Integrating a map view with Ionic

The following is an excerpt from "Developing an Ionic Edge," by Bleeding Edge Press.

At this point in the book, the user of our Trendicity example app has been asked to provide a city and optional state or country where the favorite location should be. Then, by using the `GeolocationService` we've built for Trendicity, that input was converted to an object with latitude and longitude properties, which were stored in the favorite object.

These location details were needed to center the map view on to that particular location and retrieve nearby Instagram posts, which are shown as markers on the map. This map view is what we will be focusing on in this excerpt.

Picking an AngularJS directive for Google Maps

Ionic does not have a map directive built into the framework, but thankfully we can use a community directive without any issues. In the case of Trendicity, we selected a suitable Angular-based library that met three of our needs:

1. Google Maps as a tile service
2. Fast performance on mobile devices
3. No need for offline capabilities

To be completely honest, this was probably the hardest part for us when building Trendicity. There is a wide variety of Google Maps related directives out in the wild. Their functionality defers from very basic `new google.maps.Map()` calls to complete asynchronous Google Maps v3 API loading and map initialization with nested directives for markers and polygons.

We will be covering a selection of three Google Maps Angular projects, and how you can easily implement a third-party library in your existing Ionic project in this excerpt.

The candidates

Let's take a look at the candidate libraries.

ANGULAR-GOOGLE-MAPS

When searching online for 'angular google maps' you might come across the `angular-google-maps` library as one of your first results. It is another project by the Angular UI team and it's very well documented. The project is very much alive and has a large community using it. But then again, the usage of this particular directive and its nested directives for markers and marker windows can be tough to understand.

UI-MAPS

Confusingly, the same Angular UI team also has the `ui-maps` project. When we're snooping around on the GitHub repositories page we quickly notice there is something wrong with this project. As of this moment, the project has some open pull requests and was last updated over a year ago. That didn't give us much confidence in using it. Based on some community reports and a lengthy topic in the issues section, we discovered that the Angular UI team discontinued this project in favor of `angular-google-maps`.

LEAFLETJS

Leaflet is something entirely different. It existed before we even started thinking about building hybrid apps with Angular. Leaflet is a widely used open source JavaScript library used to build web mapping applications. It supports most mobile and desktop platforms, supporting HTML5, and CSS3. Alongside OpenLayers and the Google Maps API, it is one of the most popular JavaScript mapping libraries, and is now used by major web sites such as FourSquare, Pinterest, and Flickr.

AND THE WINNER IS

... `angular-google-maps`. Why not Leaflet? Keeping in mind that we wanted to demonstrate implementing a Google Maps powered map to interact with it, `angular-google-maps` was the best choice. When we switched out the Open Street Maps tile provider with Google Maps when using Leaflet, there was a huge performance drop on both mobile and desktop. Considering the performance and the fact that we have no need to display tiles from an offline data source, Leaflet is great at that, we decided to go with `angular-google-maps`.

Note that there are plenty other Angular projects that implement Google Maps. This is just a selection of the many libraries that you can find out there.

Creating the geolocation utility service

Before we get started using the `angular-google-maps` library, we first take a look at the `GeolocationService`. In the `GeolocationService` we have implemented three straight forward geolocation related methods.

One of these methods is the `getDefaultPosition()` method. As every good method name already suggests, this is a way to get the default location object. We use the method to return the fallback position, when retrieving the user's actual position fails for any reason. There's no rocket science here.

Implementing ngCordova and the $cordovaGeolocation service

Next up is the `getCurrentPosition()` method. This method returns a promise that will retrieve the user's current position using the `$cordovaGeolocation` service.

```
// GeolocationService
// /www/js/services/geolocation.js

this.getCurrentPosition = function () {
    var defer = $q.defer();

    $ionicPlatform.ready(function () {
        var posOptions = {timeout: 10000, enableHighAccuracy:
false};

        $cordovaGeolocation
            .getCurrentPosition(posOptions)
            .then(
                function (position) {
                    $log.debug('Got geolocation');
                    defer.resolve(position);
                },
                function (locationError) {
                    $log.debug('Did not get geolocation');

                    defer.reject({
                        code: locationError.code,
                        message: locationError.message,
                        coords: fallbackPositionObject
                    });
                }
            );
    });

    return defer.promise;
};
```

To mimic the same behavior as the `$cordovaGeolocation` service, we create a deferred object to return its promise later on. After that we will call `$cordovaGeolocation.getCurrentPosition()` to retrieve the users current location by leveraging their GPS hardware using the installed Cordova plugin from the ngCordova project. Following the instructions from **ngCordova website** we have installed the `org.apache.cordova.geolocation` Cordova plugin as follows:

```
$ ionic plugin add org.apache.cordova.geolocation
```

And of course we have installed the `ngCordova` project using Bower:

```
$ bower install --save ngCordova
```

$ionicPlatform.ready()

After the plugin is installed, we are able to fully leverage the `$cordovaGeolocation` service. A keen reader would have noticed by now that the `getCurrentPosition()` method of the `GeolocationService` is basically the same as the one from `$cordovaGeolocation`. Why bother creating a method that does exactly the same thing? There is one important difference and it's wrapped around `$cordovaGeolocation`: `$ionicPlatform`.

As you can see, we have wrapped `$cordovaGeolocation` inside of an `$ionicPlatform.ready()` callback. By doing so the `$cordovaGeolocation` service will start looking up the user's location when the device is actually ready to do so. When the application is within a *webview* (Cordova), it will fire the callback once the device is ready. If the application is within a web browser, it will fire the callback after the `window.load` event.

Converting addresses to geolocation objects using the Google Maps Geocode API

Next up is the `addressToPosition()` method. You may know already that it is useful when you want to convert a certain address as a string to a geolocation object. The functionality is pretty straight forward: address as a string goes in, and geolocation comes out as a promise.

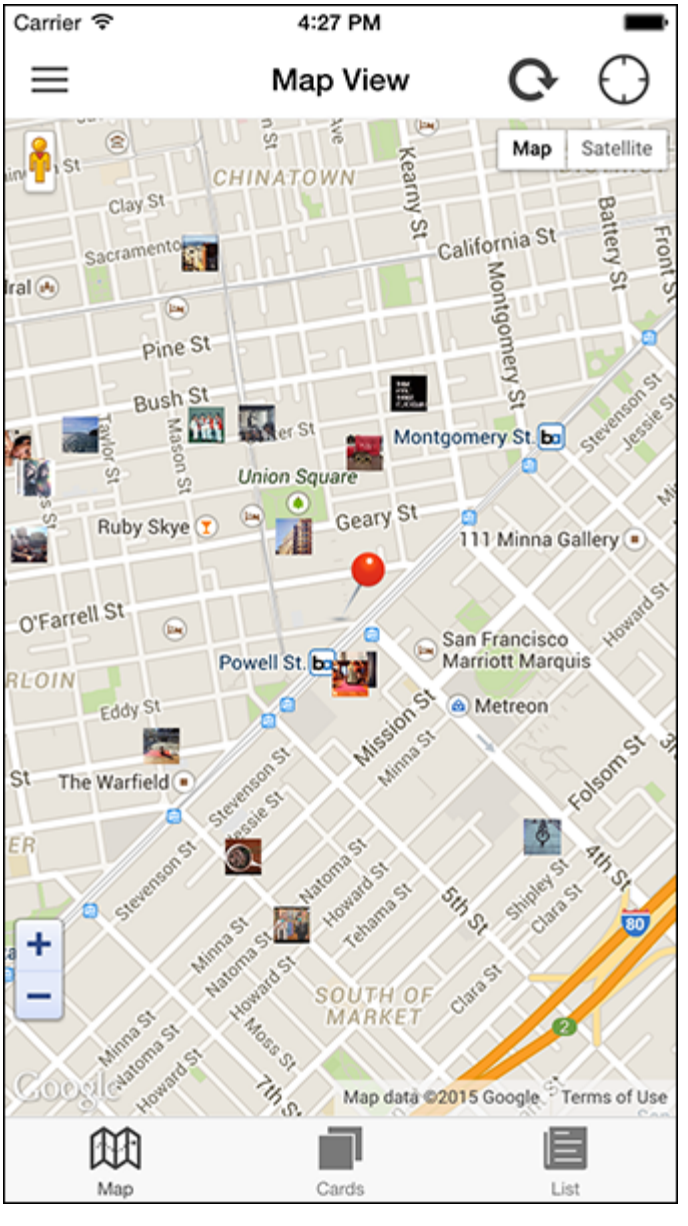
```
// GeolocationService
// /www/js/services/geolocation.js
this.addressToPosition = function (strAddress) {
    var geocodingApiUrl = 'http://maps.googleapis.com/maps/api/
geocode/json?address=' + strAddress + '&sensor=false';
```

```
        var convertResultToLatLng = function (result) {
            var location = result.data.results[0].geometry.location;

            // Transforming the 'location.lat' and 'location.lng'
            // object to 'location.latitude' to be compatible with
            // other location responses like in getCurrentPosition
            return {
                latitude: location.lat,
                longitude: location.lng
            }
        };

        return $http.get(geocodingApiUrl)
            .then(
                convertResultToLatLng,
                function (reason) {
                    return $q.reject(reason);
                }
            );
    };
```

Using Angular's `$http` service, we literally *get* the geolocation object from the Google Maps Geocode API. After retrieving a successful dataset, the specific latitude and longitude values are filtered out of the results and returned as an object for later use.

Setting up the map view inside a tab

Once we have the `GeolocationService` up and running we are ready to move on to the big guy, implementing `angular-google-maps` in the tab's view. The resulting view is shown below, and we will guide you step-by-step through its creation. Here is the map view as seen from a browser:



And, the HTML used to generate the map view:

```
// /www/templates/tap-map.html
<div data-tap-disabled="true" id="googleMap">
  <img class="map-pin-center" src="img/map-pin.png">
  <ui-gmap-google-map
```

```
        control="map.control"
        center="map.center"
        zoom="map.zoom">

        <ui-gmap-markers
            models="map.markers"
            coords="'coords'"
            icon="'icon'"
            click="'showPost'">
        </ui-gmap-markers>
    </ui-gmap-google-map>
</div>
```

We have omitted the wrapping `ion-content` directive in this example.

Touchstart, touchend, and click events on touch-enabled devices

Adam Bradley, one of the core developers of Ionic, wrote the following about the infamous 300ms click delay, also known as 'ghost click' or 'fastclick', which browsers implement for touch-enabled devices:

On touch devices such as a phone or tablet, browsers implement a 300ms delay between the time the user stops touching the display and the moment the browser executes the click. It was initially introduced so the browser can tell if the user wants to double-tap to zoom in on the web page. Basically, the browser waits roughly 300ms to see if the user is double-tapping, or just tapping on the display once.

Out of the box, Ionic automatically removes the 300ms delay in order to make Ionic applications feel more like native applications. Other solutions, such as fastclick and Angular's ngTouch should not be included, to avoid conflicts.

But Ionic is not the only one with a workaround for this behavior. As soon as you start developing a library that is mobile friendly, you will need to address this issue sooner or later when you start tweaking the performance.

Now we have two libraries handling the click events inside of our `<ion-content>` element. This can lead to unexpected or unwanted behavior like markers that can't be clicked or that trigger an unwanted double click. Ionic has a built in workaround for situations like these, where you can't influence the behavior of both libraries managing the click events. In our map view, we have to add the `data-tap-disabled` directive to disable click event management by Ionic and allow `angular-google-maps`, and thus Google Maps to do its own event management.

ui-gmap-google-map

Next up is the `ui-gmap-google-map` directive. This is the wrapping element that actually initiates Google Maps. For Trendicity, we have used three configuration attributes of this directive: control, center, and zoom.

```
// /www/templates/tap-map.html
<ui-gmap-google-map
  control="map.control"
  center="map.center"
  zoom="map.zoom">

  <!-- contents -->
</ui-gmap-google-map>
```

`control` is an empty object that will be extended with functionality by `angular-google-maps` once the map has been initialized. The two functions added are `getGMap()`, which returns the direct reference of the Google map instance being used by the directive, and `refresh(optionalCoords({latitude:, longitude})`, which refreshes the current map on its current coords if no coords are specified.

`center` is an object or array containing a latitude and longitude to center the map on. By default we will try to center the map on the user's current location using the `GeolocationService.getCurrentPosition()` method.

`zoom` is an expression to evaluate as the map's zoom level.

ui-gmap-markers

With the main `ui-gmap-google-map` directive properly configured, the next order of business is to add markers to the map. Thankfully, `angular-google-maps` contains another helpful directive that allows you to efficiently manage multiple map markers via AngularJS -`ui-gmap-markers`. This directive is not to be confused with the `ui-gmap-marker` directive (singular marker).

Since Trendicity's map view will be displaying several markers, using the plural `markers` directive is more efficient because it has been designed to overcome the high overhead of using `ng-repeat` with the singular `marker` directive. The four configuration options for this directive - models, coords, icon, and click - are shown below.

```
// /www/templates/tap-map.html
<ui-gmap-markers
    models="map.markers"
    coords="'coords'"
    icon="'icon'"
    click="'showPost'">
</ui-gmap-markers>
```

`models` is an array of objects defining each marker to add to the map. It is worth mentioning that each model object in the array must contain an identifier property.

`coords` is the name (string) of the property of the model in the models array containing the marker coordinates, and must refer to an object containing `latitude` and `longitude` properties, or a **GeoJSON Point** object. The special value `'self'` can be used to tell the directive that the objects in the array directly contain the marker coordinate object values.

`icon` is the name (string) of the property of the model in the models array containing the URL to the icon image to use for each marker.

`click` can be a string or expression defining the event handler to be executed when clicking a marker. In this case, a string is used to define the name of the handler function on the marker model.

Overriding the Nav Bar

Before continuing onto the Map View Controller, it is worth mentioning that the map view's template file also contains an `ion-nav-bar` that overrides the one defined for all tabs inside the `www/templates/home.html` file. The reason for this is that allowing users to search for posts that are trending or from their personal feed doesn't make as much sense on the map view since discovery is directly tied to the location pin.

Instead, two new buttons are used on the right side of the nav bar - one for refreshing the view after the pin's location has changed and another for repositioning the pin on the user's current location. To accomplish this task, a new `ion-nav-bar` directive (shown below) is placed inside the `ion-view` and outside the `ion-content`.

```
// /www/templates/tap-map.html
<ion-nav-bar>
    <ion-nav-buttons side="left">
        <button menu-toggle="left" class="button button-icon
icon ion-navicon"></button>
    </ion-nav-buttons>
    <ion-nav-buttons side="right">
        <button class="button button-icon icon ion-refresh" ng-
click="refresh()"></button>
        <button class="button button-icon icon ion-pinpoint" ng-
click="locate()"></button>
    </ion-nav-buttons>
</ion-nav-bar>
```

Map View Controller

That wraps up the HTML for the map view inside of the map tab. We have seen how to force Ionic not to handle taps and clicks inside of our map by using the `data-tap-`

`disabled` directive, covered the setup of the two directives provided by the `angular-google-maps` library, and discussed how to override the map view's default nav bar.

To explore the implementation of the `MapViewCtrl` and learn how easy it can be to manage Google Map components from AngularJS, check out the complete book.