Hybrid Geo-App Exercise

This exercise will cover all the “do” requirements for period-6. It will implement a small mobile app which communicates with a Node/Express/Mongo-backend and will integrate with the “hardware” on the device.

Some of you have complained over the requirement that it “has to be deployed on a real device” when you don’t have an Android device. If this is the case, and you can live without trying this for real;-) you should be able to test this via a browser (since it can request permission for your location) via the `ionic serve` command.

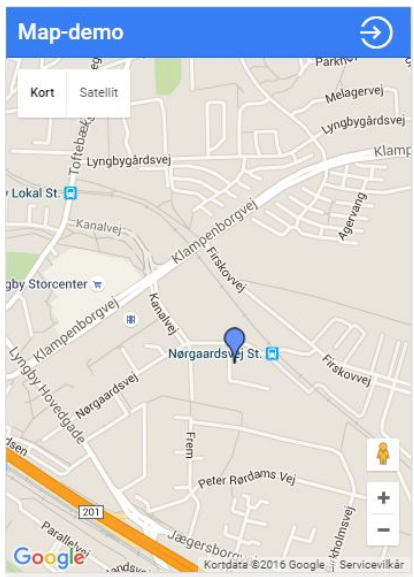
Technologies involved in this exercise

- Ionic (=Angular) + Cordova
- Google Maps API
- Express, MongoDB, Mongoose and MongoDB’s ability to handle geolocation data

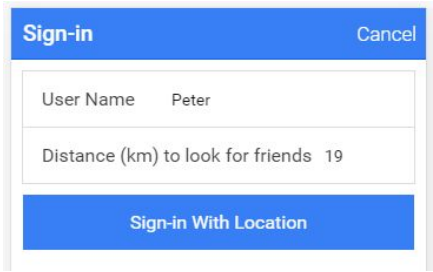
Description

The task is to create a very simple proof of concept solution for a friend-finder App, with the following (minimum) requirements.

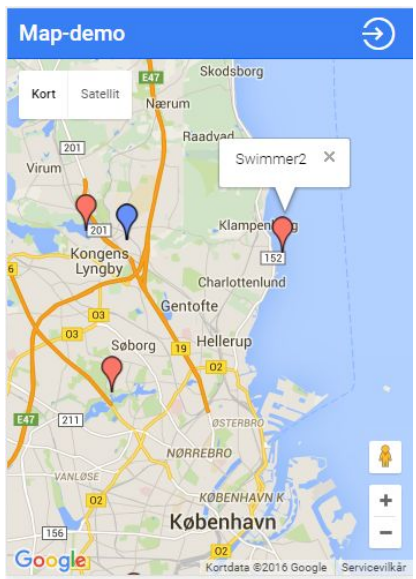
When the App starts, it should show a map with the users positon “marked” as sketched in this figure.



When the "login" button is pressed, a simple Form should allow the user to "login" via a User Name and a number representing the radius (in kilometres) for which to search for friends as sketched below.



After signing in, the map must be updated with a marker for each friend found inside the given range[1].



You could add a feature, so if a marker is pressed, the name of the friend will be shown as sketched on the figure.

[1] See hints for backend for additional details

Backend

Note: *This backend description comes before the Frontend (app) description. This is not the order in which you should implement this exercise, but you need to understand the backend API, in order to implement the frontend (APP)*

Since this exercise is meant only to implement a simple prototype, we will use the following assumptions:

- All users of this system are friends, that is, when you “log-in” you are added to the database and will be “visible” to all other users.
- A user only “exists” for 30 minutes to prevent invalid location data. Next time you “log-in” you will be added again. We assume that all users "in some way" have a unique username.

The backend should provide only a single POST route as sketched below:

This will “log-in” (again, we assume users have unique names), with your current location and return an array with all friends (users) found inside the radius given in the distance parameter (in km).

POST: `api/friends/register/:distance`

JSON-Format for JSON provided with the request	JSON-Example

`{`
`  "userName":"USERBAME",`
`  "loc": {`
`  "type": "Point",`
`  "coordinates": [longitude,latitude]`
`  }`
`}`
	`{`
	`  "userName": "Ib",`
	`  "loc": {`
	`  "type": "Point",`
	`  "coordinates": [12.487442,55.773718 ]`
	`  }`
	`}`

JSON-Format for JSON provided with the response ("friends" found inside the given radius)	Example

`[`
`{"userName":"USER1",`
`  "loc":{"type":"Point",`
`  "coordinates":[ longitude, latitude]}`
`},`
`{"userName":" USER2",`
`"loc":{"type":"Point",`
`"coordinates":[longitude, latitude]}`
`},...`
`}`
`]`
	`[`
	`{"userName":"Runner11",`
	`  "loc":{"type":"Point",`
	`  "coordinates":[12.515734,55.646729]}`
	`},`
	`{"userName":"Runner1",`
	`"loc":{"type":"Point",`
	`"coordinates":[12.502635,55.719345]}`
	`...`
	`]`

The format used for location above is called GeoJSON and we are using the simplest type; a point.

When providing latitude and longitude parameters for GeoJson it must be given as *longitude*, *latitude* which is the opposite of how it's used in navigation and in Googles MAP API.

Note in the following how Google uses latitude and longitude as sketched below:

```
var latLng = new google.maps.LatLng(position.coords.latitude, position.coords.longitude);
```

Make sure to remember this, so you don't spend a lot of silly time in debugging related to this aspect

Getting Started

You can use this backend as a starting point (Remember, it's a POST-route, so you can't just click):

`http://ionicboth-plaul.rhcloud.com/api/friends/register/34`

It has a few test users (inside a radius of 30 km from Lyngby) that don't disappear after 30 min. So use this URL for a start.

You can test this route using postman and a test user as sketched below (Position is almost here ;-):

```
{"userName":"Donald Duck","loc": { "type": "Point","coordinates": [12.511813,55.770319]}}
```

Remember to set **Content-type** to "**Application/json**"

Use this URL, and continue to the section "Implementing the Hybrid App".

A later section will guide you through the steps of making your own geo-backend ;-)

Implementing the Hybrid-app

As stated in the introduction to this period we will only scratch the surface, so we will learn most of the stuff from existing tutorials and twist them to our needs. The main purpose with this period is; to get more experience with Express, Mongo, and Angular (via Ionic) and obviously also introduce the new aspect of MEAN - Mobile-MEAN ;-)

Use links on this slide, for information about *ngCordova*: http://js2016.azurewebsites.net/hybrid1/hybrid1.html#19

Everything we do in this part is Angular (with lots of ionic directives) + JavaScript, so you should be on solid ground.

Step-1)

Implement a basic application that uses the Google map-API for the map, and the ngCordovas plugin: `cordova-plugin-geolocation.`

There are several tutorials with getting started examples out there. I suggest you follow the one below, mainly because it is simple and easy to twist to our needs. This example mixes AngularJS and plain JavaScript which normally is not recommended but we will live with that for this exercise.

http://www.joshmorony.com/integrating-google-maps-with-an-ionic-application/

Complete the tutorial, test in a local browser (ionic serve) and on a real device (ionic run android, when you phone is connected via a cable)

Step-2)

Let's make the example a bit more interesting and add a marker to indicate our position. Add the code below, right under the statement: `$scope.map = map;`

```
var marker = new google.maps.Marker({
  position: myLatlng,
  map: $scope.map,
 //icon: new google.maps.MarkerImage("http://maps.google.com/mapfiles/ms/icons/" + "red.png"),
  title: "Me"
});
var infowindow = new google.maps.InfoWindow({
  content: "Me"
});
marker.addListener('click', function() {
  infowindow.open($scope.map, marker);
});
```

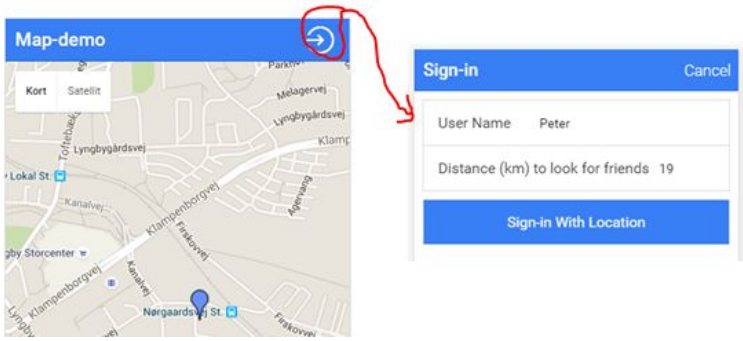
Comment out the line with the icon to see how you can get different coloured markers if you want that for your "friends".

Step-3)

When this is done and tested, both in a browser (ionic serve), and on a real device we will start to make this a bit more interesting/realistic.

Remember, the task is to provide a simple friend-finder, so we need a dialog to allow users to enter their username (we assume everyone has a unique name) + a distance (in kilometres) to indicate the radius for the circle inside which we want all our friends. This will be sent to the server, including the current position (latitude and longitude)

The simplest way to do this is to use a Modal window as sketches in this figure.



Replace the ion-header-bar with the code below to get the log-in button

a)

```
<ion-header-bar class="bar-positive">
  <h1 class="title">Map-demo</h1>
  <div class="buttons">
    <button class="button button-icon ion-log-in" ng-click="openModal()">
    </button>
  </div>
</ion-header-bar>
```

b) See this link for a live ionicModal example: http://ionicframework.com/docs/api/service/$ionicModal/

Feel free to use the code below for the Modal (place it right under the closing ion-pane tag):

```
<script id="my-modal.html" type="text/ng-template">
  <ion-modal-view >
    <ion-header-bar class="bar bar-header bar-positive">
      <h1 class="title">Sign-in</h1>
      <button class="button button-clear button-primary" ng-click="modal.hide()">Cancel</button>
    </ion-header-bar>
    <ion-content class="padding">
      <div class="list">
        <label class="item item-input">
          <span class="input-label">User Name</span>
          <input ng-model="user.userName" type="text">
        </label>
        <label class="item item-input">
          <span class="input-label">Distance (km) to find friends</span>
          <input ng-model="user.distance" type="number">
        </label>
        <button class="button button-full button-positive" ng-click="registerUser(user)">Sign-in With
Location</button>
      </div>
    </ion-content>
  </ion-modal-view>
```

c)

Add an ng-controller statement to the body tag to let this part know the *MapCtrl* also. Since Controllers are not singletons, this will actually give two instances of the controller, one for the modal, and one for the directive:

```
<ion-nav-view></ion-nav-view>
```

To avoid problems related to this, and since the tutorial do not suggest additional pages, I suggest that you replace the directive above (which corresponds to the `ng-view` directive you already know) with this:

```
<ion-content >
  <div id="map" data-tap-disabled="true"></div>
</ion-content>
```

Now you can also remove the app.config(...) section in the code since it's not used.

An alternative to the steps suggested in section c) above is to factor all state from the controller out into one or more services. If you do this you could extend your app with either tabs or a sidemenu as explained on the getting-started page http://ionicframework.com/getting-started/
The data you get back from the backend is sorted by distances (closest first), so you could add a separate log-in view, a view (the one you will implement in step four) with all your nearest friends on a map, and a view with all your friends listed by name (sorted by how close they are).

d) Add the necessary code to open/close the modal, inspired by the $ionicModal live example given above.

Step-4)

Add the `registerUser()` method, which should post user name, distance and geolocation to the backend, and receive the list of friends found inside the radius given via the distance parameter.

You can use the method below as a starting point. It builds (assuming you have used the Modal given above) the necessary object which you should pass to the data property of the $http-post request you are supposed to complete.

```
$scope.user = {};
$scope.registerUser = function (user) {
  $scope.modal.hide();
  console.log("1: " + user.userName);
  console.log("2: " + user.distance);
  user.loc = {type: "Point", coordinates: []}; //GeoJSON point
  user.loc.coordinates.push(myLatlng.lng()); //Observe that longitude comes first
  user.loc.coordinates.push(myLatlng.lat()); //in GEoJSON
  console.log(JSON.stringify(user));
     $http({
      method: "POST",
      url: " http://ionicboth-plaul.rhcloud.com/api/friends/register/"+user.distance,
      data: user
  }).then(...) //Your task is to handle the response
}
```

Step-5)

Complete the method started in step-4. You should use the returned array (unless empty, if no friends where found) to update the Map, with markers for each friend found.

Implementing your own backend

If you cloned the backend for the project/task-app introduced last week I recommend you continue with this so you can reuse your OpenShift gear. If not, clone it (https://github.com/Lars-m/ionicClassDemoBackend.git), deploy to OpenShift, and continue with below:

Since the backend only expose a single POST-route, all we need is that route and the mongoose scheme required to hold users and their locations.

Step-1)

This exercise will introduce two extremely cool features in Mongo.

- A TTL-index (time to live) that sets a TTL on the document (https://docs.mongodb.com/manual/core/index-ttl/). This is how we will ensure that users are deleted after 30 min.
- A 2dsphere index which supports queries that calculate geometries on an earth-like sphere (https://docs.mongodb.com/manual/core/2dsphere/). This is how we will query for nearby "friends"

Open the file database.js in the config folder and localize these two statements:

```
connection = db;

done();
```

In between those two lines, add the code to create the two indexes:

```
connection.collection("friends").createIndex({"created": 1},{expireAfterSeconds:30*60 } );
connection.collection("friends").createIndex({loc:"2dsphere"});
```

Restart the project and use the Mongo Client (or RoboMongo if installed) to verify that the indexes have been created:

```
use ionicbackend
db.friends.getIndexes()
```

Step-2

In this step you should create the route for the post request explained in the section *backend*. Add a JavaScript file `friends.js` in the `routes` folder and set up `app.js` to use this file.

The purpose of the single API-route is to:

- Check if a user exist, and if, update the user with the new location and date (this prolongs the TTL)
- If the user did not exist (remember, in all cases they will disappear after 30 min.) create the user
- Finally the distance parameter is used to query for all friends found inside that radius and the result is returned as a JSON array.

The first two steps should be pretty straight-forward, given our current knowledge of Mongo. For the last part however, we are going to use the 2dsphere index created above to make a geo-query.

Read this article for a quick introduction to the Geospatial features in MongoDB;
http://tugdualgrall.blogspot.dk/2014/08/introduction-to-mongodb-geospatial.html

Feel free to use or get inspired by the example provide below

```
router.post("/register/:distance", function (req, res) {
  var db = connection.get();
  var user = req.body;
  var distance = req.params.distance*1000;//in km
  delete user.distance;
  user.created = new Date();  //This is the property with the TTL index
  db.collection("friends").findOneAndReplace(
    {userName: user.userName}, user, function (err, result) {
    if (err) {
      res.statusCode = 500;
      return res.json({code: 500, msg: err.message});
    }
    if (result.value == null) {  //User was not found
      db.collection("friends").insertOne(user, function (err, result) {
        if (err) {
          res.statusCode = 500;
          return res.json({code: 500, msg: err.message});
        }
        return findNearestAndMakeResponse(user, distance, res)
      });
    }
    else {
      return findNearestAndMakeResponse(user, distance, res);
    }
  });
});


function findNearestAndMakeResponse(user, distance, res) {
  var db = connection.get();
  db.collection("friends").find({
      userName: {$ne: user.userName},
      loc: {$near: user.loc,$maxDistance: distance}
    },{_id: 0, created: 0}).toArray(function (err, docs) {
    if (err) {
      res.statusCode = 500;
      return res.json({code: 500, msg: err.message});
    }
    return res.json(docs);
  });
}
```

The first part of this example includes "nothing new". But you should notice the `findNearestAndMakeResponse(..)` method which does most of the "magic".

The `find` method uses MongoDB's Geospatial Query Operators; `$near` and `$maxDistance`, to query for all users (expect the user himself,because of the $ne operator) inside the given radius, with the requesting users location as centre.

Step-3)

Add the necessary code to fix same-origin issues (see project.js), before you upload to Openshift.

Change the URL in you app to use your own backend and test via a phone.