NYS FAIR EVENTS MOBILE APPLICATION WITH CLIENT-SIDE CACHING

A Master's Project

Presented to

Department of Computer and Information Sciences

SUNY Polytechnic
Institute

Utica, New
York

In Partial Fulfilment of the requirements for the Master of Science Degree

By

Sumant Kanala

(U00287895)

December 2017

© SUMANT KANALA 2017

NYS Fair Events Mobile application with client-side caching

Master of Science project in Computer and Information Sciences
Department of Computer Sciences
SUNY Polytechnic Institute

Approved and recommended for acceptance as a project in partial fulfillment of the requirements for the degree of **Master of Science in Computer and Information Sciences**

____

Date

____

Chen-Fu Chiang, Ph. D. (Adviser)

____

Iulian Gherasoiu, Ph. D.

____

Ali Tekeoglu, Ph. D.

NYS Fair Events Mobile application with client-side caching

Declaration

I declare that this project is my own work and has not been submitted in any form for another degree or diploma at any university or other institute of tertiary education. Information derived from the published and unpublished work of others has been acknowledged in the text and a list of references is given.

Sumant Kanala

Abstract

NYS Fair Events collects data about fair events which happen in New York state throughout the year, bundles them, displays the upcoming events and useful information about the event itself, the weather and forecast prediction, and a Google Maps to show the route to the event from the user's location.

The motivation for creating this project arose with understanding the growing market for mobile applications and by working for a startup for several months now in the field of web development. A trend has been established in which more users are switching towards mobile apps as their preferred information exchange tool than their traditional PCs and hence the development of better apps should be geared towards mobile phones and tablet PCs.

The development of the app is mainly divided into two steps, the client and server side. For the client side I developed a Cordova-based mobile app which is cross-platform and can be compiled to work on Android and IOS based mobile devices. For the server side, I used Node.js runtime environment and deployed it onto Heroku's free dyno tier which is a cloud-based Platform as a service (paaS). Based on user's actions, data is requested from the server's endpoints and appropriate information is served and shown to the user in an intuitive manner.

Contents

Abstract 4

Chapter 1: Introduction 7

1.1 Cordova 8

1.2 Node.js. 10

Chapter 2: Communication between client and server 11

2.1 Client-Server Model 11

2.2 Client and Server role 11

2.3 Client and Server communication 12

Chapter 3: Requirements 13

3.1 Events Data 13

3.2 Tools and Libraries 13

3.2.1 Backend Libraries 13

3.2.2 Frontend Tools/Libraries/Framework 14

Chapter 4: Functionality 15

4.1 Events List Page 15

4.1.1 Server Side 16

4.1.2 Client Side 19

4.2 Events Information Page 25

4.2.1 Client Side 27

4.2.2 Server Side 28

4.2.3 Back to Client Side 29

4.3 Improvements and Usability features 32

4.3.1 Server-side scalability 32

4.3.2 Client-side enhancements 33

References 34

List of Figures

Figure 1: Apache Cordova Logo 7

Figure 2: Node.js logo 8

Figure 3: Client-Server Model 10

Figure 4: Landing page on mobile showing events 15

Figure 5: Library Imports for server code 16

Figure 6: Body parser middleware 16

Figure 7: Access Control Headers 17

Figure 8: Weather API Initialization 17

Figure 9: Weather API Initialization with API key used from environment variables 18

Figure 10: GET API endpoint for the list events 18

Figure 11: Bootstrapping Angular into HTML 19

Figure 12: Route Provider for rendering dynamic views..19

Figure 13: Container div with main heading tag on main.html................................20

Figure 14: Spinning Loader div layout 21

Figure 15: Spinning Loader in action 22

Figure 16: Controller code in html to display list events 22

Figure 17: MainCTRL controller 23

Figure 18: Function which takes user to next view..24

Figure 19: (i) Events Information view/page on an Android mobile 25

(ii): Daily data weather information 26

(iii): Extended weather information 26

Figure 20: WeatherAndMaps.html 27

Figure 21: EventCTRL controller 27

Figure 22: POST API endpoint for event information 28

Figure 23: Success callback of the POST request 29

Figure 24: WeatherAndMaps.html file: (i) back button and temperature toggle 30

(ii): Current temperature 30

(iii): Hourly weather row data 30

(iv): Daily weather in three columns data 31

(v): Additional weather information 32

(vi): Link to go to Google Maps …..32
(vii): div which loads Google Maps …...32
Figure 25: Master forking worker processes …..32
Figure 26: Toggle Switch State: (i) Toggle switch off …………………………........33
(ii): Toggle Switch animation happening when user clicks …………..........33
(iii): Toggle switch on ……………………...33

Chapter 1: Introduction

1.1 Cordova

Apache Cordova (formerly **PhoneGap**) is an open source, hybrid mobile application development framework originally created by Nitobi. Adobe Systems purchased Nitobi in 2011, rebranded it as PhoneGap, and later released an open source version of the software called Apache Cordova.

Apache Cordova enables software programmers to build applications for mobile devices using CSS3, HTML5, and JavaScript instead of relying on platform-specific APIs like those in Android, iOS, or Windows Phone.

The resulting applications are **hybrid**, meaning that they are neither truly native mobile application (because all layout rendering is done via **Web views** instead of the platform's native UI framework) nor purely Web-based (because they are not just Web apps, but are packaged as apps for distribution and have access to native device APIs).

The core of Apache Cordova applications use CSS3 and HTML5 for rendering and JavaScript for logic. HTML5 provides access to underlying hardware such as the accelerometer, camera, and GPS. However, browsers' support for HTML5-based device access is not consistent across mobile browsers, particularly older versions of Android. To overcome these limitations, Apache Cordova embeds the HTML5 code inside a native WebView on the device, using a foreign function interface to access the native resources of it.

Apache Cordova can be extended with native plug-ins, allowing developers to add more functionalities that can be called from JavaScript, making it communicate directly between the native layer and the HTML5 page. These plugins allow access to the device's accelerometer, camera, compass, file system, microphone, and more.

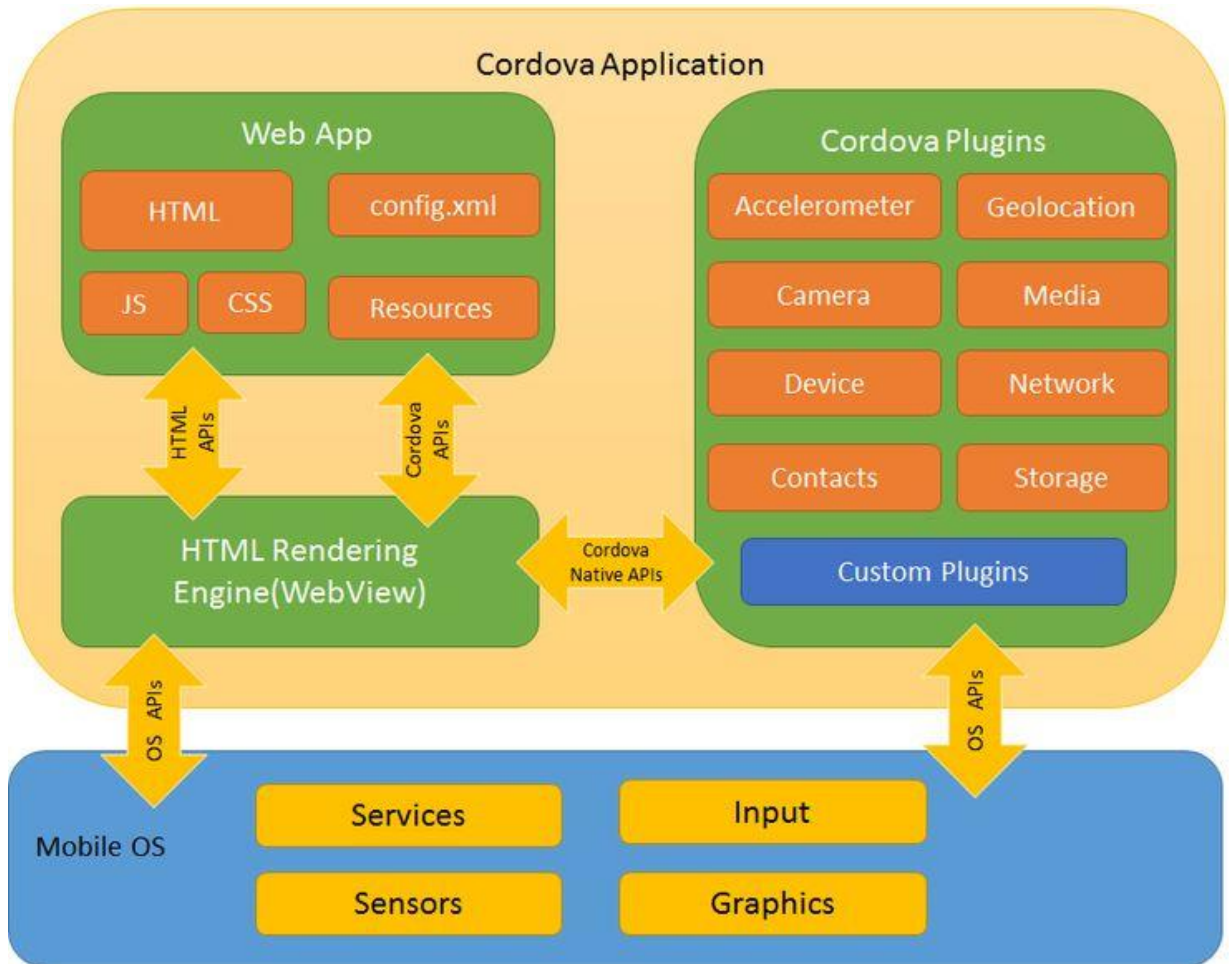


Figure 1: Inner workings of Apache Cordova [2]

1.2 Node.js

Node.js is cross-platform runtime used to write server-side code in JavaScript. JavaScript was used for client-side programming, in which scripts were written and embedded into the webpage's HTML, which is parsed and executed by the JavaScript engine in the client's web browser. Node.js enables JavaScript to be used for server-side scripting, and runs scripts server-side to produce dynamic web page content before the page is sent to the user's web browser. Consequently, Node.js has become one of the foundational elements of the "JavaScript everywhere" paradigm, allowing web application development to unify around a single programming language, rather than rely on a different language for writing server side scripts which makes it slightly easier and faster to develop applications.

Though ".js" is the conventional filename extension for JavaScript code, the name "Node.js" does not refer to a particular file in this context and is merely the name of the product. Node.js has an event-driven architecture capable of asynchronous I/O. These design choices aim to optimize throughput and scalability in Web applications with many input/output operations, as well as for real-time Web applications (e.g., real-time communication programs and browser games).

Corporate users of Node.js software include GoDaddy, Groupon, IBM, LinkedIn, Microsoft, Netflix, PayPal, Rakuten, SAP, Tuenti, Voxer, Walmart, Yahoo!, and Cisco Systems.

Node.js is primarily used to build network programs such as Web servers. The biggest difference between Node.js and PHP is that most functions in PHP block until completion (commands execute only after previous commands have completed), while functions in Node.js are designed to be non-blocking(commands execute concurrently or even in parallel, and use callbacks to signal completion or failure) [3].

Figure 2: Node.js Logo [4]

Chapter 2: Requirements

2.1 Events Data

Data regarding events information such as venue, description, location are obtained from the official NYS website and the weather data is obtained darksky.net (formerly **OpenWeather**). Darksky.net is an open source, cost-free weather gatherer website through which data can be obtained through REST API calls. A developer must register and sign up for an API key and using that in the appropriate programming language of their choice, so that it is used to make calls for weather data for a location.

2.2 Tools and Libraries

A myriad of **open source** tools and libraries are used in this project, thanks to the ever-growing interest among developers for encouraging open source software.

2.2.1 Backend Libraries (Node.js)

- **fs – File System [5]**: File I/O is provided by simple wrappers around standard POSIX functions. To use this module execute **require('fs')** command. All the methods have asynchronous and synchronous forms.
  The asynchronous version takes a completion callback as its last argument. The arguments passed to the completion callback depend on the method, but the first argument is always reserved for an exception. If the operation was completed successfully, then the first argument will be null or undefined.
  When using the synchronous form any exceptions are immediately thrown. Exceptions may be handled using try/catch, or they would be handled in the outermost parent function which called it.

  Example code snippet of the **asynchronous** version:
  ```
  const fs = require('fs');

  fs.unlink('/user/tmp.txt', (err) => {
    if (err) throw err;
    console.log('successfully deleted /user/tmp.txt!!');
  });
  ```

  Example code snippet of the **synchronous** version:
  ```
  const fs = require('fs');

  fs.unlinkSync('/user/tmp.txt);
  console.log('successfully deleted /user/tmp.txt!!');
  ```

- **body-parser [6]**: Parse incoming HTTP request bodies in a middleware before your handlers, available under the '**req.body**' property. Parsers can be of following forms:

i) JSON body parser
ii) Raw body parser
iii) Text body parser
iv) URL-encoded form body parser

- **Express [7]**: A simple framework on top of node to organize your web application into and MVC architecture on the server side. Different templating engines can be used for the view such as Pug, EJS, Jade, etc.. Many kinds of database wrappers are available for both relational and unstructured database systems such as MYSQL, Mongoose for MongoDB, Redis, DynamoDB, etc..

- **Request [8]**: A library for making simple HTTP calls and which supports https by default. We use this to make request to https://nysfair.ny.gov and its corresponding subdomains from which we get our events data.

- **Cheerio [9]:** This is a crucial library which is used to parse HTML from the context of server. It uses the core concepts of **jQuery**, which is a popular frontend library.

- **Forecast [10]**: This is the library which provides us the weather information about any region in the world given the latitude and longitude. Implementation coming up below to see it in action.

3.2.1 Frontend Tools/Libraries/Framework

- **AngularJS [11]**: It is a JavaScript full stack framework which lets developers tie HTML with dynamic views and add extended HTML vocabulary to enhance the application. It's easy to use syntax and readability make it one of the most popular **Single Page Applications (SPAs)**.
  SPAs are Web apps that load a **single** HTML **page** and dynamically update that **page** as the user interacts with the app. SPAs use AJAX and HTML5 to create fluid and responsive Web apps, without constant **page** reloads. However, this means much of the work happens on the **client side**, in JavaScript.

- **Font Awesome [12]**: An awesome icon pack which contains nearly all useful utility icons.

- **Weather-icons [13]**: Another useful and **consistent** icons pack consisting of all possible weather icons with seasonal, moon phases, directional, wind directions, etc.

- **Materialize [14]**: For easy and quick layout for UI we use materialize UI which follows most of the google material UI design rules and is very consistent.

- **jQuery [15]**: One of the most useful libraries ever created for manipulating and accessing the DOM of the app's native WebView.

- **Cordova [16]**: Mobile application development framework which packages the client side code written using all the above frontend libraries and can be used to build mobile apps for various platforms such as Android, IOS, Blackberry, etc..
- **Google Maps API [17]**: An API which allows developers to display google maps in your browser or in this case our mobile app's WebView.

- **Animate-CSS [18]**: A CSS library which can animate text or an HTML element with a variety of animation actions.

- **Fonts: 'Helvetica Neue**' is the primary font color for text and '**Lily**' is the specific font for the Events **heading**. Fallbacks to **Arial** and **sans-serif**, where fallbacks here mean the if the font 'Lily' is not available, the mobile browser uses **Arial** and **sans-serif** fonts which are available by default.

Chapter 4: Functionality

4.1 Events List Page

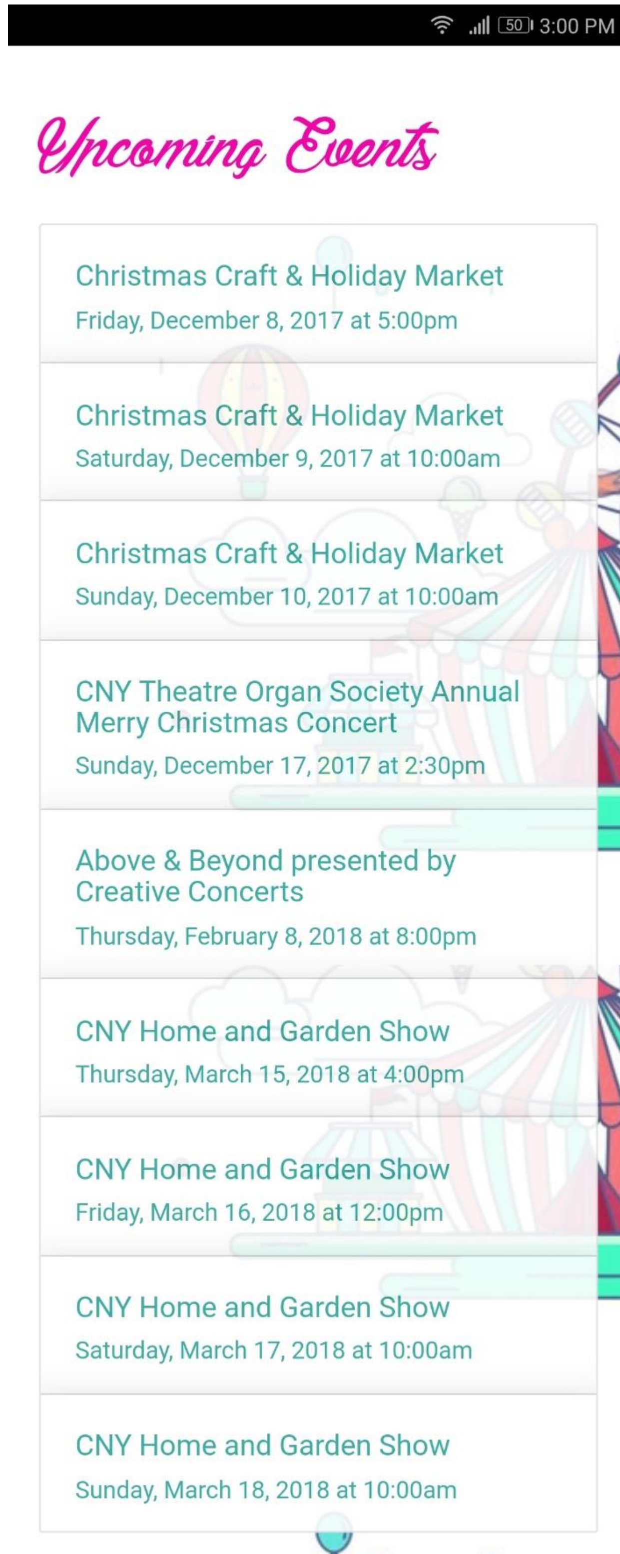


Figure 4: Landing page on mobile showing events list

The following section describes the server-side code needed for starting the application(app) and data needed for rendering the first page of the app presented in **figure 4**.

4.1.1 Server side

Since this is a relatively small mobile app the entire server code (Node.js) is served in a single file with two **REST** API endpoints.

Figure 5 describes the module imports required for our application.

```
// Default node modules

// External modules/dependencies
const express = require('express');
const request = require('request');
const bodyParser = require('body-parser');
const app = express();
const cheerio = require('cheerio');
const Forecast = require('forecast');
const PORT = process.env.PORT || 8080;
```

Figure 5: Library Imports for server code

The **'request'** module is used for acquiring/requesting data from URL and then in the callback we use the data.

Figure 6 describes the use of middleware '**body-parser**' module which is used later in the application to parse data which is sent through a HTTP **POST** or **PUT** request.

```
app.use(bodyParser.urlencoded({
  extended: true
}));

app.use(bodyParser.json());
```

Figure 6: Body parser middleware

- app.use(bodyParser.json()) module enables the web-server to accept and parse only **JSON** formatted data.
- bodyParser.urlencoded({extended: ...}) is a configuration option to enable or disable the simple algorithm for **shallow parsing** (i.e. **false**) or complex algorithm for **deep parsing** that can deal with nested objects (i.e. **true**).

```
app.use((req, res, next) => {
  res.header('Access-Control-Allow-Origin', '*');
  res.header('Access-Control-Allow-Headers', 'X-Requested-With');
  next();
});
```

Figure 7: Access Control Headers

Code presented in **figure 7** shows Headers for http responses such as allowing requests from any origin. This is generally not secure since anyone could query for our http request, it is out of the scope of this project.

```
// Initialize
var forecast = new Forecast({
  service: 'darksky',
  key: '146b6f4c49e7827fdfb373b580368cc5',
  units: 'fahrenheit',
  cache: true,      // Cache API requests
  ttl: {            // How long to cache requests. Uses syntax from moment.js
    minutes: 10,
    seconds: 00
  }
});
```

Figure 8: Weather API Initialization

In **Figure 8**, forecast API is initialized with the API key which serves weather information for any location when queried for with a valid **latitude** and **longitude** location. In production code, **API key** value is served from environment variables since you do not want to expose your token to users or anyone who has access to the code base of your application. With node.js this is generally achieved as follows.

```
// Initialize
var forecast = new Forecast({
  service: 'darksky',
  key: process.env.forecast_api_key,
  units: 'fahrenheit',
  cache: true,      // Cache API requests
  ttl: {            // How long to cache requests. Uses syntax from mom
    minutes: 10,
    seconds: 00
  }
});
```

Figure 9: Weather API Initialization with API key used from environment variables

The following code is the first endpoint used in the app. It requests data from **NYS** fair website and sends the appropriate content back to the client.

- the **request** module is used to fetch the HTML page from the URL.
- **cheerio** module is initialized and the received HTML content if fed to it. It is basically the **jQuery** equivalent for server side. It can parse DOM nodes efficiently and can help us query for all the needed elements.
- From the NYS Fair website '**. event-info'** class' **<div>** tag contains list of all events.
- Specific event list tags are queried which are then stored as **event_url**, **event_name** and **event_date** and in the form of key value pairs in an **Array** object.
- After collecting all the event's information, it is sent back to the client.

```
app.get('/api/', (req, res) => {
  var url = "https://nysfair.ny.gov/entertainment/events-calendar";
  request(url, function(error, resp, html) {
    if(error) return console.error("index.js:55 :: URL does not exist", error);
    var $ = cheerio.load(html);
    var event_list = [];
    $('.event-info').each(function(i, elem) {
      event_list.push(
        {event_url: $(this).children()['0'].children[0].attribs.href,
        event_name:$(this).children()['0'].children[0].children[0].data,
        event_date: $(this).children()['1'].children[0].data});
    });
    res.send(event_list);
  // end of request
  });
// end of endpoint
});
```

Figure 10: GET API endpoint for the list events

Section 4.1.2 describes the **client-side** code which basically receives this data, and arranges them in a clean and concise list format which is the first page of our app.

4.1.2 Client-side

The entry for the business logic is written in a single file **main.js** shown as snippets starting with **Figure 11**.

```
"use strict";
var map;
var domElement = document.querySelector("html");
var app = angular.module('app', ['ngRoute']);
```

```
document.addEventListener("deviceready", function() {
  angular.bootstrap(domElement, ["app"]);
}, false);
```

Figure 11: Bootstrapping Angular into HTML

In the first block of code the html element tag is selected and stored in a variable, so that **angular.js** code can be **bootstrapped** into HTML.

Extra code is required to bootstrap angular by wrapping the code in an event Listener. This is done because the bootstrapping needs to be executed only when all the functionality inside the **Cordova's WebView** is ready and loaded.

Hence, Cordova has this convenient wrapper which can be accessed by JavaScript's native event listener '**deviceready**'. The anonymous function after it is called a callback which means whenever the device is ready, the **callback** function is executed. This concept is called **reactive programming** in general and in JavaScript produces non-blocking code and has embraced it from a long time. A popular library called '**RxJS**' embraces this concept and built something called an **Observable**. This is beyond the scope of this project, but it is something to keep in mind.

```
app.config(function($routeProvider, $locationProvider) {
  $routeProvider
    .when("/", {
      templateUrl: "public/main.html"
    })
    .when("/weatherAndMaps", {
      templateUrl: "public/weatherAndMaps.html"
    })
    .otherwise({
      redirectTo: "/"
    });
});
```

Figure 12: Route Provider for rendering dynamic views

Figure 12 is code for setting up routing. With **Angular** we can create **single page applications (SPAs)** which is a powerful concept. Dynamic views can be created, used, and this can be utilized with routing location URL paths to different views pages.

For example, say our application is hosted on **https://www.fair-events.herokuapp.com.**

If a user visits this URL**,** then anything after this becomes a relative path, hence '/' resolves to this URL and hence we can serve up a view which in our case is 'public/main.html' which means in **public** folder of our codebase **main.html** file is served.

Figure 13 shows a snippet from **main.html** file which is loaded as a dynamic view by angular code from an **index.html** file which serves main.js file containing the bootstrapped code.

```
<div class="container" ng-controller="mainCTRL">
  <div class="row">
    <div class="col s12">
      <h3 class="animated bounceInRight" style="position:fixed; font
        -family: Lily;color:#f206a9;z-index:999;">Upcoming Events</h3>
    </div>
  </div>
```

Figure 13: Container div with main heading tag on main.html

In **Figure 13**, **container** class div tag which adds some padding to the content to be served.

h3 tag with the custom font-family **Lily** is used as a header shown as **'Upcoming Events'** in **Figure 4**.

```
    <div style="position:fixed;left:50%;top:50%;transform: translate(
      -50%);z-index: 100;" ng-if="eventList.length===0">
      <div class="preloader-wrapper active">
        <div class="spinner-layer spinner-blue">
          <div class="circle-clipper left">
            <div class="circle"></div>
          </div><div class="gap-patch">
            <div class="circle"></div>
          </div><div class="circle-clipper right">
            <div class="circle"></div>
          </div>
        </div>
```

```
<div class="spinner-layer spinner-red">
  <div class="circle-clipper left">
    <div class="circle"></div>
  </div><div class="gap-patch">
    <div class="circle"></div>
  </div><div class="circle-clipper right">
    <div class="circle"></div>
  </div>
</div>
```

```
<div class="spinner-layer spinner-yellow">
  <div class="circle-clipper left">
    <div class="circle"></div>
  </div><div class="gap-patch">
    <div class="circle"></div>
  </div><div class="circle-clipper right">
    <div class="circle"></div>
  </div>
</div>
```

```
      <div class="spinner-layer spinner-green">
        <div class="circle-clipper left">
          <div class="circle"></div>
        </div><div class="gap-patch">
          <div class="circle"></div>
        </div><div class="circle-clipper right">
          <div class="circle"></div>
        </div>
      </div>
    </div>
  </div>
```

Figure 14: Spinning Loader div layout

Code in **Figure 14** makes up an animation i.e., a **spinning loader** which is useful as it indicates to the user that the content is being loaded, is represented figuratively in **Figure 15**.

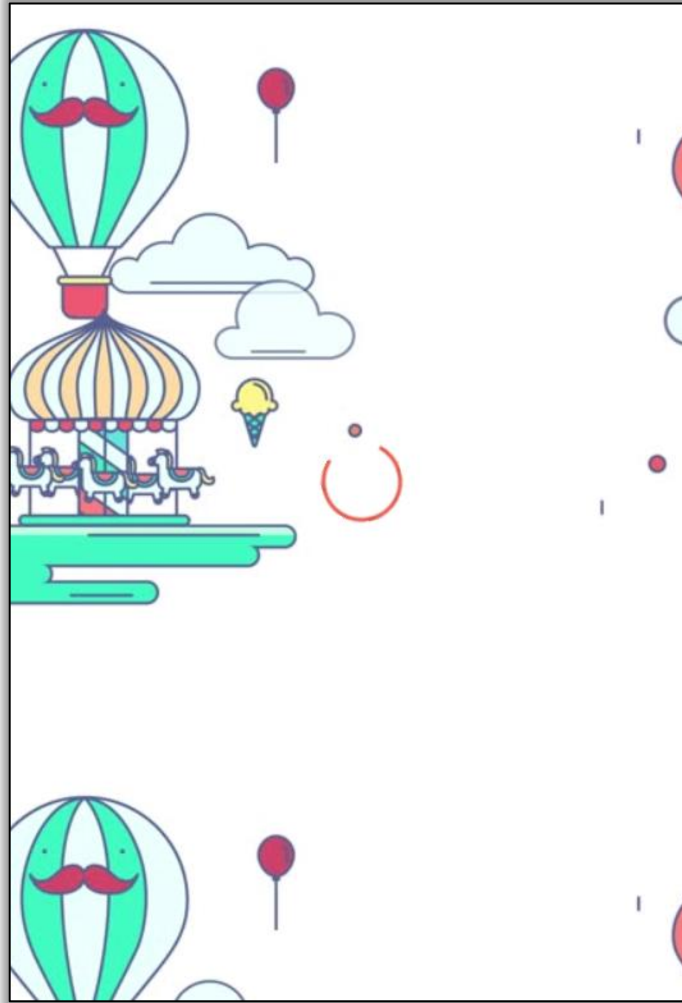


Figure 15: Spinning Loader in action

User Experience (UX) enhancements like the **spinning loader** are necessary for a mobile app since there aren't clear entry/exit points the user can navigate to, hence developers have to develop clever and intuitive ways to handle these situations.

```
        <ul class="collection" ng-if="eventList.length!==0" style="top:75
          px">
          <a ng-repeat="list in eventList" class="collection-item
          dismissable z-depth-5" ng-click="goToWeatherAndMaps(
          list.event_url)">
            <h5 style="font-size: 1.2rem">{{list.event_name}}</h5>
            <h6>{{list.event_date}}</h6>
          </a>
        </ul>
      </div>
    </div>
  </div>
```

Figure 16: controller code in html to display list events

In **Figure 13**, a custom element called **ng-controller** is defined and assigned a value. This is used for placing our angular code inside HTML with the help of something called controllers whose job is placing the **state's data** into the **view**. The controller is called **mainCTRL**.

In **Figure 16**, angular HTML element **ng-repeat** acts like a loop and all elements inside it including itself are copied over and the data from each of '**eventList**', i.e a '**list'** is placed in HTML through a concept called **'interpolation'**.

This means data from the list object can be placed in curly braces and it will be read by HTML as simple string text.

Here we can see list object contains **event_name** and **event_date** which are rendered as shown in **Figure 4** with some custom styling applied as appropriate.

Figure 17 shows how this data is set in the controller and picked up by the frontend HTML.

```
app.controller('mainCTRL', ['$scope', '$location', 'eventFactory', '
  $http', function($scope, $location, eventFactory, $http){
  $scope.eventFactory = eventFactory;

  if(eventFactory.main_list.length===0) {
    $scope.eventList = [];
    $http({
      method: 'GET',
      url: "https://fair-events.herokuapp.com/api/"
    }).then(function successCallback(response) {
        $scope.eventList=response.data;
        eventFactory.addMainList(response.data);
        $scope.$apply();
      }, function errorCallback(response) {
    });
  } else {
    $scope.eventList=eventFactory.main_list;
  }
```

Figure 17: mainCTRL controller

The **$scope** is a variable which keeps track of all other objects and functions which are initialized with the $scope keyword attached to them.

$location is used to change views/partials dynamically.

eventFactory is a factory method which is instantiated much like a controller but is used for different purposes. When the topic of factories is discussed this is explained.

$http is a service which can be used to make http requests from within the code much like an xhr request, but in a much cleaner way of doing it.

Data is requested from an endpoint which is hosted on **Heroku** at https://fair-events.herokuapp.com/ with the help of **$http** module.

In the success callback, object's data value is assigned to '**$scope.eventList'** and added to the factory's method **'addMainList'**.

An array of objects that contain key's **event_name** and **event_date** for each event. It is picked up by our controller defined in HTML as in **Figure 16.**

Now, when the user clicks on any of the links from **Figure 4**, i.e. the first page, as per the **Figure 16** and the upcoming **Figure 18,** user is navigated to **'/weatherAndMaps'** route and from **Figure 12**, routeProvider loads the '**public/weatherAndMaps.html**' view.

```
  $scope.goToWeatherAndMaps = function(url) {
    eventFactory.addUrl(url);
    $location.url('/weatherAndMaps');
  };
}]);
```

Figure 18: Function which takes user to next view

In the Figure 19 (i), the page is divided into **three components**.

The first component takes up **50vh** which is 50 percent of the **Vertical Height** of any device it takes up. This layout would be impractical on a tablet, hence it could dealt with the orientation of mobile with CSS's **media queries** which can be used to style our HTML depending on the orientation, screen height, width, etc.

It displays all the useful weather information of the location of the event we clicked on in the first page.

The second component is redundant with the third, but on mobile users won't notice the minor navigation icon inside the **third** component hence a simple link which states **'TAKE ME THERE'** makes it look clean and interactive which when clicked upon navigates the user to the location from his **current** location by opening the **google maps** application.

The third component itself is a google maps render of the location of the event for convenience.

Both the **second** and **third** components combined take up **50vh,** i.e. **bottom half** of the application's height.

4.2 Events Information Page

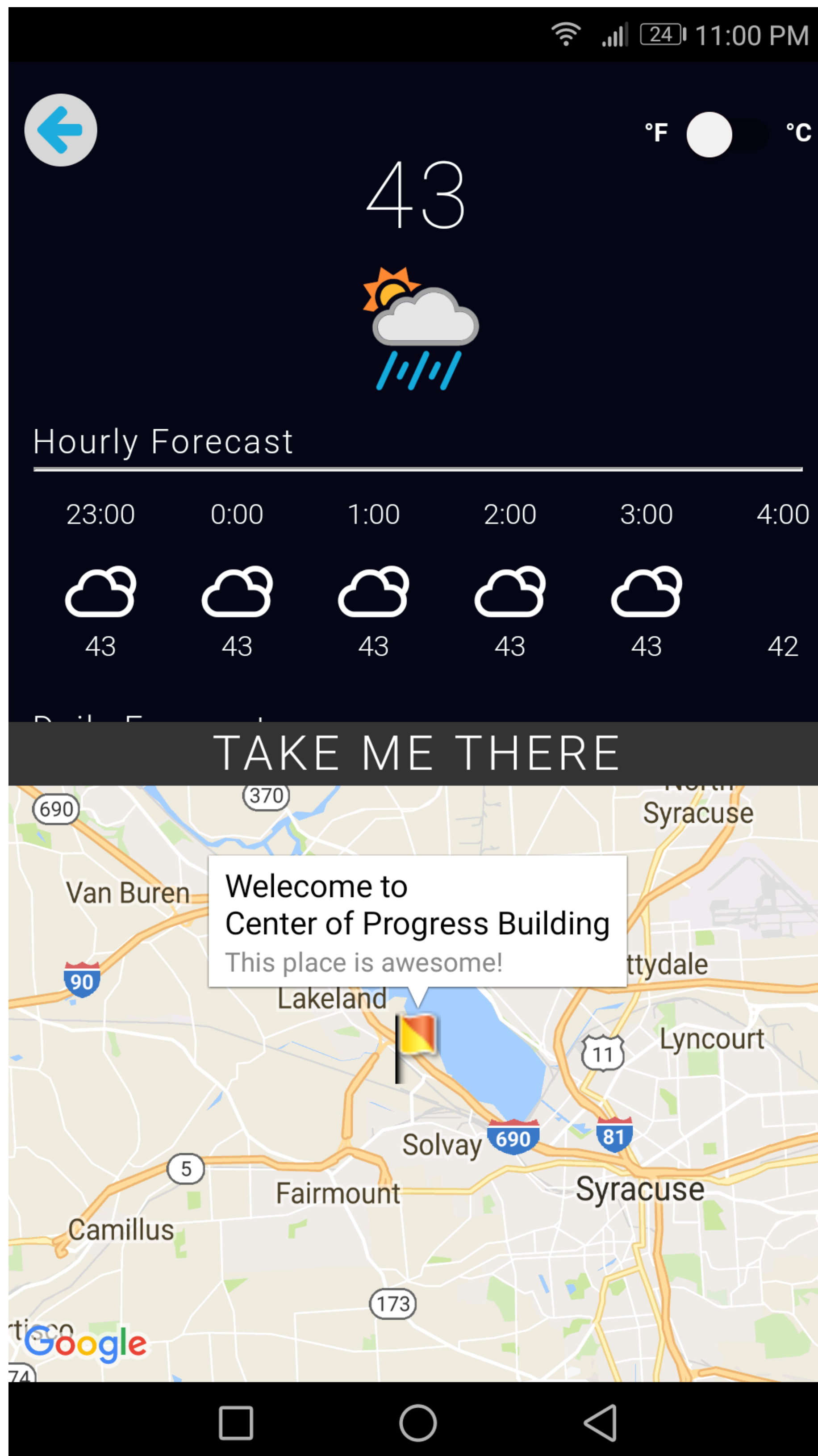
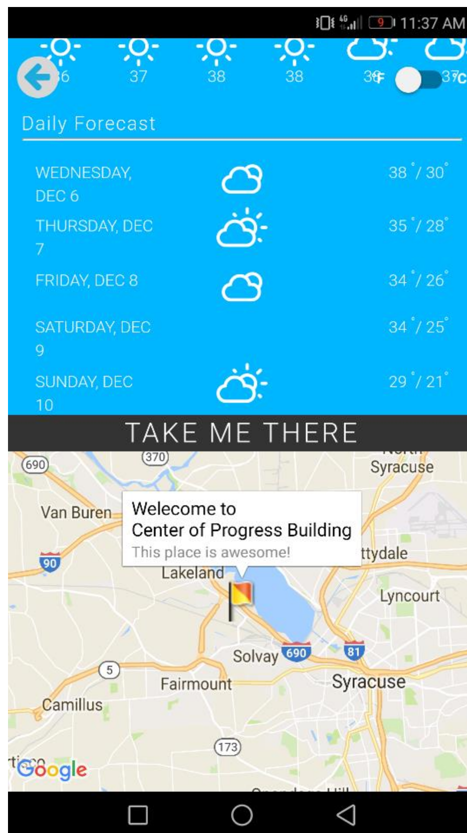
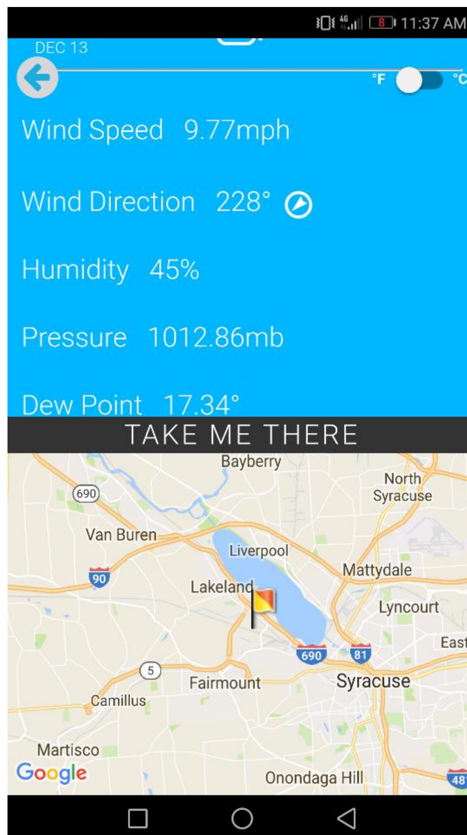


Figure 19: (i) Events Information view/page on an Android mobile



(ii) Daily data weather information



(iii) Extended weather information

4.2.1 Client side

```
<div id="custom" ng-controller="eventDataCTRL">
```

Figure 20: weatherAndMaps.html

In **Figure 20**, the controller is defined in the **div** tag**,** so that we could serve data for this view from the controller. It's code snippets are shown in Figure 21.

```
app.controller('eventDataCTRL', ['$scope', '$location', 'eventFactory', '$http',
  function($scope, $location, eventFactory, $http){
  $scope.eventFactory = eventFactory;
  $scope.spinner=true;
  function changeHandler(e){...
  }
  document.getElementById('units_checkbox').addEventListener('change', changeHandler)
    ;
  var div = document.getElementById("map_canvas");
  // Initialize the map view
  map = plugin.google.maps.Map.getMap(div);
  // Wait until the map is ready status.
  map.addEventListener(plugin.google.maps.event.MAP_READY, function(){
    $http({
      method: 'POST',
      url: "https://fair-events.herokuapp.com/api/parseLatAndLongWeather/",
      headers: {
        'Content-Type': 'application/json'
      },
      data: JSON.stringify({uri:eventFactory.event_url[0]})
    }).then(function successCallback(response) {
```

Figure 21: eventCTRL controller

In this controller, the HTML element (here a div with a class **'map_canvas'**) is loaded into a variable **'map.'**

An **eventListener** to the **maps ready** event is initialized when the map is ready and the business logic is written in its callback.

Data is requested through a **HTTP** call to the second endpoint which is served from our **Heroku** hosted backend to the relative URL **'/api/parseLatAndLongWeather'**.

Data sent in the POST request is the event's **url** as a **string,** since **HTTP** request **body** accepts only strings**.**

In the POST endpoint of our server code, the **uri** object is extracted from the request's body. The request module is used to download the HTML from the URL. The cheerio module once again is used to initialize **HTML** to make all its **DOM** nodes accessible easily like the **jQuery** library.

The **<p>** paragraph tags are extracted under the <div> tag with class **'. entry-content'**.

4.2.2 Server side

```
app.post('/api/parseLatAndLongWeather', (req, res) => {
  var url = req.body.uri;
  request(url, function(error, resp, html) {
    if(error) return console.error("index.js:55 :: URL does not exist", error);
    var $ = cheerio.load(html);

    // Info to parse 'url' data such as venue, time, fee, website, contact info
    var truth = $('.entry-content')['0'].children.filter(d => d.name==='p').map(p => p.
      children[0]);
    var len=truth.length;
    var dates = $('.eventdatelisting')['0'].children
      .filter(d => d.name==='tbody')[0].children
      .filter(tr => tr.name==='tr')
      .map(tr_ch => tr_ch.children)
      .map(td => td
        .filter(b => b.name==='td'))
      .map(ind => ind
        .map(ch => ch.children))
      .map(obj => obj
        .map(obj_ch => obj_ch
          .map(obj_ch_sub => obj_ch_sub.data)))
      .map(t => [t[0][2].trim(), t[1][1].trim()]);
```

```
    var venue=truth[len-5].next.data.trim();
    var fee=truth[len-3].next.data;
    var formatted_fee=fee.slice(2,fee.length);
    var event_website=truth[len-2].next.next.attribs.href;
    var contact=truth[len-1].next.data;
    var formatted_contact=contact.slice(2,contact.length).split('.').join('-');

    var tmp = $("#text_directions_button_contain")['0'].children[1].children[1].attribs.
      href;
    var lat_long_array = tmp.split('/')[5].split(',');
    forecast.get(lat_long_array, true, function(err, weather) {
      if(err) return console.error(err);
      res.send({
        info: weather,
        dates: dates,
        venue: venue,
        fee: formatted_fee,
        event_website: event_website,
        contact: formatted_contact,
        position: lat_long_array
      });
    });
```

Figure 22: POST API endpoint for event information

In these p tags, useful data such as **venue location**, **fee** for the event, website **URL** and **contact information** are available which are extracted and sent back as a response **JSON object** to the frontend.

4.2.3 Back to client side

The data is handled and sent to the frontend render in the HTML.

```
document.getElementById('weather').style.visibility = "visible";
eventFactory.displayWeather(response.data.info);
$scope.spinner=false;
var url=response.data.info;
$scope.current_temp=Math.round(url.currently.temperature);
$scope.humidity=100*url.currently.humidity;
$scope.windSpeed=url.currently.windSpeed;
$scope.windBearing=url.currently.windBearing;
var b=$scope.windBearing;
$('#dir').css({
   '-webkit-transform':'translateY(5px) rotate('+b+'deg)',
  '-moz-transform':'translateY(5px) rotate('+b+'deg)',
  '-ms-transform':'translateY(5px) rotate('+b+'deg)',
  '-o-transform':'translateY(5px) rotate('+b+'deg)',
  'transform':'translateY(5px) rotate('+b+'deg)',
});
```

```
    $scope.dewPoint=url.currently.dewPoint;
    $scope.visibility=url.currently.visibility;
    $scope.uvIndex=url.currently.uvIndex;
    $scope.pressure=url.currently.pressure;
    $scope.original_current_temp=$scope.current_temp;
    var hourly_tmp_arr=url.hourly.data;
    $scope.hourly=hourly_tmp_arr.map(d => ({temperature:Math.round(d.temperature)
      , icon:'wi-'+icons[d.icon.toLowerCase()], time:new Date(d.time*1000).
      getHours()+':00'}));
    $scope.original_hourly=$scope.hourly;
    var daily_tmp_arr=url.daily.data;
    $scope.daily=daily_tmp_arr.map(d => ({temperatureHigh:Math.round(d.
      temperatureHigh), temperatureLow:Math.round(d.temperatureLow), icon:'wi-'+
      icons[d.icon.toLowerCase()], date:days[new Date(d.time*1000).getDay()] + '
      , ' + months[new Date(d.time*1000).getMonth()] + ' ' + new Date(d.time*1000
      ).getDate()}));
    $scope.original_daily=$scope.daily;
    $scope.lat_long=response.data.position;
    onMapReady(response.data.position, response.data.venue);
    $scope.$apply();
  }, function errorCallback(response) {
});
```

Figure 23: success callback of the POST request

The weather data is sent to the factory service's '**displayWeather'** function which handles the first component's background color based on the '**time of the day'** and '**weather conditions'**.

The **spinner loader** is disabled and various information aspects of the event and weather information such as current temperature, humidity, wind speed, wind Direction, dew point, visibility, UV Index, pressure, hourly temperatures for the next 24 hours and daily

temperatures for the upcoming week are stored into various **$scope** variables so that it is updated on the frontend appropriately.

The location data is sent to a '**onMapReady**' function which loads the event's location information into the **third component** which is **google maps.**

```
<div id="weather" style="visibility: hidden;">
  <div id="back-circle" ng-click="goToMain()">
    <div id="back-btn">
      <i class="fa fa-arrow-left" style="font-size:24px"></i>
    </div>
  </div>
  <div class="switch">
    <label style="color: white; font-weight: bold">
      &deg;F
      <input id='units_checkbox' type="checkbox">
      <span class="lever"></span>
      &deg;C
    </label>
  </div>
```

Figure 24: weatherAndMaps.html file: (i) back button and temperature toggle

The first component consists of two div elements.

The first element is the **back button** which takes the user back to the first page, i.e. the **list of events** page.

The second element is the **toggle** switch for changing **temperature units** from **Fahrenheit to Celsius** scale and vice versa.

```
<div class="row">
  <div class="col s12">
    <p style="margin-top: 5vh; margin-bottom: 0vh;max-height: 18vh; text-align:
      center; font-size: 7vh;font-weight: lighter;">{{current_temp}}</p>
  </div>
```

(ii) Current temp

The current temperature is placed at the center of the screen from the $scope value **current_temp.**

The **Hourly Forecast** data is retrieved from the **hourly** variable which is an array, and has data for the next 24 hours from the current hour onwards. Each of them contains an object with properties **time** and **temperature**.

These are arranged in a single row which expands and can be scrolled to view all the hourly weather information. Refer **Figure 19 (i)**.

```
<div class="col s12" style="overflow-x: auto; overflow-y: hidden; white-space:
  nowrap; height: 16vh; letter-spacing: 0px">
  <span class="hourly" style="display: inline-block;
    width: 16.666vw; text-align: center; font-size: 0.9rem;" ng-repeat='hour
    in hourly'>
    <div style="max-width: 100%; height: 5vh">{{hour.time}}</div>
    <div style="max-width: 100%; height: 5vh"><i ng-class="hour.icon" class="wi
    " style="font-size: 2rem;"></i></div>
    <div style="max-width: 100%; height: 5vh">{{hour.temperature}}</div>
  </span>
</div>
```

(iii) Hourly weather row data

Similarly, the **Daily Forecast** has data for forecast for one week from the current date.

```
  Daily Forecast
</div>
<hr style="margin-left: 0.8rem; margin-right: 0.8rem"/>
<div class="col s12" style="min-height: 50vh; font-size: 0.9rem; margin-top: 1
  vh; letter-spacing: 0px">
  <div class="row" ng-repeat='day in daily' style="display: flex;
    flex-direction: row; margin-bottom: 1vh;">
    <div class="col s4" style="flex-grow: 1;  text-align: left;">{{day.date}}
    </div>
    <div class="col s4" style="flex-grow: 1; text-align: center"><i class="wi"
    ng-class="day.icon" style="font-size: 2rem;"></i>
    </div>
    <div class="col s4" style="flex-grow: 1; text-align: right">
    {{day.temperatureHigh}} <sup>&deg;</sup>/ {{day.temperatureLow}}<sup>&
    deg;</sup>
    </div>
  </div>
</div>
```

(iv) Daily weather in three columns data

Each of the **daily** array variable obtained from **$scope** is an object with properties date, icon, **temperatureHigh** and **temperatureLow**.

This data is arranged with an event's data in each row containing three columns for data from all three variables. Refer **Figure 19 (iv)**.

Additional weather information such as Wind Speed, Wind direction, Humidity, Pressure, Dew Point, UV Index, and Visibility is also provided as shown in **Figure 19 (v).**

```
<div class="col s12" style="min-height: 50vh; text-transform: capitalize; letter-spacing: 0px;">
  <p>Wind Speed  
    <span style="text-transform: none;">{{windSpeed}}mph</span>
  </p>
  <p>Wind Direction  
    <span style="text-transform: none;">{{windBearing}}&deg; 
      <i class="wi wi-wind" id="dir" style="font-size: 2rem;"></i>
    </span>
  </p>
  <p>Humidity  
    <span style="text-transform: none;">{{humidity}}%</span>
  </p>
  <p>Pressure  
    <span style="text-transform: none;">{{pressure}}mb</span>
  </p>
  <p>Dew Point  
    <span style="text-transform: none;">{{dewPoint}}&deg;</span>
  </p>
  <p>UV Index  
    <span style="text-transform: none;">{{uvIndex}}</span>
  </p>
  <p>Visibility  
    <span style="text-transform: none;">{{visibility}}mi</span>
  </p>
</div>
```

(vi) Additional weather information

Figure 19 (vi) shows the link styled like a button which takes us to that location in google maps from our current location.

```
<a id="clickable" ng-href="https://www.google.com/maps?daddr={{lat_long[
0]}},{{lat_long[1]}}">
  Take me there
</a>
```

(vi) Link to go to Google Maps

As the last step in our HTML, the map component where the google maps plugin would initialize the map data, is placed in this div.

```
  <div id="map_canvas"></div>
</div>
```

(vii) div which loads Google Maps

4.3 Improvements and Usability features

4.3.1 Server-side scalability

When building systems, the developer must always account for more traffic i.e. more users accessing the application. A simple scalable solution for the web-app is developed since the two endpoints are simple **http REST** calls.

A **master process** is set up which **forks** or **creates worker** processes depending upon the number of cores the CPU has. Essentially, this replicates the server code into **'cpuCount'** separate instances where **'cpuCount'** is the number of cores of the server. Each worker is then assigned to serve the same server code in the **app.js** file.

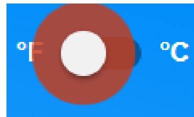
```
const cluster = require('cluster');

if (cluster.isMaster) {
  // Count the machine's CPUs
  let cpuCount = require('os').cpus().length;

  // Create a worker for each CPU
  for (let i = 0; i < cpuCount; i += 1) {
    cluster.fork();
  }

  // Listen for dying workers
  cluster.on('exit', function () {
    cluster.fork();
  });
} else {
  require('./app');
}
```

Figure 25: Master forking worker processes

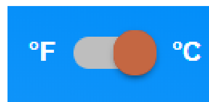
4.3.2 Client-side enhancements

- In the first page, to scroll seamlessly without using the **touchpad** of the **device**, the user can press the **volume buttons** to navigate **up** or **down** to navigate through the event list.
- In the second page i.e. the information page, when the user clicks on the toggle switch a tactile feedback sound is produced.

Figure 26 Toggle Switch State: (i) Toggle Switch off



(ii) Toggle Switch animation happening when user clicks



(iii) Toggle Switch on

References

[1] https://en.wikipedia.org/wiki/Apache_Cordova

[2] https://en.wikipedia.org/wiki/File:Open_source_Apache_Cordova_logo_image.png

[3] https://en.wikipedia.org/wiki/Node.js

[4] https://en.wikipedia.org/wiki/File:Node.js_logo.svg

[5] https://nodejs.org/api/fs.html

[6] https://github.com/expressjs/body-parser

[7] https://expressjs.com/

[8] https://github.com/request/request

[9] https://github.com/cheeriojs/cheerio

[10] https://darksky.net/dev

[11] https://angularjs.org/

[12] http://fontawesome.io/

[13] https://github.com/erikflowers/weather-icons

[14] http://materializecss.com/

[15] https://jquery.com/

[16] https://cordova.apache.org/

[17] https://developers.google.com/maps/documentation/javascript/

[18] https://daneden.github.io/animate.css/