultimo può essere lanciato per testare l'applicazione attraverso il comando

```
cordova emulate nomePiattaforma
```

mentre per testare l'applicazione direttamente sul dispositivo fisico si utilizza

```
cordova run nomePiattaforma
```

2.3.6 Gestione Plugin

Come precedentemente spiegato, Cordova utilizza una architettura fortemente basata su plugin. Fino ad ora non è stato aggiunto nessuno, e quello che otteniamo è la possibilità di avere una normale applicazione web con il vantaggio di avere tutti i file in locale (potendo quindi funzionare anche in assenza di connettività) e resa disponibile come pacchetto per la piattaforma specifica.

Ma per poter sfruttare a pieno le potenzialità di Cordova e quindi poter accedere ad *API native* vediamo come aggiungere un plugin.

Apache ha creato un repository web nel quale cercare i plugin, per poter leggere una breve descrizione e la documentazione. Questo repository è disponibile al seguente indirizzo:

http://plugins.cordova.io/

Oltre ai plugin resi disponibili direttamente da Apache, è possibile trovare anche plugin di terze parti resi disponibili da sviluppatori. Il repository è nuovo, quindi non sono ancora presenti tutti i plugin sviluppati, per cui consiglio di fare una ricerca anche su github.com.

Una volta ottenuto l'identificativo di un plugin (ad esempio org.apache.cordova.camera) l'installazione è semplice, basterà utilizzare il comando

```
cordova plugin add “identificatore plugin”
```

Per rimuovere un plugin basta sostituire *add* con *rm* al comando qui sopra, è possibile ottenere una lista dei plugin correntemente attivi attraverso:

```
cordova plugin ls
```

2.4 Realizzazione di un plugin

Un plugin è composto da un opportuno file contenente metadati chiamato *plugin.xml,* del codice Javascript, che esporrà le API utilizzabili dall'applicazione e opzionalmente del codice nativo. Il codice nativo è opzionale e nel caso sia assente il plugin è completamente realizzato in Javascript, e non potrà quindi utilizzare le API native fornite dal produttore della piattaforma.

Per comprendere meglio la anatomia di un plugin cordova riporterò come esempio [3] un plugin che permette di ottenere il nome dell'operatore e il codice del paese in cui è il dispositivo.

2.4.1 Javascript

Attraverso la chiamata al metodo *cordova.exec* avviene il passaggio di controllo al codice nativo di ogni piattaforma, per cui il codice Javascript del plugin andrà ad invocare questo metodo.

Una volta che il codice nativo ha terminalo l'esecuzione invocherà una funzione di callback e restituisce il valore di ritorno all'oggetto Javascript. La signature completa del metodo *exec* è la seguente:

```
cordova.exec(successCallback, errorCallback, 'PluginObject', 'pluginMethod',
[arguments])
```

I primi due parametri specificano che funzioni eseguire in caso di successo o errore lato codice Javascript, il parametro *PluginObject* è una stringa che identifica l'oggetto nativo che contiene il metodo richiesto specificato dal parametro *pluginMethod*. L'ultimo argomento opzionale è un array che contiene gli argomenti da passare al metodo nativo.

Il codice Javascript del plugin sarà:

```
var cordova = require('cordova');

var carrier = {

        getCarrierName : function(successCallback, errorCallback) {

                cordova.exec(successCallback, errorCallback, 'CarrierPlugin',
                'getCarrierName',       []);

        },

        getCountryCode : function(successCallback, errorCallback{

                cordova.exec(successCallback, errorCallback, 'CarrierPlugin',
                'getCountryCode', []);

        }

};

module.exports = carrier;
```

L'ultima riga serve a rendere visibile l'oggetto carrier all'intero delle applicazioni che sono in esecuzione all'interno del container Cordova.

2.4.2 Java

Per implementare il plugin lato codice nativo su piattaforma Android occorre estendere la classe *CordovaPlugin* fornita da Cordova, e realizzare i metodi *initialize()* e *execute()*.

Il primo serve a rendere il codice Java *"aware"* del container Cordova, mentre il secondo realizza il dispatch dei metodi e fornisce l'implementazione dei metodi richiedibili da

Javascript.

Utilizzando quindi le API native fornite da Android, il plugin sarà realizzato come segue:

```
public class CarrierPlugin extends CordovaPlugin {

        public static final String ACTION_GET_CARRIER_NAME =
"getCarrierName";

        public static final String ACTION_GET_COUNTRY_CODE =
"getCountryCode";

        public TelephonyManager tm;



        public void initialize(CordovaInterface cordova, CordovaWebView webView)
{

                super.initialize(cordova, webView);

                Context context = this.cordova.getActivity().getApplicationContext();

                tm =
(TelephonyManager)context.getSystemService(Context.TELEPHONY_SERVICE);

        }


        public boolean execute(String action, JSONArray args, CallbackContext
callbackContext) throws JSONException {

                try {

                        if (ACTION_GET_CARRIER_NAME.equals(action)) {

                                callbackContext.success(tm.getSimOperatorName());

                                return true;

                        }

                        else {

                                if (ACTION_GET_COUNTRY_CODE.equals(action)) {
```

```
                                callbackContext.success(tm.getSimCountryIso());
                                return true;
                            }
                        }
                    callbackContext.error("Invalid Action");
                    return false;
                    } catch (Exception e) {
                        System.err.println("Exception: " + e.getMessage());
                        callbackContext.error(e.getMessage());
                        return false;
                    }
            }
}
```

3 AngularJS

3.1 Le basi di AngularJS

> *I hereby declare AngularJS to be MVW framework – Model-View-Whatever. Where Whatever stands for "whatever works for you".*
>
> *Igor Minar [4]*

AngularJS è un framework per applicazioni web *“single page”* sviluppato da Google.

Si differenzia dai framework precedenti come *jQuery* nei quali lo sviluppatore doveva avere una conoscenza completa del DOM per poi manipolarlo scrivendo la logica in javascript, ad esempio:

```
var btn = $("<button>Hi</button>");
btn.on('click', function(evt) { console.log("Clicked button") });
$("#checkoutHolder").append(btn);
```

AngularJS invece aumenta il potere espressivo dell'HTML fornendo agli sviluppatori nuovi tag built-in e la possibilità di crearne degli altri.

Inoltre la libreria è di dimensioni veramente ridotte, appena 9KB (la versione minimizzata) ed è disponibile open source.

3.2 Data Binding

I framework web classici come Rails a partire da dati del modello producono una view per l'utente, generando una view *“single-way”* che riflette i dati del modello solo al momento al momento dell'istanziazione.

AngularJS invece adotta un approccio unico, creando dei *live templates* come view che sono sempre aggiornati rispetto ai dati del modello. Questo ci permette di scrivere nell'HTML

```
<input ng-model="name" type="text" placeholder="Your name">
```

```
<h1>Hello {{ name }}</h1>
```

Le doppie parentesi graffe indicano un'espressione AngularJS, il framework valuterà l'espressione per effettuare il rendering del contenuto.

Senza scrivere nessuna riga di codice in Javascript abbiamo ottenuto un *two-way data binding* direttamente nella view.

In questo esempio, ogni volta che cambierà il valore del tag <input> verrà aggiornato anche il contenuto del tag <h1> sottostante.

3.3 Scopes

Gli *scopes* sono componenti core di ogni applicazione AngularJS, vengono utilizzati come *glue* tra i controller e le view. Grazie al *live binding* di AngularJS uno *scope* è immediatamente aggiornato se l'utente interagisce con la view, e viceversa la view è immediatamente aggiornata se modifichiamo lo *scope* eseguendo qualche operazione nel codice Javascript.

AngularJS gestisce i diversi *scope* di una applicazione attraverso una relazione gerarchica che rispetta la composizione del DOM, all'avvio dell'applicazione viene infatti creato un *rootScope* e tutti gli altri *scope* discendono da questo.

L'oggetto *scope* è un *plain old Javascript object*, possiamo quindi aggiungere tutte le proprietà di cui necessitiamo. Esso funge da *data model* e tutte le sue proprietà sono automaticamente accessibili dalla view:

```
<div ng-app="myApp"> <h1>Hello {{ name }}</h1>
</div>

angular.module('myApp', []) .run(function($rootScope) {
    $rootScope.name = "World";
});
```

3.4 Controllers

I *controllers* in AngularJS sono funzioni definite dallo sviluppatore per aggiungere funzionalità allo scope della view. Attraverso la Dependency Injection AngularJS passa automaticamente lo scope associato ad ogni *controller* al relativo costruttore. Nel costruttore può essere configurato lo stato iniziale dello scope, lo sviluppatore si deve preoccupare solamente di scrivere la funzione costruttore, poiché l'instanziazione dei controller è a carico del framework.

Ad esempio:

```
function FirstController($scope) {
        $scope.message = "hello";
}
```

Per creare delle azioni invocabili dalle view basterà creare delle funzioni sullo scope associato e poi effettuare il binding, ad esempio sfruttano la built-in directive ng-click che gestisce l'evento click del browser. Quindi volendo fare un esempio leggermente più complesso avremo nella view:

```
<div ng-controller="FirstController">
<h4>Il mio contatore</h4>
<button ng-click="add(1)">Add</button>
<a ng-click="subtract(1)">Subtract</a>
<h4>Valore : {{ counter }}</h4>
</div>
```

E nel relativo controller:

```
app.controller('FirstController', function($scope) {
```

```
$scope.counter = 0;

$scope.add = function(amount) { $scope.counter += amount; };

$scope.subtract = function(amount) { $scope.counter -= amount; };

});
```

Sia il bottone che il link sono collegati ad una azione definita nello scope, per cui quando verranno premuti/selezionati AngularJS chiamerà i rispettivi metodi.

Come si può notare nei *controllers* scritti dal programmatore non c'è una modifica *manuale* del DOM, il tutto viene gestito automaticamente da AngularJS.

Tutti gli scope che sono creati con ereditarietà prototipale, hanno cioè accesso agli scope *"avi"*. Quindi ogni qual volta AngularJS non riesce a trovare una proprietà richiesta nello scope corrente risalirà questa catena di ereditarietà fino al *rootScope* associato all'applicazione.

È importante sapere che per motivi di gestione ottimale della memoria e di performances i controllers sono instanziati solo quando sono necessari, e deallocati quando non lo sono più. Per cui ogni volta che cambia lo stato di una applicazione o ricarichiamo una view, i controller correnti vengono deallocati e instanziati i nuovi controller.

3.5 Directives

AngularJS fornisce molte built-in directives che servono ad aumentare l'espressività dell'HTML.Le *directives built-in* sono tag che iniziano con il prefisso ng, per un elenco dettagliato si rimanda alla documentazione di AngularJS [5].

Inoltre AngularJS permette di definire nuove directive agli sviluppatori con le funzionalità desiderate e di unire diverse directive attraverso un processo chiamato composizione.

Per creare una nuova directive bisogna definirla come segue:

```
angular.module('myApp', []) .directive('myDirective', function() {

return {

restrict: 'E',
```

```
template: '<a href="http://google.com"> Click me to go to Google</a>'
} });
```

Nell'esempio sono state utilizzate solo la proprietà *restrict* e *template* per semplicità, per l'elenco esaustivo si rimanda alla documentazione di AngularJS su come creare directives custom [6].

Avendo definito questa directive, ogni qual volta nell'HTML verrà incontrata la directive AngularJS invocherà la funzione associata. Possiamo quindi scrivere nella nostra view

```
<my-directive></my-directive>
```

e AngularJS provvederà a inserire nel DOM un tag <a> con link a google.

Per poter passare dello stato ad una custom directive attraverso gli attributi:

```
<my-directive my-url="http://google.com" my-link-text="Click me to go to Google">
</my-directive>
```

definendo opportunamente la directive:

```
angular.module('myApp', []) .directive('myDirective', function() {
return {
restrict: 'E',
replace: true,
scope: {
        myUrl: '@', // binding strategy
        myLinkText: '@' // binding strategy
},
```

```
template: '<a href="{{myUrl}}">' + '{{myLinkText}}</a>'
} })
```

Così facendo la directive avrà uno scope interno contenente l'url e il testo del link. Ci sono diverse strategie di binding tra attributi passati nella view e lo scope interno, nell'esempio è stata utilizzata la strategia *@* che copia il valore dell'attributo nella corrispondente proprietà dello scope.

3.6 View

In applicazioni *"single page"* la navigazione da una view ad un altra è un aspetto critico, vogliamo mostrare del nuovo contenuto all'utente senza forzarlo ad aspettare l'apertura di un altra pagina HTML. AngularJS per ottenere questo risultato permette di scrivere diversi templates, per poi mostrare il contenuto di interesse in base allo *"stato"* dell'applicazione.

AngularJS fornisce il modulo *ngRoute* (da scaricare ed includere manualmente) che realizza la direttiva *ng-view* utilizzata come placeholder per il contenuto della view dinamico, ad esempio:

```
<header>
        <h1>Header</h1>
</header>
<div class="content">
        <ng-view></ng-view>
 </div>
<footer>
         <h5>Footer</h5>
</footer>
```

Così facendo l'header e il footer della pagina rimarranno sempre gli stessi, mentre il contenuto verrà aggiornato (senza ricaricare l'intera pagina) in base al valore corrente

dell'URL.

Per configurare quali template caricare e in che circostanza occorre configurare routeProvider come segue:

```
angular.module('myApp', []). config(['$routeProvider', function($routeProvider) {
  $routeProvider
   .when('/', {
    templateUrl: 'views/home.html',
    controller: 'HomeController'
   });
}]);
```

Si specifica l'url (*"/"* nell'esempio) e che template e controller verranno utilizzati. Per semplicità sono stati riportati solamente alcune proprietà esprimibili, per un elenco esaustivo si rimanda alla documentazione di ngRoute [7].

Passando degli argomenti (con *":"*) attraverso l'URL, AngularJS provvederà ad effettuare il parsing e a creare un array chiamato *routeParams* utilizzabile dal controller che gestisce la view. E poiché l'utente deve solamente scrivere la funzione costruttore del controllore basterà inserirlo tra gli argomenti e AngularJS si preoccuperà di iniettarlo:

```
$routeProvider
  .when('/inbox/:name', {
    controller: 'InboxController',
    templateUrl: 'views/inbox.html'
  })

app.controller('InboxController', function($scope, $routeParams) {
// Qui è abbiamo accesso all'array $routeParams
});
```

3.6.1 UI-Router

UI-Router è una libreria che ci permette di organizzare l'interfaccia in stati invece che semplici url, questo ci da la possibilità di avere view innestate usando la directive *ui-view* invece che *ng-view*.

Infatti come accade usando *ngRoute,* il template definito per lo stato corrente verrà utilizzato per visualizzare il contenuto all'utente, ma usando *ui-route* i template possono a loro volta includere altre *ui-view*.

La configurazione è leggermente diversa, poiché si usa la keyword *"state"* invece di *"when"*.

```
$stateProvider
  .state('inbox', {
    url: '/inbox/:inboxId',
    template: '<div><h1>Welcome to your inbox</h1>\
          <a ui-sref="inbox.priority">Show priority</a>\
          <div ui-view></div>\
          </div>',
controller: function($scope, $stateParams) { $scope.inboxId = $stateParams.inboxId;
} })
  .state('inbox.priority', {
    url: '/priority',
    template: '<h2>Your priority inbox</h2>'
});
```

Abbiamo uno stato figlio (specificato usando il *"."*) chiamato *inbox.priority*. La ui-view dentro la view padre verrà quindi popolata con il template di *inbox.priority*.

Quando abbiamo più ui-view innestate possiamo dare un nome ad ognuna e collegare ad ognuna di esse un template diverso. Ad esempio:

```
<div>
        <div ui-view="filters"></div>
        <div ui-view="mailbox"></div>
</div>
```

E ogni sub-view può avere il suo template e controller:

```
$stateProvider
  .state('inbox', {
    views: {
      'filters': {
          template: '<h4>Filter inbox</h4>',
                        controller: function($scope) {} },
      'mailbox': {
          templateUrl: 'partials/mailbox.html'
      }
});
```

3.7 Dependency Injection

AngularJS permette agli sviluppatori di utilizzare il pattern *dependency injection* e specificare le dipendenze di ogni componente senza preoccuparsi di come reperire quest'ultime. A runtime l'*injector* di AngularJS si occuperà di reperire le dipendenze

specificate e se necessario instanziare i componenti richiesti. Possiamo ad esempio definire un servizio e poi specificare ad AngularJS che un controller ne avrà bisogno a runtime:

```
angular.module('myApp', [])
    .factory('greeter', function() {
        return {
            greet: function(msg) { alert(msg); }
    } })
    .controller('MyController', function($scope, greeter) {
        $scope.sayHello = function() { greeter.greet("Hello!");
    };
});
```

Nel costruttore del controller *"MyController"* è specificata la dipendenza dal servizio *"greeter"*.

3.7.1 Annotazione con inferenza

AngularJS assume che i nomi degli argomenti siano i nomi delle risorse da reperire o instanziare a runtime, e per questo motivo questo approccio può funzionare solo con codice Javascript non minimizzato e non offuscato, poiché AngularJS ha bisogno di effettuare il parsing degli argomenti immutati.

3.7.2 Annotazione esplicita

Si possono comunque indicare manualmente i nomi delle dipendenze, per poter minimizzare il codice e non avere problemi specificando la proprietà *$inject* di una funzione (che corrisponde ad un controller o un servizio).

```
var aControllerFactory =
function aController($scope, greeter) {
   console.log("LOADED controller", greeter);
   // ... Controller
};
aControllerFactory.$inject = ['$scope', 'greeter'];
```

In questo approccio diventa però importante l'ordine con il quale vengono specificati i parametri della funzione, che deve coincidere con l'ordine delle dipendenze specificate nell'array *$inject.*

3.7.3 Annotazione inline

Questo tipo di annotazione è equivalente alla precedente ma permette di definire l'array di dipendenze inline alla definizione della funzione:

```
angular.module('myApp')
  .controller('MyController',
['$scope', 'greeter', function($scope, greeter) {
}]);
```

3.7.4 NgMin

NgMin è un tool che sgrava lo sviluppatore dal definire le dipendenze in maniera esplicita e poter comunque minimizzare il codice, trasforma il seguente codice:

```
angular.module('myApp', [])
        .controller('IndexController', function($scope, $q) {
        });
```

automaticamente in questo:

```
angular.module('myApp', [])
        .controller('IndexController', [ $scope', '$q',
                function ($scope, $q) {
} ]);
```

3.8 Services

Prima di introdurre i *services* vale la pena ricordare che i controllers sono volativi (e lo sviluppatore non ne controlla il ciclo di vita), quindi non idonei a contenere lo stato dell'applicazione. Per questo motivo, e per fornire una modalità di scambio di dati in maniera consistente esistono i *services*.

I *services* sono oggetti singleton instanziati dall'injector di AngularJS e lazy-loaded. Come per le *directives* AngularJS fornisce molti services built-in (ad esempio *$http*) e permette agli sviluppatori di crearne di personalizzati.

Per usare un *service* è sufficiente elencarlo nelle dipendenze del *controller*, *directive* o altro *service* che vuole invocarlo.

3.8.1 Factory

Il modo più semplice per creare e registrare (per essere trovato dall'injector) un nuovo *service* è utilizzare l'API .*factory* di *angular.module*.

```
angular.module('myApp', []) .
factory('UserService', function($http) {
        var current_user;
        return {
                getCurrentUser: function() {
                        return current_user; },
                setCurrentUser: function(user) {
                        current_user = user;
} }
```

La funzione *factory* ha due argomenti: il primo specifica il nome del servizio, il secondo è una funzione che verrà eseguita da AngularJS una volta soltanto per instanziare il service.

3.8.2 Service

La funzione *service* di *angular.module* è simile alla funzione factory, ed ha come quest'ultima due argomenti, dei quali il primo indica il nome mentre il secondo è una funzione costruttore, che verrà invocata da AngularJS usando la keyword *new*.

3.8.3 Provider

Tecnicamente le funzioni *factory* e *service* sono solo delle *"scorciatoie"* per la funzione *provider* che è la vera responsabile per la creazione e registrazione dei vari *"services"*.

Utilizzando però direttamente *provider* è possibile configurare dall'esterno il servizio che si sta definendo attraverso la funzione *config*, iniettando alcuni valori.

3.9 Eventi

Per ottenere un accoppiamento più debole tra i diversi componenti dell'applicazione AngularJS mette a disposizione un meccanismo di creazione e propagazione di eventi.

Rispecchiando la gerarchia degli scope, le due operazioni basilari di propagazione degli eventi sono *$emit* per risalire la catena di scope dai figli al *rootScope* e *$broadcas*t con funzionamento opposto, da uno scope ai suoi scope figli.

Quando si scatena un evento è possibile specificare oltre al nome anche degli argomenti.

Per registrarsi ad un evento si utilizza la funzione *$on* specificando il nome dell'evento di interesse e una funzione di gestione. Un gestore di un evento può anche decidere di interrompere la propagazione dell'evento.

Anche in questo caso AngularJS ha diversi eventi built-in, degno di nota è l'evento *$destroy* che viene emesso sullo scope prima che venga distrutto e permette di pulire l'ambiente (ad esempio distruggendo un timeout che si era registrato). Per un elenco dettagliato degli eventi built-in si rimanda alla documentazione [8].

Attraverso questo meccanismo è inoltre possibile emettere eventi *"globali"* poiché ogni componente dell'applicazione AngularJS può richiedere di farsi iniettare il *rootScope* e poi utilizzarlo con la funzione di *$broadcas*t.

3.10 Moduli

In Javascript non è mai una buona idea utilizzare l'ambiente globale e per questo motivo AngularJS permette di dividere l'applicazione in moduli, nei quali racchiudere le funzionalità.

Quando vogliamo definire un nuovo modulo dobbiamo usare la funzione *angular.module* a due argomenti, il primo specifica il nome del modulo, mentre il secondo è una lista di dipendenze da iniettare.

Per ottenere invece il riferimento ad un modulo utilizziamo la funzione *angular.module* ad un solo parametro, il nome del modulo. Ottenuto il modulo possiamo definire i nostri componenti per quel modulo (controller/service/directive).

3.11 Promises

Una promise è un oggetto che rappresenta il valore di ritorno o un eccezione di di una funzione, può essere vista come un proxy per un oggetto.

Storicamente in Javascript sono state utilizzate funzioni di *callback* per poter lavorare con dati non disponibili in maniera sincrona, come i dati ottenuti da una richiesta XHR, con le *promises* possiamo invece interagire con i dati, supponendo che siano già stati restituiti.

Le *promises* permettono di far sembrare funzioni asincrone come se fossero sincrone, permettendoci di catturare sia i valori di ritorno che le eventuali eccezioni, ma sono sempre eseguite in maniera asincrona, per cui possono essere usate senza preoccuparsi del fatto che possano bloccare l'esecuzione.

AngularJS fornisce il *service $q* per semplificare la gestione delle *promises*, una volta creata un *promise* con il metodo *$q.defer()* possiamo utilizzare il metodo *resolve(value)* nel caso l'esecuzione sia andata come previsto e il metodo *reject(reason)* per notificare invece l'errore. Un ulteriore metodo è *notify(value)* che permette di mandare una notifica (ad esempio se l'esecuzione richiede un tempo lungo potremmo notificare il progresso attraverso questo metodo). Sia nel caso di successo che nel caso di errore la funzione ritornerà la promise creata al chiamante che può interagire con il risultato attraverso la funzione *then*().

La funzione *then*() accetta tre parametri, le funzioni che saranno eseguite in caso di successo, errore e ricezione di notifica. È importante notare che le funzioni di successo o fallimento saranno invocate in maniera asincrona appena la *promise* è risolta, (solamente una delle due) e con un singolo parametro che indica il risultato in un caso o il motivo del fallimento nell'altro. Mentre la funzione di notifica può essere invocata anche più volte.

Le *promises* sono profondamente integrate in AngularJS, infatti anche i services built-in come *$http* le utilizzano per restituire i risultati ai chiamanti.

4 Realizzazione di una Applicazione Ibrida

4.1 Strumenti Utilizzati

4.1.1 Cordova e plugins

Per realizzare l'applicazione è stato utilizzato Cordova 3.4.1 ed i seguenti plugin:

- SocialSharing-PhoneGap-Plugin: per permettere all'utente di condividere le foto scattate. Documentazione ed ulteriori informazioni sono presenti presso la repository github: https://github.com/EddyVerbruggen/SocialSharing-PhoneGap-Plugin
- Cordova Camera Plugin: per avere accesso alla fotocamera del dispositivo. Documentazione e ulteriori informazioni nella pagina del plugin: http://plugins.cordova.io/#/package/org.apache.cordova.camera
- Cordova Device Plugin: fornisce una descrizione dell'hardware e del software del dispositivo. Documentazione e ulteriori informazioni nella pagina del plugin: http://plugins.cordova.io/#/package/org.apache.cordova.device
- Cordova File Plugin: per memorizzare nel filesystem locale le foto scattate. Documentazione e ulteriori informazioni nella pagina del plugin: http://plugins.cordova.io/#/package/org.apache.cordova.file
- Cordova Geolocation Plugin: utilizzato per determinare la posizione dell'utente. Documentazione e ulteriori informazioni nella pagina del plugin: http://plugins.cordova.io/#/package/org.apache.cordova.geolocation
- Cordova StatusBar Plugin: permette di avere un'integrazione con la status bar di iOS 7, in modo tale da avere stesso look and feel di una applicazione nativa. Documentazione e ulteriori informazioni nella pagina del plugin: http://plugins.cordova.io/#/package/org.apache.cordova.statusbar

4.1.2 Ionic

Per realizzare l'interfaccia grafica dell'applicazione è stato utilizzato Ionic [9] un framework

open source che utilizza AngularJS e fornisce agli sviluppatori componenti (HTML/CSS/Javascript) ed alcuni plugin per cordova.

I componenti forniti hanno *directive* da utilizzare nelle view HTML ed *API* da utilizzare nei corrispondenti controller.

Ionic è un framework molto giovane e nella fase iniziale di realizzazione e diffusione. All'inizio della scrittura dell'applicazione era stata da poco pubblicata la prima beta pubblica, ad oggi è stata rilasciata la sesta beta.

Per l'applicazione sono stati utilizzati i seguenti componenti:

- ion-nav-view: componente che tiene traccia della storia di navigazione dell'utente all'interno dell'applicazione fornendo animazioni tra una view ed un altra. Utilizza UI-Router per gestire i vari stati dell'applicazione
- ion-nav-bar: per gestire l'header di ogni view inserendo titoli e pulsanti personalizzati
- ion-tabs: permette di utilizzare il layout *a tabs* ed associare ad ogni *tab* una diversa view
- ionicModal: per la gestione di view modali
- ionicPopup: per notificare l'utente

4.1.3 Estensioni AngularJS

Sono state utilizzate anche le seguenti estensioni per AngularJS:

- Angular-google-maps : fornisce *directives* per utilizzare google maps all'interno di applicazioni AngularJS [10]
- Bindonce: utilizzato quando possibile per aumentare le performances [11]

A causa del two-way data binding di AngularJS la directive *ng-repeat* causa dei deterioramenti di performances poiché AngularJS utilizza un meccanismo di *dirty checking* per controllare se sono state effettuate modifiche negli elementi. Utilizzando bindonce viene “rotto” questo two-way data binding per gli elementi specificati, e dopo aver fatto il rendering degli elementi nella view AngularJS non controllerà eventuali modifiche. Ho utilizzato bindonce nella lista di fotografie visualizzate nella timeline.

Le future versioni di AngularJS dovrebbero utilizzare il metodo Object.observe() (non ancora supportato da tutti i browser) per velocizzare queste operazioni ed il team di

AngularJS ha dichiarato di avere già ottenuto un incremento delle prestazioni del 20-40% [12].

4.2 Struttura dell'applicazione

Per la scrittura dell'applicazione ho realizzato tre moduli AngularJS, ed ho organizzato i file in modo da avere un componente in ogni file separato, definendo tutti i moduli nel file *app.js* e utilizzando il metodo *getter angular.module* per ottenere in ogni file il modulo desiderato per poterci definire e registrare il componente.

4.2.1 Services

Il modulo MyPhotoDiary.services è così definito:

```
angular.module('MyPhotoDiary.services',[]);
```

ed è richiamato in ogni file dentro la cartella scripts/services per poter definire un servizio in ogni file separato (ho fatto questa scelta perché preferisco avere più file piccoli ognuno con limitate responsabilità, poi in fase di deploy i fari file *.js* vengono concatenati e il file risultante viene minimizzato).

Ho realizzato i seguenti *services*:

- CameraService: utilizza le API fornite dal plugin Cordova Camera per scattare fotografie e restituisce una *promise*.
- GeolocationService: utilizza le API fornite dal plugin Cordova Geolocation plugin per ottenere le coordinate gps e restituisce una *promise*.
- PicturesService: gestisce le fotografie dell'applicazione, utilizza il *local storage* per memorizzare la descrizione, il path su filesystem locale e le coordinate di ogni fotografia. Per l'operazione di rimozione di fotografie dal filesystem utilizza StorageSettings e restituisce una *promise* (le operazioni su filesystem in Javascript sono asincrone). Emette l'evento *NewPicture* nel *rootScope* ogni volta viene realizzata una nuova fotografia, in modo tale che qualunque controller possa gestire questo evento una volta sottoscritto. (AngularJS rilascia un controller se l'utente non è nella view correlata, quindi ho scelto di avere un evento globale in modo tale da

poterlo gestire indipendentemente dalla view corrente dell'utente)

- SettingsService: memorizza le preferenze dell'utente utilizzando il *local storage*.
- StorageService: utilizza le API fornite dal plugin Cordova File e permette di memorizzare e cancellare le foto dal filesystem. Per le operazioni restituisce una *promise*.

4.2.2 Controllers

Il modulo MyPhotoDiary.controllers è così definito:

```
angular.module('MyPhotoDiary.controllers',['MyPhotoDiary.services']);
```

e come già spiegato per il modulo relativo ai services è richiamato all'interno di ogni file dentro la cartella scripts/controllers.

Sono stati realizzati i seguenti controllers:

- MainNavBarController: gestisce la navigation bar presente in ogni view dove è situato un pulsante per scattare una nuova fotografia. Utilizza i services *CameraService*, *GeolocationService*, *PicturesService* e *StorageService* per catturare e memorizzare la nuova foto. Usa anche *ionicModal* per mostrare all'utente una view modale per confermare o scartare la nuova fotografia, e *ionicPopup* per visualizzare eventuali errori.
- PlacesController: responsabile della creazione dei marker da visualizzare sulla mappa. Utilizza *GeolocationService* per centrare la mappa nella posizione corrente dell'utente e *PicturesService* per recuperare le foto da mostrare. Gestisce l'evento *NewPicture* reindirizzando l'utente alla view timeline.
- SettingsController: permette di scegliere di quante foto effettuare il prefetch per essere visualizzate nella timeline e di eliminare tutte le foto dell'applicazione. Usa *PicturesService* per calcolare quante fotografie sono attualmente memorizzate e *SettingsService* per memorizzare le preferenze utente. Anche questo controller gestisce l'evento *NewPicture* reindirizzando l'utente alla view timeline.
- TimelineController: gestisce la condivisione e l'eliminazione della foto selezionata nella corrispettiva view. Inoltre carica ulteriori foto e le aggiunge alla view non

appena l'utente raggiunge il limite inferiore dello scroll della view, infatti per motivi di performances non vengono immediatamente caricate tutte le foto nella view. Utilizza *PicturesService* per richiedere le foto memorizzate mano a mano che l'utente scorre la view e *SettingsService* per sapere di quante foto effettuare il prefetch. Gestisce l'evento *NewPicture* aggiungendo la nuova foto in cima alla view e tornando all'inizio della lista di fotografie.

4.2.3 Modulo principale

Il modulo MyPhotoDiary è così definito:

```
angular.module('MyPhotoDiary',  ['ionic', 'MyPhotoDiary.controllers',
'MyPhotoDiary.services', 'google-maps', 'pasvaz.bindonce'])
```

Ed è configurato nel file *app.js* specificando per ogni stato dell'applicazione (usando UI-router) il rispettivo nome, url, template e controller.

Gli stati sono:

- tabs: stato astratto con template *tabs.html*. Uno stato astratto non può mai essere direttamente attivato, sarà attivato uno dei suoi stati figli.
- tabs.places: mappato su url */places* utilizza il template *places.html* ed il controller *PlacesController.*
- tabs.timeline: mappato su url */timeline* utilizza il template *timeline.html* ed il controller *TimelineController*.
- tabs.settings: mappato su url */settings* utilizza il template *settings.html* ed il controller *SettingsController*.

All'avvio dell'applicazione viene impostato lo stato tabs.timeline.

4.2.4 Views

I template di ogni view sono salvati all'interno della cartella *templates* e sono:

- confirmPhoto: una view modale che visualizza la foto appena scattata e permette di

confermarne il salvataggio oppure di scartarla. E' gestita da *MainNavController*.

- places: contiene una google map e un marker per ogni foto nella posizione in cui è stata scattata. Utilizza la direttiva <google-map> fornita da *Angular-google-maps*.
- settings: attraverso uno slider permette all'utente di scegliere il numero di fotografie di cui fare il prefetch nella timeline ed ha un pulsante per eliminare tutte le foto memorizzate.
- tabs: visualizza tre tab, uno per lo stato tabs.timeline, uno per lo stato tabs.places e uno per lo stato tabs.settings.
- timeline: mostra in una lista in ordine cronologico le fotografie memorizzate e sotto ognuna di essa due pulsanti, uno per eliminare ed uno per condividere la foto.

4.3 Tools

Per velocizzare il workflow sono stati utilizzati alcuni tool open source, ne riporto in seguito una breve descrizione per ognuno.

4.3.1 Grunt

Grunt [13] è un task runner scritto in Javascript con molti tool e plugin resi disponibili dalla comunità, tra i plugin che ho utilizzato segnalo *bower-install* che inietta le dipendenze gestite con *bower* nell'HTML, *concat* che concatena diversi fine in uno unico, *ngmin*, *cssmin* che comprime i file CSS, *uglify* per minimizzare il codice Javascript e *htmlmin* che minimizza il codice HTML.

È installabile attraverso npm (package manager di node.js).

4.3.2 Bower

Bower [14] è un package manager per componenti web. Per l'applicazione ho specificato come dipendenze *AngularJS*, *Ionic*, *angular-google-maps* e *angular-bindonce.* Bower provvede a scaricare in locale l'ultima versione (o la versione specificata) delle dipendenze, in modo tale che facendone il deploy sul dispositivo non sia necessaria alcuna connettività di rete.

Anche bower è reperibile attraverso npm.

4.3.3 Yeoman

Yeoman [15] è un tool che automatizza il setup degli strumenti precedentemente descritti grazie a *generator* resi disponibili dalla comunità (attualmente ci sono più di 700 *generators*).

Per lo sviluppo di questa applicazione ho utilizzato il *generator* chiamato *generator-ionic* disponibile in maniera open source all'indirizzo https://github.com/diegonetto/generator-ionic.

Anche yeoman è reperibile con npm.

5 Realizzazione di una Applicazione Nativa

5.1 Content Provider

In Android i *Content Provider* sono utilizzati per disaccoppiare il layer di persistenza dal layer applicativo. Ogni applicazione ha accesso ad un database privato, usando i C*ontent Provider* è possibile invece condividere i dati con altre applicazioni, ed è inoltre possibile eseguire query asincrone su di essi per ottenere un'applicazione *responsive*. Android ha diversi *Content Provider built-in* utilizzabili dagli sviluppatori, ad esempio *Media Store*, *Contacts*, *Calendar* e *Call Log*. È inoltre possibile definire nuovi content provider.

5.1.1 PicturesProvider

Per l'applicazione ho realizzato un provider per memorizzare e ottenere i dati relativi alle foto scattate dall'utente. Il provider al suo interno utilizza un database *SQLite* gestito estendendo la classe di *SQLiteOpenHelper* fornita da Android.

Il database ha una singola tabella che memorizza il path su filesystem dell'immagine, la descrizione inserita dall'utente, le coordinate geografiche ed un id univoco.

È possibile interrogare il provider per ottenere tutte le immagini attualmente memorizzate attraverso l'url *content://com.danielecampogiani.picturesprovider/pictures*, mentre se si desidera ottenere informazioni su una specifica foto è sufficiente inserire in fondo all'url 'id della foto : *content://com.danielecampogiani.picturesprovider/pictures/id*.

Dopo aver fatto un match sull'url richiesto al provider (per discriminare se sono richieste tutte le immagini, o una specifica) il content provider delega il reperimento dei dati al database, e nel caso di modifiche notifica i *Content Resolver* permettendo così di ottenere delle callback lato applicativo senza dover rieseguire query (come spiegato più avanti).

5.2 Activity

Una *Activity* rappresenta una schermata mostrata ad un utente solitamente occupando tutto lo schermo del dispositivo, per creare un'attività è sufficiente estendere la classe *Activity* e definire i metodi di callback richiamati dal sistema Android per gestire il ciclo di vita dell'attività.

5.2.1 MainActivity

Per l'applicazione ho creato una sola *Activity*, poiché utilizzo all'intero di essa diversi *Fragment* che vengono automaticamente mostrati/nascosti utilizzando un' *actionBar* a *tabs*.

Per gestire le transazioni di *Fragment* ho creato la classe *TabListener* che si occupa di instanziare quando necessario il *fragment* che gestisce e di aggiungerlo e rimuoverlo dallo schermo utilizzando *FragmentTransaction* quando l'utente seleziona il tab relativo.

Ho inoltre aggiunto un menu alla actionBar con un pulsante che quando selezionato crea *l'intent Mediastore.Action_Image_Capture* specificando un uri negli *extra* dell'*intent* poi *"lancia"* questo intent aspettando la restituzione del controllo con il metodo *startActivityForResult*. Così facendo verrà lanciata l'applicazione scelta dall'utente per scattare fotografie, e una volta che la foto è stata fatta sarà memorizzata nell'uri specificato negli extra dell'*intent* e il controllo tornerà all'attività *MainActivity*.

Ogni qual volta possibile ho eseguito le operazioni all'interno di *AsyncTask*, questo permette di eseguire codice in background, senza quindi utilizzare il thread dedicato alla UI, e sincronizzare l'interfaccia grafica al termine della loro esecuzione.

La classe *AsyncTask* si occupa della creazione, gestione e sincronizzazione di thread, per cui lo sviluppatore si deve preoccupare solamente di scrivere la logica applicativa.

Nello specifico ho utilizzato *SavePictureAsyncTask* per leggere le coordinate geografiche dall'immagine restituita dall'applicazione fotocamera e per memorizzare l'immagine nel content provider, e *NewPictureAsyncTask* per le operazioni di I/O necessarie per creare il nuovo file sul filesystem.

5.3 Fragments

I *fragment* vengono utilizzati per creare UI dinamiche e flessibili che si adattano a diversi schermi. Ogni *fragment* è un componente a se stante (con un suo ciclo di vita) utilizzabile da diverse activity. Per creare un nuovo *fragment* è sufficiente estendere la classe *Fragment*.

5.3.1 TimelineFragment

È il fragment collegato al tab *"Timeline"* e mostra in una lista (estende la classe *ListFragment)* le foto scattate in ordine cronologico. Sotto ogni foto è presente la

descrizione e due pulsanti, uno per cancellare la foto e uno per condividerla.

Per ottenere le foto da mostrare utilizzo un *loader* di *cursor* poiché le operazioni su database possono essere *time-consuming*.

I *loaders* sono un meccanismo per caricare dati in maniera asincrona e per monitorare le sorgenti di dati. Possono essere utilizzati per qualunque tipo di dato, nell'applicazione li ho utilizzati per caricare dei *cursor* ossia il risultato di query su database in Android.

Per ottenere un comportamento asincrono all'interno del fragment ho implementato l'interfaccia *LoaderManager.LoaderCallbacks<Cursor>* realizzando i metodi:

- onCreateLoader: responsabile di creare il loader (usando la classe *CursorLoader* di Android) specificando i parametri della query desiderata.
- onLoadFinished: richiamato quando la query asincrona restituisce dei risultati che vengono passati come parametro in questo metodo. Il metodo è invocato anche ogni qual volta cambiano i dati di interesse per la query specificata nel metodo onCreateLoader.
- onLoaderReset: per resettare il loader.

Per lo specifico fragment, il loader creato interroga il *PicturesProvider* richiedendo l'id, la descrizione e il path su filesystem di ogni foto.

Nel metodo *onLoadFinished* aggiorno la grafica mostrando i risultati utilizzando come *adapter* della lista la classe *TimelinePicturesAdapter*.

5.3.2 PlacesFragment

Il fragment collegato al tab *“Places”* mostra sullo schermo una google map centrata sulla posizione attuale dell'utente e ogni foto nelle coordinate geografiche nelle quali è stata scattata.

Anche in questo *fragment* ottengo dati in maniera asincrona implementando l'interfaccia *LoaderManager.LoaderCallbacks<Cursor>* richiedendo a *PicturesProvider* la latitudine, la longitudine e il path su filesystem per ogni foto.

Una volta ottenuti i dati provvedo ad aggiornare l'interfaccia grafica utilizzando l'*AsyncTask AddMarkersAsyncTask* che elabora le foto restituite dal loader creando dei *marker* da mostrare sulla mappa.

Per ottenere la posizione attuale dell'utente utilizzo la classe *LocationClient* e implemento le interfacce di Android *GooglePlayServicesClient.OnConnectionFailedListener* e *GooglePlayServicesClient.ConnectionCallbacks* e aggiornando l'interfaccia nella callback che restituisce la posizione dell'utente.

5.4 Adapter

Gli *adapter* sono usati per effettuare il binding tra i dati e le view che estendono *AdapterView*, come ad esempio *ListFragment* (che ho esteso per realizzare *TimelineFragment*). Gli adapter sono responsabili di creare le view figlie per mostrare i dati.

5.4.1 TimelinePicturesAdapter

Classe che estende *CursorAdapter* e per ogni cursor (una foto) utilizza una *ImageView* per visualizzare su schermo la foto, una *TextView* per mostrare la descrizione e due pulsanti, uno per condividere la foto e uno per cancellarla. Il pulsante per la cancellazione mostra inoltre un *AlertDialog* per chiedere conferma all'utente una volta selezionato. Il pulsante per la condivisione invece lancia l'*intent Intent.Action_Send* specificando come tipo di dato *image/jpeg*, così facendo l'utente può scegliere con quale applicazione condividere il file.

Anche in questo caso ho utilizzato un *AsyncTask* per non svolgere operazioni potenzialmente lunghe nel thread della UI. *DeletePicturesAsyncTask* rimuove l'immagine dal content provider e dal filesystem.

5.5 Layouts

I *Layouts* permettono di separare il layer di presentazione dalla logica di business, specificando l'interfaccia grafica in file XML invece che crearla via codice.

Una volta definiti i file XML, questi possono essere *"inflated"* nei rispettivi componenti (*Activity* e *Fragment*) usando il metodo *setContentView* per le *Activity* e il metodo *Inflator.inflate* per i Fragment (l'oggetto *Inflator* è passato al *fragment* nel metodo *onCreateView*).

L'utilizzo di *layouts* è considerato una *best practice* in Android, perché è un meccanismo che permette di creare interfacce ottimizzate per diversi tipi di hardware, ad esempio diverse dimensioni di schermo, o la presenza o meno di una tastiera o touchscreen.

Per l'applicazione ho realizzato i seguenti layout:

- activity_main.xml : contiene un holder in cui collocare i fragment selezionati scegliendo un tab dell'actionBar.
- dialog_description_layout.xml : utilizzato per creare un *AlertDialog* in cui richiedere all'utente una descrizione testuale della foto appena scattata.
- fragment_places.xml : usato dal *fragment PlacesFragment* contiene al suo interno un *MapFragment* per mostrare la google map sullo schermo.
- fragment_timeline.xml : usato dal *fragment TimelineFragment* contiene al suo interno una *ListView* per mostrare tutte le fotografie.
- timeline_picture_layout : usato da TimelinePicturesAdapter responsabile di mostrare ogni singola fotografia nella lista sopra descritta. Per ogni foto mostra l'immagine vera e propria, la descrizione e due pulsanti, uno per cancellare la foto e uno per condividerla.

6 Confronto

6.1 Dimensione APK

6.1.1 Applicazione Ibrida

Per quanto riguarda l'applicazione ibrida il file *apk* è stato generato due volte, una volta senza minimizzare il codice e una volta minimizzandolo ottenendo dei risultati non aspettati, come mostrato in tabella:

	Codice non minimizzato	Codice minimizzato
Cartella www	10,5 MB	1,1 MB
Apk signed	3,7 MB	704 KB

Come si può notare, semplicemente minimizzando il codice otteniamo un file circa cinque volte più piccolo. Una volta installata sul dispositivo l'applicazione occupa 1,2 MB.

6.1.2 Applicazione Nativa

Il file apk generato dal codice dell'applicazione nativa è di 2,1 MB. Nel codice utilizzo una libreria esterna per la gestione delle animazioni, senza usare questa libreria il file apk è di 2 MB.

Una volta installata l'applicazione occupa nel dispositivo 6,4 MB.

6.2 Linee di codice

Trattandosi di applicazioni scritte in linguaggi diversi il confronto è da considerarsi puramente indicativo, inoltre non viene preso in considerazione il codice di librerie utilizzate ma solo il codice effettivamente scritto per l'applicazione.

6.2.1 Applicazione Ibrida

Sono stati scritti 10 file in Javascript per un totale di 649 righe di codice, un file CSS di 5 righe e 6 file HTML per un totale di 164 righe. Sommando tutti i file otteniamo 818 righe di codice.

6.2.2 Applicazione Nativa

Per quanto riguarda l'applicazione nativa sono stati scritti 6 file Java per un totale di 828 righe di codice e 8 file XML per un totale di 180 righe di codice. Nel complesso sono state scritte 1008 righe di codice.

6.3 Tempo necessario per scattare e memorizzare un'immagine

I seguenti risultati sono stati ottenuti utilizzando un Nexus 7 2013 con Android in versione 4.4.2 (custom rom ParanoidAndroid 4.3-Beta7) dotato di CPU quad core a 1.5 GHz e 2GB di RAM.

Le foto in esame sono state scattate con la fotocamera frontale.

Ho calcolato due intervalli di tempo: il primo (chiamato *acquisizione* nelle successive tabelle) è il lasso di tempo che trascorre tra la pressione del tasto per scattare una fotografia e la restituzione del controllo all'applicazione (sia nativa che ibrida) con la foto appena scattata, e il secondo (chiamato *salvataggio* nelle tabelle) è il tempo necessario per memorizzare l'immagine nel filesystem e nel database dell'applicazione.

Ho eseguito questo test sia con le due applicazione *“pulite”* (ossia senza nessuna foto ancora memorizzata) sia con cinque foto memorizzate senza riscontrare però cambiamenti degni di essere considerati.

Inoltre ho eseguito il test considerando il fatto che la foto che stessi scattando fosse la prima (dall'apertura dell'applicazione) oppure se fosse una foto successiva, e anche in questo caso non ci sono stati cambiamenti degni di nota.

6.3.1 Applicazione Ibrida

In tabella sono riportati in millisecondi i due intervalli sopra descritti nel caso di applicazione con zero foto memorizzate e prima foto scattata dall'apertura dell'applicazione

Test	Acquisizione	Salvataggio
1	3648	161
2	3495	213
3	3702	261
4	3613	205
5	3552	145

In media sono necessari 3602 millisecondi per acquisire un immagine e 197 millisecondi per memorizzarla.

Non sono riportati i test con applicazione con cinque foto memorizzate oppure il caso in cui la foto scattata non fosse la prima dall'apertura dell'applicazione perché hanno differenze non significative rispetto al caso riportato.

6.3.2 Applicazione Nativa

Analogamente all'applicazione Ibrida riporto in tabella i valori dei due intervalli ottenuti in cinque test

Test	Acquisizione	Salvataggio
1	3221	57
2	3335	51
3	3317	40
4	3134	53
5	3266	51

Anche in questo caso non ho riscontrato cambiamenti degni di nota effettuando test con applicazione con cinque foto memorizzate, o scattando una foto che non fosse la prima dall'apertura dell'applicazione, per cui riporto solo il caso di applicazione *“pulita”* e prima foto scattata.

In media sono necessari 3254,6 millisecondi per scattare una fotografia e 50,4 millisecondi per memorizzarla nel filesystem e nel database.

6.3.3 Analisi dei risultati

Per quanto riguarda l'acquisizione dell'immagine l'applicazione ibrida è leggermente più

lenta, invece per quanto riguarda la memorizzazione impiega circa quattro volte il tempo impiegato dall'applicazione nativa.

6.4 Utilizzo di memoria

Ho misurato l'utilizzo di RAM delle due applicazioni, e come si può notare in tabella l'applicazione ibrida richiede una quantità di memoria decisamente maggiore.

6.4.1 Apertura

È qui di seguito riportata la RAM occupata all'apertura, al variare del numero di immagini memorizzare nell'applicazione:

Fotografie Salvate	Applicazione Ibrida	Applicazione Nativa
2	106 MB	18 MB
3	110 MB	18 MB
4	107 MB	18 MB

L'applicazione ibrida richiede sempre almeno 106 MB di memoria, mentre l'applicazione nativa ne usa quasi un sesto.

6.4.2 Scorrimento nella Timeline

Riporto in tabella l'utilizzo di RAM delle due applicazioni dopo aver visualizzato tutte le foto nella timeline (scorrendo la view fino in fondo e poi tornando in cima), anche in questo caso al variare di foto memorizzate:

Fotografie Salvate	Applicazione Ibrida	Applicazione Nativa
2	108 MB	23 MB
3	119 MB	33 MB
4	123 MB	33 MB

L'applicazione nativa utilizza quindi circa un quarto della memoria richiesta dall'applicazione ibrida.

6.4.3 Mappa

Spostandosi nella view che mostra le foto sulla mappa abbiamo questi dati:

Fotografie Salvate	Applicazione Ibrida	Applicazione Nativa
2	132 MB	45 MB
3	133 MB	42 MB
4	136 MB	44 MB

Anche in questo caso l'applicazione ibrida risulta molto più *"pesante"*, richiedendo circa tre volte la memoria necessaria all'applicazione nativa.

6.5 Considerazioni finali

Il modello applicativo utilizzato per realizzare l'applicazione ibrida (utilizzando il framework AngularJS) è molto più semplice del modello usato da Android per la realizzazione di applicazioni native. In AngularJS oltre alle view HTML, le directive e i controller esistono i servizi, componenti software singleton che si occupano di tutto il resto. In Android invece esistono molto più componenti.

Risulta quindi più facile all'inizio sviluppare un'applicazione ibrida, ma una volta capiti i diversi componenti che Android fornisce, si riesce ad avere un'applicazione con una miglior struttura, e a scrivere meno codice perché Android realizza già molte funzionalità di base.

Per quanto riguarda i tools forniti agli sviluppatori non c'è paragone, il supporto fornito a sviluppatori Cordova è praticamente inesistente e limitato all'uso del terminale attraverso lo strumento chiamato *Cordova Command-line Interface*. Per questo motivo mi sono trovato costretto ad utilizzare una serie di tool esterni per migliorare il workflow come *Grunt*, *Bower* e *Yeoman*. Anche il debug è più complicato trattandosi di codice interpretato.

Per lo sviluppo dell'applicazione nativa ho invece utilizzato *Android Studio* una soluzione completa fornita da google per lo sviluppo, il test e il deploy delle applicazioni.

Confrontando le performances delle due applicazioni risulta nettamente vincitrice l'applicazione nativa, sia per il consumo di memoria che per la fluidità ottenuta. L'applicazione ibrida in particolare ha uno scroll molto più lento nelle liste.

Punto a favore dell'applicazione ibrida invece è la possibilità di riutilizzo del codice scritto, infatti i componenti scritti possono essere utilizzati sia in una applicazione mobile/ibrida che un normale sito web. Ovviamente in un normale sito web non si hanno a disposizione le API fornite da Cordova, ma i componenti che non le usano possono essere *“portati”* anche su desktop, mi sono infatti spesso trovato a provare l'applicazione su browser desktop prima di effettuare il deploy su dispositivo per controllare velocemente le modifiche apportate. Nello specifico l'unica API che non ho potuto utilizzare da desktop è quella di accesso alla fotocamera, mentre per la persistenza e la geo-localizzazione non ho avuto problemi poiché Chrome fornisce le stesse API che Cordova implementa su dispositivi mobili. Vale la pena ricordare inoltre ricordare che una applicazione scritta utilizzando Cordova può essere utilizzata su più piattaforme mobile, mentre nel caso di applicazione nativa si è limitati ad una sola piattaforma.

Android pur utilizzando codice Java forza lo sviluppatore a realizzare componenti estendendo quelli forniti dalla piattaforma, per cui questi componenti non possono essere *“portati”* su una normale *JVM* desktop.

Inoltre per la scrittura di una applicazione nativa, il produttore della piattaforma fornisce uno stack completo di supporto (dall'interfaccia grafica all'accesso all'hardware) mentre Cordova fornisce solo un set di API per l'accesso al dispositivo e nessun supporto per la realizzazione di UI, e ho dovuto quindi utilizzare un framework esterno.

7 Bibliografia

1. Cordova, platform support : http://cordova.apache.org/docs/en/3.4.0/guide_support_index.md.html#Platform%20Support
2. Apache Cordova 3 Programming, Capitolo 1 , “The What, How, Why and More of Apache Cordova” : http://www.cordovaprogramming.com/
3. Apache Cordova 3 Programming, Capitolo 13, “Creating Cordova Plugins”: http://www.cordovaprogramming.com/
4. Post google+ : https://plus.google.com/+IgorMinar/posts/DRUAkZmXjNV
5. Directive components in ng : https://docs.angularjs.org/api/ng/directive
6. Creating custom directives : https://docs.angularjs.org/guide/directive
7. $route : https://docs.angularjs.org/api/ngRoute/service/$route
8. Scope : https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$on
9. Ionic advanced HTML5 hybrid mobile app framework : http://ionicframework.com/
10. Google Maps for AngularJS : http://angular-google-maps.org/
11. Bindonce, zero watches binding for AngularJS : https://github.com/Pasvaz/bindonce
12. Object.observe() and AngularJS : https://mail.mozilla.org/pipermail/es-discuss/2012-September/024978.html
13. Grunt, the Javascript task runner : http://gruntjs.com/
14. Bower, a package manager for the web : http://bower.io/
15. Yeoman, modern workflows for modern webapps : http://yeoman.io/