

Homework 8: Ajax, JSON, Angular, Responsive Web Design and Node.js

Weather Search App

1. Objectives

- Achieve familiarity with the AJAX and JSON technologies.
- Create the front-end by integrating and combining HTML5, CSS, Bootstrap, and Angular.
- Create the back-end using JavaScript on Node.js.
- Both front-end and back-end must be implemented in one of the approved cloud environments.
- Build responsive web design with Bootstrap to enhance UX across multiple screens.
- Deploy your Web app on Google Cloud Platform/Amazon Web Services/Microsoft Azure.
- Leverage APIs for added functionalities. Key APIs you will work with include tomorrow.io API, Google Maps API, Google Geocoding & Places API, IPInfo API and Twitter API.

2. Background

2.1 AJAX and JSON

AJAX (Asynchronous JavaScript + XML) incorporates several technologies which include Standards-based presentation using XHTML and CSS, displaying results and interactions using the Document Object Model (DOM), data interchange and manipulation using XML and JSON, Asynchronous data retrieval using XMLHttpRequest. In summary, JavaScript binds everything together. Peruse the class slides at <https://csci571.com/slides/ajax.pdf> for detailed information.

JSON, short for JavaScript Object Notation, is a lightweight data interchange format. Its primary application for this task is in AJAX web application programming, where it serves as an alternative to the XML format for data exchange between client and server. Peruse the class slides at: <https://csci571.com/slides/JSON1.pdf> for detailed information.

2.2 Bootstrap

Bootstrap is a free collection of tools for creating responsive websites and web applications. It contains HTML and CSS-based design templates for typography, forms, buttons, navigation and other interface components, as well as optional JavaScript extensions. **Please use Bootstrap 4 or use ng-bootstrap (v 9.x.x) in this homework.** To learn more details about Bootstrap please refer to the lecture material on Responsive Web Design (RWD). Peruse the class slides at: <https://csci571.com/slides/Responsive.pdf> for detailed information.

2.3 Google Cloud Platform (GCP)

Google Cloud Platform (GCP) applications are easy to create, easy to maintain, and easy to scale as your traffic and data storage needs change. With GCP, there are no servers to maintain. You simply upload your application and it's ready to go. App Engine applications automatically scale based on incoming traffic. Load balancing, micro services, authorization, SQL and NoSQL

databases, Memcached, traffic splitting, logging, search, versioning, roll out and roll backs, and security scanning are all supported natively and are highly customizable in GCP. Peruse GCP support information for Node.js at: <https://cloud.google.com/appengine/docs/flexible/Node.js/>

2.4 Amazon Web Services (AWS)

AWS is Amazon's implementation of cloud computing. Included in AWS is Amazon Elastic Compute Cloud (EC2), which delivers scalable, pay-as-you-go compute capacity in the cloud, and AWS Elastic Beanstalk, an even easier way to quickly deploy and manage applications in the AWS cloud. You simply upload your application, and Elastic Beanstalk automatically handles the deployment details of capacity provisioning, load balancing, auto-scaling, and application health monitoring. Elastic Beanstalk is built using familiar software stacks such as the Apache HTTP Server, PHP, and Python, Passenger for Ruby, IIS for .NET, and Apache Tomcat for Java. Peruse AWS support information for Node.js at: <https://aws.amazon.com/getting-started/projects/deploy-nodejs-web-app/>

2.5 Microsoft Azure

Microsoft Azure is a cloud computing service created by Microsoft for building, testing, deploying, and managing applications and services through a global network of Microsoft-managed data centers. It provides software as a service (SaaS), platform as a service (PaaS) and infrastructure as a service (IaaS) and supports many different programming languages, tools and frameworks, including both Microsoft-specific and third-party software and systems. Peruse Azure support information for Node.js at: <https://docs.microsoft.com/en-us/javascript/azure/?view=azure-node-latest>

2.6 Angular:

Angular is a toolset for building the framework most suited to your application development. It is fully extensible and integrates well with other libraries. Every feature can be modified to suit your unique development feature needs. Angular combines declarative templates, dependency injection, end to end tooling, and integrated best practices to solve development challenges. Angular empowers developers to build applications that live on the web, mobile, or the desktop.

For this homework, Angular 8+ (Angular 8, 9, or 10) can be used, but **Angular 11 is recommended**. However, please note Angular 8+ can be difficult to learn if you are not familiar with Typescript and component-based programming. <https://angular.io/>

Note: AngularJS (a.k.a Angular 1.0) cannot be used in this project.

2.7 Node.js

Node.js is a JavaScript runtime built on Chrome's V8 JavaScript engine. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient. Node.js package ecosystem, **npm**, is the largest ecosystem of open-source libraries in the world. Peruse Node.js support information at: <https://Node.js.org/en/>

Express.js is strongly recommended. Express.js is a minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications. It is in fact the

standard server framework for Node.js. Peruse Express.js support information at: <http://expressjs.com/>

Important Explicit Notes:

1. In this document when you see GCP/AWS/Azure it implies that you can either use Google App Engine, Amazon Web Services or Microsoft Azure Services, however the CSCI 571 staff will only provide support for GCP.
2. All APIs calls to tomorrow.io should be done through the Node.js server, functioning as a “proxy”. All other API calls should be done through your front-end JavaScript.
3. It is recommended to perform all HTTP calls to your Node backend through the angular framework (as opposed to using jQuery.ajax() / fetch() / XMLHttpRequest())

3. High Level Description

In this exercise, you will need to create a webpage that allows users to search for weather information using the tomorrow.io API and display the results on the same page below the form.

There are two ways the user can provide the location information for which they are trying to look up the detailed weather information. First is using the fields “Street address” and “City”, and/or “State”. Second is using the detect current location box.

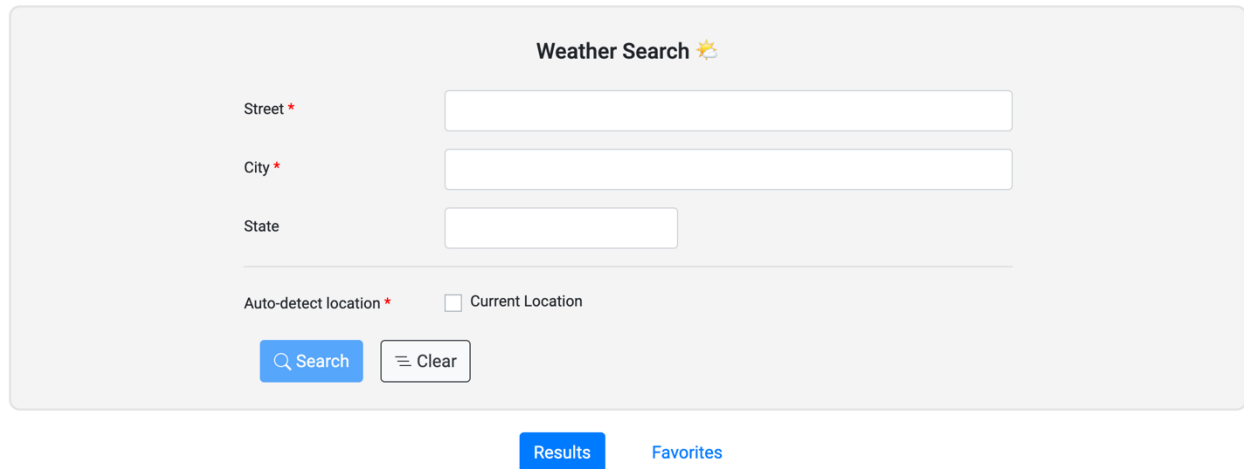
Dynamic Validation must be performed on all the form text fields – i.e., you must not wait until the submit button is hit to verify that the text input is invalid. More information on this is provided below

Upon submitting the search button, upon successful response the results pane should be displayed which contains three tabs namely “Day View”, “Daily Temp. Chart”, and “Meteogram”.

You must display appropriate error messages in cases where you do not get a valid response from the API. In the time between hitting the submit button and displaying the results / error messages you should display a loading bar to indicate that your application is doing something in the background

The webpage should have a Favorites tab, which would support the functionality of adding cities and removing cities from that tab. Additionally, the webpage should have a Tweet button to share the weather details on Twitter. Implementation requirements and details will be explained in the next sections.

When a user initially opens your webpage (landing page), your page should look like **Figure 1**.



The image shows a web form titled "Weather Search" with a sun icon. It contains three input fields: "Street *" (required), "City *" (required), and "State". Below these fields is a checkbox labeled "Auto-detect location *" with the text "Current Location" next to it. At the bottom of the form are two buttons: "Search" (with a magnifying glass icon) and "Clear" (with a reset icon). Below the form, there are two links: "Results" (in a blue box) and "Favorites" (in blue text).

Figure 1. Initial Search Form

3.1 Search Form

3.1.1 Design

You must replicate the search form displayed in **Figure 1** using a **Bootstrap form**. The form fields are similar to Homework Assignment #6.

There are 3 fields in the search form which are **required** if the **Current Location** is not checked:

1. **Street:** Initially, this field is left blank.
2. **City:** This input field should support “autocomplete” which is explained in section 3.1.2. Please note that the user does not necessarily selects the autocomplete suggestions. Initially, this field is left blank.
3. **State:** There are multiple options for the user to select from, which contain all the States of the US as shown in **Figure 2**.

The image shows a web form titled "Weather Search" with a sun icon. It contains four input fields: "Street *", "City *", "State", and "Auto-detect location *". The "State" field has a dropdown menu open, showing a list of US states. "California" is selected and highlighted with a blue bar. To the right of the form are two empty input fields. At the bottom of the form are two buttons: a blue "Search" button with a magnifying glass icon and a white "Clear" button with a trash icon.

Figure 2. State Options

The search form has two buttons:

1. **Search:** The “Search” button should be disabled whenever either of the required fields is empty or validation fails, or the user location has not been obtained yet. Please refer to section 3.1.3 for validation details. Please refer to section 3.1.4 for details of obtaining user location.
2. **Clear:** This button must reset the form fields, clear all validation errors if present, switch the view to the Results tab and clear the results area.

3.1.2 AutoComplete

Autocomplete for cities is implemented by using the *Place Autocomplete API* service provided by Google. Please go to this page to learn more about this service:

[https://maps.googleapis.com/maps/api/place/autocomplete/json?input=\[location\]&key=\[key\]](https://maps.googleapis.com/maps/api/place/autocomplete/json?input=[location]&key=[key])

An example of an HTTP request to the *Places Autocomplete API* Get Suggestion that searches for the city text field “Los” is shown below:

[https://maps.googleapis.com/maps/api/place/autocomplete/json?input=Los&key=\[KEY\]](https://maps.googleapis.com/maps/api/place/autocomplete/json?input=Los&key=[KEY])

To get the API key in the above URL:

1. Please follow the steps provided in the following link:
<https://developers.google.com/places/web-service/get-api-key>

Note:

1. If the API doesn't work, hide the autocomplete feature. This situation should be handled and not throw any error.
2. You can try setting the **type** parameter in the request to "cities" to restrict suggestions to only cities.

The response is a JSON object similar to the one shown in **Figure 3**.

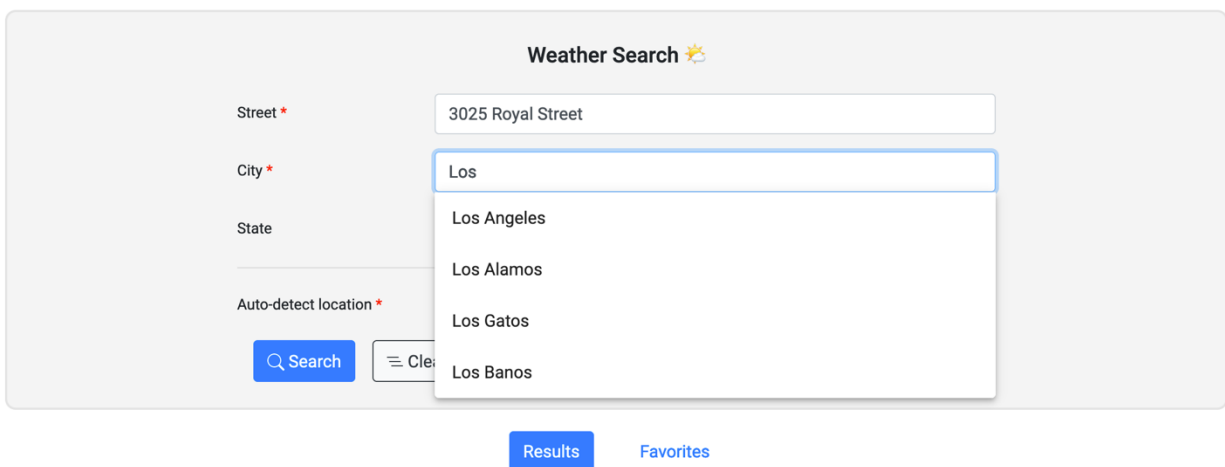
```
1  [
2    "predictions": [
3      {
4        "description": "Los Angeles, CA, USA",
5        "matched_substrings": [
6          {
7            "length": 3,
8            "offset": 0
9          }
10       ],
11       "place_id": "ChIJE9on3F3HwoAR9AhGJW_fL-I",
12       "reference": "ChIJE9on3F3HwoAR9AhGJW_fL-I",
13       "structured_formatting": {
14         "main_text": "Los Angeles",
15         "main_text_matched_substrings": [
16           {
17             "length": 3,
18             "offset": 0
19           }
20         ],
21         "secondary_text": "CA, USA"
22       },
23       "terms": [
24         {
25           "offset": 0,
26           "value": "Los Angeles"
27         },
28         {
29           "offset": 13,
30           "value": "CA"
31         },
32         {
33           "offset": 17,
34           "value": "USA"
35         }
36       ]
37     ]
38   ]
```

Figure 3. Autocomplete JSON Response

The fields we suggest extracting from the API response are highlighted above. The idea is to pick up the city and state from the response, and build a list of (city, state) pairs. Since when a user selects a city from the autocomplete dropdown, we also populate the state field appropriately.

For reference: The values highlighted are the fields in the terms array for every object in the predictions array.

You must use **Angular Material** to implement the Autocomplete. (See section 6.4).



The screenshot shows a web form titled "Weather Search" with a sun icon. It contains four input fields: "Street" (with a red asterisk), "City" (with a red asterisk), "State", and "Auto-detect location" (with a red asterisk). The "Street" field contains the text "3025 Royal Street". The "City" field has an open dropdown menu showing a list of suggestions: "Los", "Los Angeles", "Los Alamos", "Los Gatos", and "Los Banos". Below the input fields are two buttons: a blue "Search" button with a magnifying glass icon and a white "Clear" button with a hamburger menu icon. At the bottom of the form are two blue buttons: "Results" and "Favorites".

Figure 4. Autocomplete example

3.1.3 Validation

Your application should check if the “Street” and “City” edit boxes contain something other than spaces or blank. If not, then it’s invalid entry and an error message should be shown as in **Figure 5**.

If the **Current Location** checkbox is checked, then the Street, City and the State input fields should be disabled, with their values retained.

Please watch the reference video carefully to understand the validation expected behavior.

Weather Search ☀

Street * 3025 Royal

City *
Please enter a valid city.

State California

Auto-detect location * ☐ Current Location

[Favorites](#)

Figure 5. An Invalid Form

3.1.4 Obtaining User Location

As in Homework Assignment #6, you should obtain the current user location using one of the geolocation APIs. The usage of this API has been explained in the Homework Assignment #6 documents. For more information on the IP Info API, visit: the <http://ip-api.com/json>

3.1.5 Search Execution

Once the validation is successful and the user clicks on “**Search**” button, your application should make an AJAX call to the Node.js script hosted on GAE/AWS/Azure. The Node.js script on GAE/AWS/Azure will then make a request to the tomorrow.io API to get the weather information. This will be explained in the next section.

3.2 Results Tab

The results should be displayed below the form as shown in Figure 6. You are supposed to display the results in a responsive mode compatible with mobile devices. If the page is loading on a smart phone or a tablet, the display should be modified according to the width and height of the devices.

In this section, we outline how to use the form inputs to construct HTTP requests to the *Google geocode API* and *Tomorrow.io API* services and display the result in the webpage.

A sample *Google geocode API* call looks like this:

`https://maps.googleapis.com/maps/api/geocode/json?address=[Street + City + State]&key=[Key]`

Note:

1. We use the JSON response to get the latitude and longitude from the Google geocode API.
2. Please refer to the Homework Assignment #6 documentation to get the Google geocode API key.

Once the latitude and longitude are fetched from the Current Location or the Google geocode API, it has to be passed to the Tomorrow.io API to fetch the weather details.

The Tomorrow.io *API* is documented here: <https://docs.tomorrow.io/reference/welcome>

The usage of these two APIs has been explained in the Homework Assignment #6 documents. You may refer to that specification for example requests.

The Node.js script should pass the JSON object returned by the *API* to the client side or parse the returned JSON and extract useful fields and pass these fields to the client side in **JSON format**. You should use **TypeScript** to parse the JSON object and display the results using 3 tabs. A sample output is shown in **Figure 6**. See the tabs on the upper right.

The display of the results is divided into 3 tabs, i.e., **Day View** tab, **Daily Temp. Chart** tab and **Meteogram** tab. The detailed description of all the tabs is given in the following sections.

3.2.0 Results Title

An appropriate title must be set based on the location for which the user is requesting weather information. For example, if the user queries for the forecast through form text input, you can use the data from the City and State fields to display the title

“Forecast at [City], [State]”

However, when requesting for weather information based on the IP address data through IPInfo, you may use the “city” and “region” keys from the ip-info response to display the title “Forecast at ...”.

```
“ ip: "8.8.4.4"
“ hostname: "dns.google"
“ anycast: true
“ city: "Mountain View"
“ region: "California"
“ country: "US"
“ loc: "38.0088,-122.1175"
“ org: "AS15169 Google LLC"
“ postal: "94043"
“ timezone: "America/Los_Angeles"
```

The **Favorite** button (star) provides the user the ability to add or remove the city from the favorites tab. The information saved in the Favorites tab must be retained even if we exit the browser and reload the page and visit it back at a later time. You may use **HTML 5 local storage** to implement this functionality.

3.2.1 Day View Tab

This tab is in a card layout which provides the details of the weather for the next 14 days at the corresponding location provided by the user.

Forecast at Los Angeles, California

[★ Details >](#)

[Day View](#) [Daily Temp. Chart](#) [Meteogram](#)







#	Date	Status	Temp. High (° F)	Temp. Low (° F)	Wind Speed (mph)
1	Tuesday, 05 Oct 2021	 Clear	82.6	60.17	11.52
2	Wednesday, 06 Oct 2021	 Cloudy	74.55	59.22	11.03
3	Thursday, 07 Oct 2021	 Cloudy	70.43	59.85	8.01
4	Friday, 08 Oct 2021	 Drizzle	68.31	61.52	11.99
5	Saturday, 09 Oct 2021	 Clear	72.81	62.19	12.15
6	Sunday, 10 Oct 2021	 Clear	81.41	68.25	11.12

Figure 6. An Example of a Valid Search result

Sample API call to get the data needed to populate the table and details page.

[https://api.tomorrow.io/v4/timelines?location=\[LAT, LONG\]&fields=\[FIELD_NAME\]×teps=1d&units=\[UNIT\]&timezone=\[TIME_ZONE\]&apikey=\[API_KEY\]](https://api.tomorrow.io/v4/timelines?location=[LAT, LONG]&fields=[FIELD_NAME]×teps=1d&units=[UNIT]&timezone=[TIME_ZONE]&apikey=[API_KEY])

This is the same call that you performed for Homework Assignment #6, hence you may refer that spec for details regarding the fields needed.

3.2.2 Daily Temp. Chart Tab

This tab displays a range graph for the min/max daily temperatures for the selected location for the next 15 days. As sample output is shown in **Figure 7**.

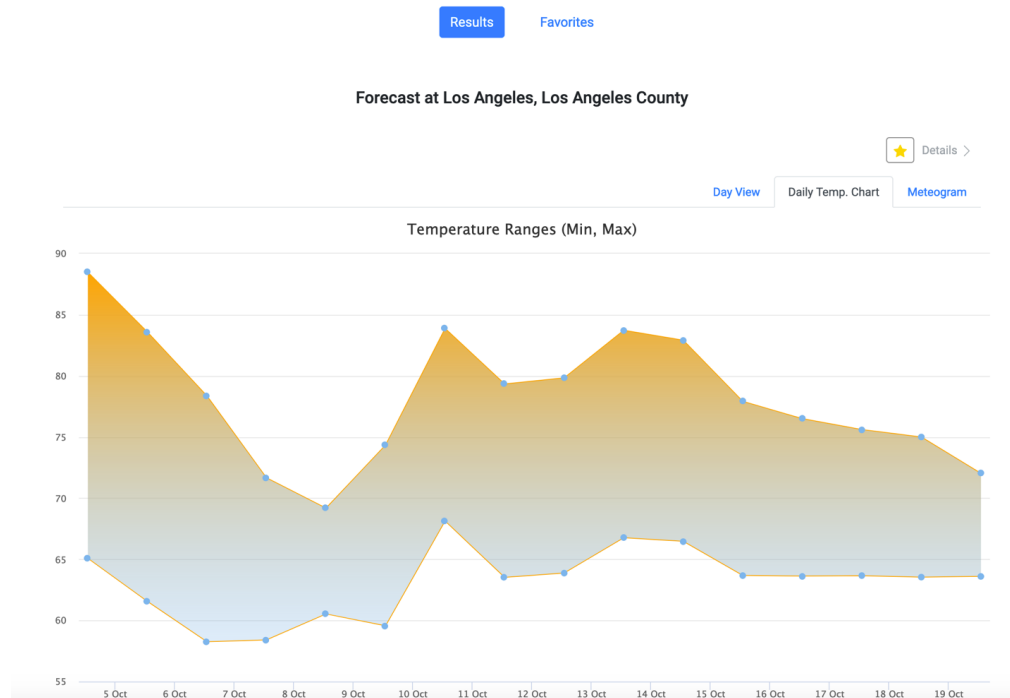


Figure 7. Daily Temperature Range Chart

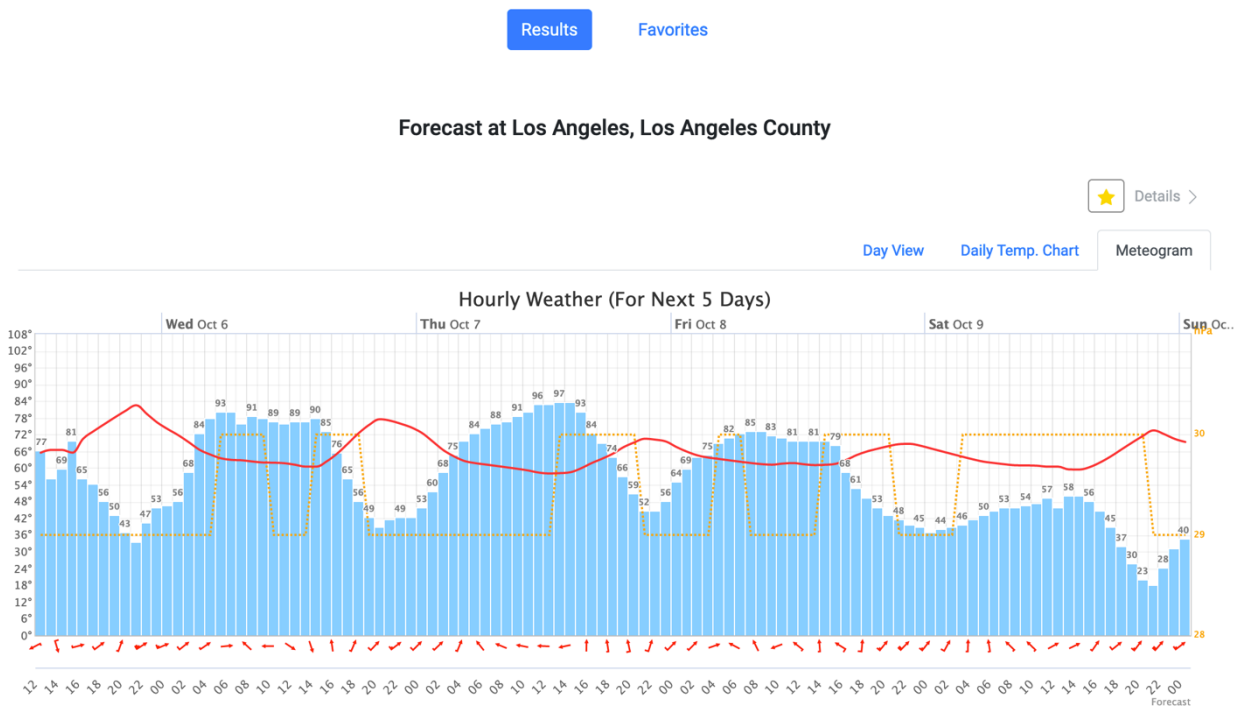
3.2.3 Meteogram Tab

The meteogram chart should consist of an hourly based plot over a period from current time (not the selected) to next 5 days. An example output is shown in **Figure 8**. An example of an HTTP request to the *Tomorrow.io* API that searches for the hourly weather information which is required for this chart is shown below:

[https://api.tomorrow.io/v4/timelines?location=\[LAT, LONG\]&fields=\[FIELD_NAME\]×teps=1h&units=\[UNIT\]&timezone=\[TIME_ZONE\]&apikey=\[API_KEY\]](https://api.tomorrow.io/v4/timelines?location=[LAT, LONG]&fields=[FIELD_NAME]×teps=1h&units=[UNIT]&timezone=[TIME_ZONE]&apikey=[API_KEY])

Reference for the development of this chart is available at:

https://www.highcharts.com/demo/combo-meteogram#https://www.yr.no/place/United_Kingdom/England/London/forecast_hour_by_hour.xml



3.4 Details Pane

On clicking a row from the results table, a new pane with more detailed weather information for the selected day should be displayed. Note: there must be a “slide” transition when moving between the results and details panes.

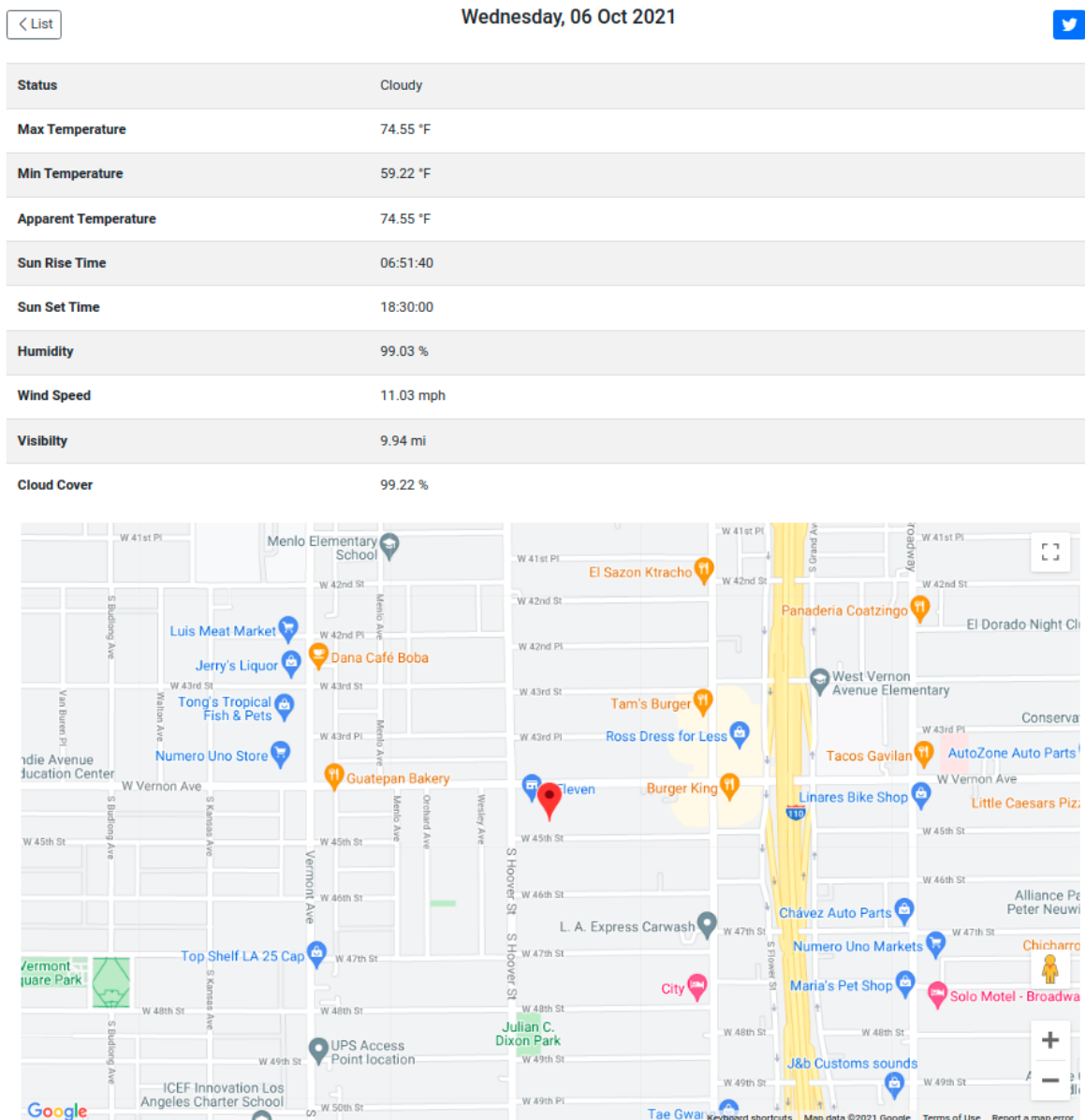


Figure 8. Sample Details Pane

The details pane is a simple table that consists of additional information that is available through the Tomorrow.io API. In addition to the fields from the API, you are also required to display an interactive Google Map section that is centered on the Lat/Long information that you performed in your Tomorrow IO API query on (i.e., lat/long from IPinfo or from the Google Geocode API).

You should use the *Google Maps JavaScript Library*, documented at:

<https://developers.google.com/maps/documentation/javascript/adding-a-google-map> to construct the map

The **Twitter** button allows the user to compose a Tweet and post it to Twitter. Once the button is clicked, a new dialog should be opened and display the default Tweet content in this format:

“The temperature in (City, State) on (Day of week, Date) is (Temperature). The weather conditions are (Summary) #CSCI571WeatherSearch”.

Replace City, State, Date, Temperature and Summary with the actual city searched with the temperature and summary or that day. For example, see Figure 16.



Figure 9. Sample Tweet

Adding the Twitter button to your webpage is very easy. You can visit these two pages to learn how to use Twitter Web Intents:

<https://dev.twitter.com/web/intents>
<https://dev.twitter.com/web/tweet-button/web-intent>



Figure 10. Favorites and Twitter Buttons

3.4 Favorites Tab

In the Favorites tab, the favorite cities are listed in a table format. The user can search for weather information for that city by clicking on the city name in the “City” column.

The information displayed in the Favorites tab is saved in and loaded from the **local storage** of the browser, using HTML5. The buttons in the “Favorites” column of the Favorites tab are only used to remove a city from the list and have a “trash” icon for it to be removed from the Favorites. (Refer to section 5.5 for the needed icons).

The columns in this tab are:

1. Counter (“#”): Just a counter value indicating the number of cities in the favorites list.
2. City: The favorite city the user put in their favorites list. On clicking it should provide the weather details for that city.
3. State: The State information you get from the form input or ipinfo API.
4. Trashcan Icon: To dynamically remove this city from the favorites list.



#	City	State	
1	Los Angeles	California	
2	New York	New York	

Figure 11. Favorites

Please note if a user closes and re-opens the browser, its Favorites should still be there. If there are no cities in the Favorites list, please show ‘Sorry! No records found. (see **Figure 12**).

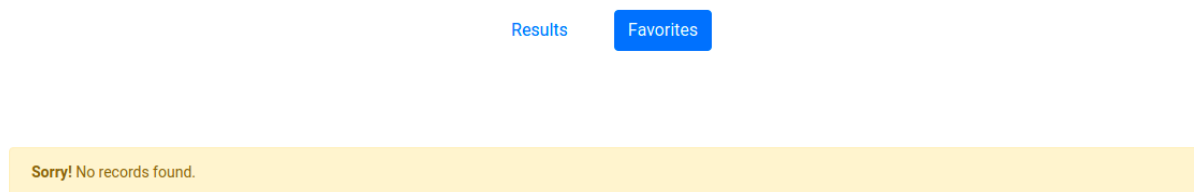


Figure 12. No Favorites

3.5 Error Messages

If for any reason an error occurs due to location error, or reaching the maximum API limit, an appropriate error message should be displayed, as shown in **Figure 13**.

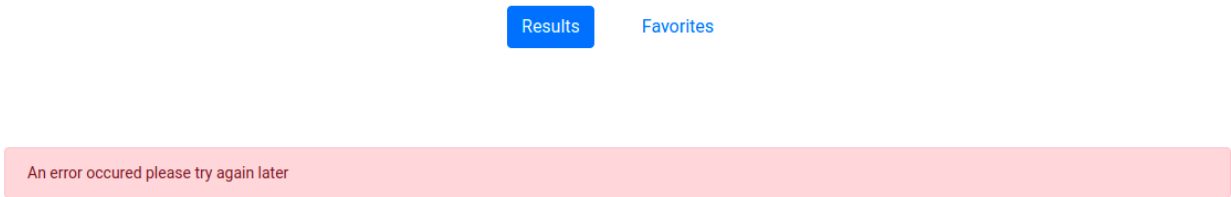


Figure 13. Error Message

3.7 Progress Bars

Whenever data is being fetched, a dynamic progress bar must be displayed as shown in Figure 14.

You can use the progress bar component of **Bootstrap** to implement this feature. You can find hints about the bootstrap components in the Hints section.

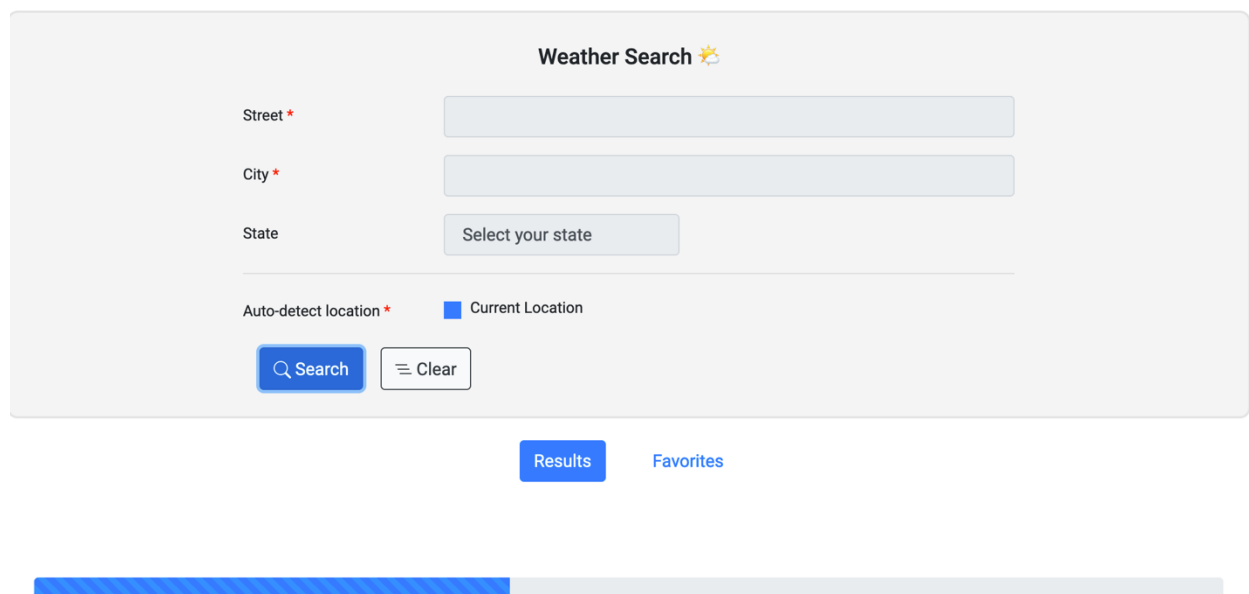


Figure 14. Progress Bar

3.8 List of US States

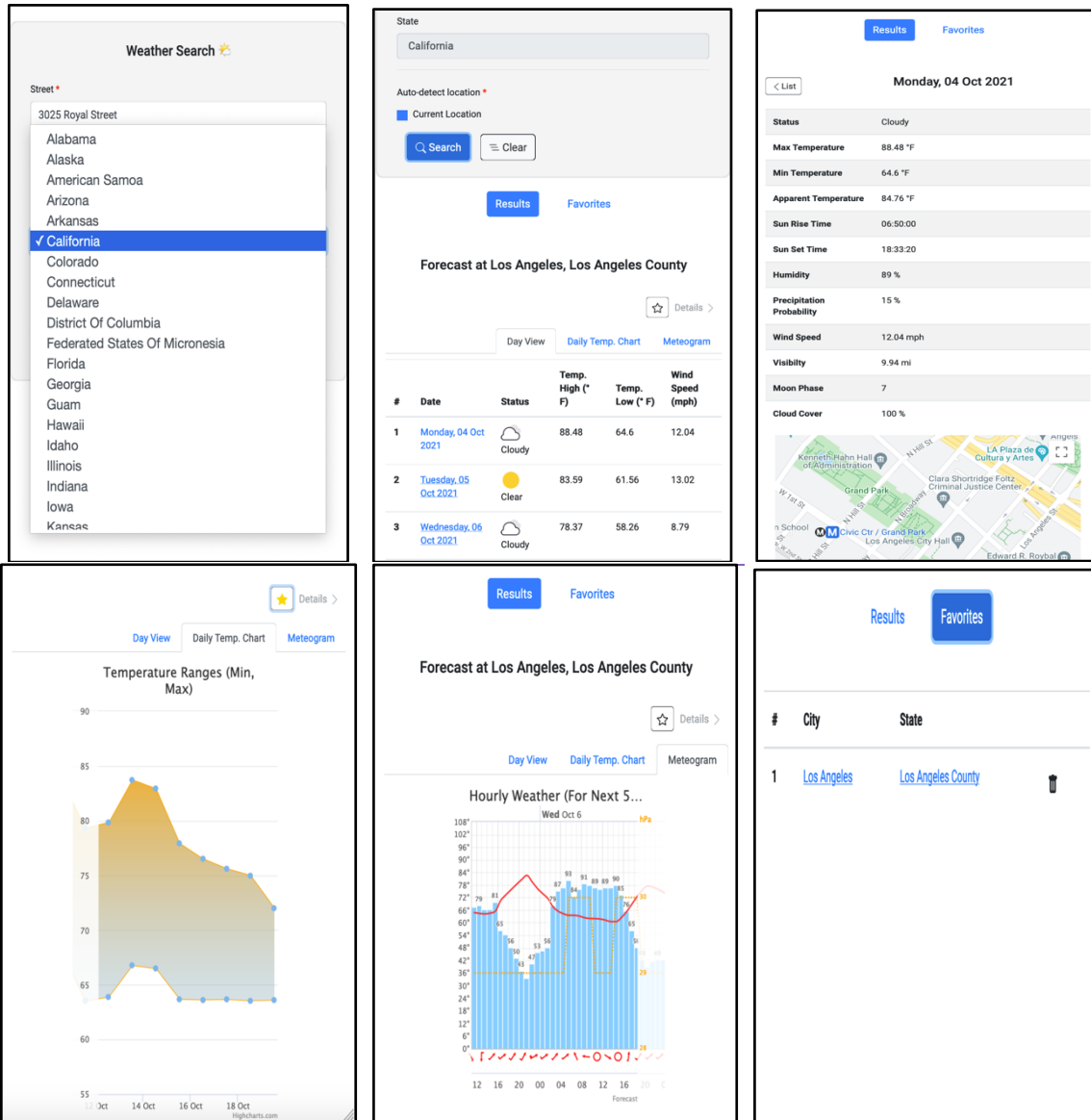
Alabama, Alaska, Arizona, Arkansas, California, Colorado, Connecticut, Delaware, Florida, Georgia, Hawaii, Idaho, Illinois, Indiana, Iowa, Kansas, Kentucky, Louisiana, Maine, Maryland, Massachusetts, Michigan, Minnesota, Mississippi, Missouri, Montana, Nebraska, Nevada, New Hampshire, New Jersey, New Mexico, New York, North Carolina, North Dakota, Ohio, Oklahoma, Oregon, Pennsylvania, Rhode Island, South Carolina, South Dakota, Tennessee, Texas, Utah, Vermont, Virginia, Washington, West Virginia, Wisconsin, Wyoming.

3.9 Responsive Design

The following are snapshots of the webpage opened with Safari on an iPhone 11 Pro max. See the video for more details.

The image displays three sequential snapshots of a 'Weather Search' web application on an iPhone 11 Pro max screen. Each snapshot shows a form with the following elements: a title 'Weather Search' with a sun icon, a 'Street' input field, a 'City' input field, a 'State' dropdown menu, an 'Auto-detect location' section with a 'Current Location' checkbox, and 'Search' and 'Clear' buttons. At the bottom are 'Results' and 'Favorites' buttons.

- Snapshot 1 (Left):** The form is in its initial state. The 'Street' field is empty and highlighted with a blue border. The 'City' field is empty. The 'State' dropdown shows 'Select your state'. The 'Auto-detect location' section has the 'Current Location' checkbox unchecked.
- Snapshot 2 (Middle):** The user has entered 'California' in the 'State' dropdown. The 'City' field is now highlighted with a red border, and a red error message 'Please enter a valid city.' is displayed below it. The 'Street' field remains empty.
- Snapshot 3 (Right):** The user has entered '3025 Royal Street' in the 'Street' field. The 'City' field is highlighted with a blue border, and a dropdown menu is open showing suggestions: 'Los', 'Los Angeles', 'Los Alamos', 'Los Gatos', and 'Los Banos'. The 'State' dropdown still shows 'California'.



You must watch the video carefully to see how the page looks like on mobile devices. All functions must work on mobile devices. Graders will test using Firefox RWD menu, with a smartphone and tablet (like iPhone 12 and iPad) Mobile browsers are different from desktop browsers. Even if your webpage works perfectly on a desktop, it may not work as perfectly as you think on mobile devices. It's important that you also test your webpage on a real mobile device. Testing it in the mobile view of a desktop browser will not guarantee that it works on mobile devices. It is recommended that you use Firefox "Responsive Design Mode" Tool for testing.

4. API Documentation

4.1 Tomorrow.io API

To use the *Tomorrow.io API*, you need first to register for Tomorrow.io Account. This is the same App id used for the Homework Assignment #6.

4.2 Geocoding API

Use *Google Geocode API* to convert the users form input into location information that can be sent to Tomorrow io

<https://developers.google.com/maps/documentation/geocoding/start>

4.3 Places API

Use the *Google Places Autocomplete API* to get the search suggestion for the City text input.

More info: <https://developers.google.com/maps/documentation/javascript/places-autocomplete>

4.4 IPInfo API

Use IPinfo API to get location information from the users IP. See <https://ipinfo.io/>

5. Libraries

- **Angular Google Maps** - This makes it easier to use Google Maps in Angular.
<https://angular-maps.com/>
- **Highcharts Angular** – For each of the charts in the pane/tabs, use HighCharts.
<https://www.highcharts.com/blog/tutorials/highcharts-and-angular-7/>

Note: You can use any additional Angular libraries and Node.js modules that you like. **You cannot use React.**

6. Implementation Hints

6.1 Images

All the icons needed in addition to the images provided for the results table in Homework Assignment #6 can be obtained through bootstrap icons at:

<https://icons.getbootstrap.com/>

6.2 Get started with the Bootstrap Library

To get started with the Bootstrap toolkit, please refer to the link:

<https://getbootstrap.com/docs/4.0/getting-started/introduction/>.

You need to import the necessary CSS file and JS file provided by Bootstrap.

6.3 Bootstrap UI Components

Bootstrap provides a complete mechanism to make Web pages responsive to different mobile devices. In this exercise, you will get hands-on experience with responsive design using the Bootstrap Grid System. You should use Bootstrap 4.0 or later.

At a minimum, you will need to use Bootstrap Forms, Tabs, Progress Bars and Alerts to implement the required functionality.

Bootstrap Forms	https://getbootstrap.com/docs/4.0/components/forms/
Bootstrap Tabs	https://getbootstrap.com/docs/4.0/components/navs/#tabs
Bootstrap Progress Bars	https://getbootstrap.com/docs/4.0/components/progress/
Bootstrap Alerts	https://getbootstrap.com/docs/4.0/components/alerts/
Bootstrap Tooltip	https://getbootstrap.com/docs/4.1/components/tooltips/
Bootstrap Cards	https://getbootstrap.com/docs/4.0/components/card/

6.4 Angular Material

Angular Material (Angular 2+): <https://material.angular.io/>
Autocomplete: <https://material.angular.io/components/autocomplete/overview>
Tooltip: <https://material.angular.io/components/tooltip/overview>

6.5 Material Icon

Icons for the search, clear all, star, star border and delete can be viewed here:
<https://google.github.io/material-design-icons/>
<https://material.io/tools/icons/>

6.6 Google App Engine/Amazon Web Services/ Microsoft Azure

You should use the domain name of the GAE/AWS/Azure service you created in Homework #7 to make the request. For example, if your GAE/AWS/Azure server domain is called example.appspot.com/example.elasticbeanstalk.com/ example.azurewebsites.net, the JavaScript program will perform a GET request with keyword="xxx", and an example query of the following type will be generated:

GAE - <http://example.appspot.com/weatherSearch?lat=xxx&long=yyy>
AWS - <http://example.elasticbeanstalk.com/weatherSearch?lat=xxx&long=yyy>
Azure - <http://example.azurewebsites.net/weatherSearch?lat=xxx&long=yyy>

Your URLs don't need to be the same as the ones above. You can use whatever paths and parameters you want. Please note that in addition to the link to your Homework Assignment #8, you should also **provide a sample link like the ones above**. When your grader clicks on this additional link, a JSON object should be returned with appropriate data.

6.7 Deploy Node.js application on GAE/AWS/Azure

Since Homework Assignment #8 is implemented with Node.js on AWS/GAE/Azure, you should **select Nginx as your proxy server (if available)**, which should be the default option.

6.8 AJAX call

You should send the request to the Node.js script(s) by calling an Ajax function (Angular, fetch() or jQuery.ajax()). You **must use a GET method** to request the resource since you are required to provide this link to your homework list to let graders check whether the Node.js script code is running in the “cloud” on Google GAE/AWS/Azure (see 6.6 above). Please refer to the grading guidelines for details.

6.9 HTML5 Local Storage

Local storage is more secure, and large amounts of data can be stored locally, without affecting website performance. Unlike cookies, the storage limit is far larger (at least 5MB) and information is never transferred to the server. There are two methods, getItem() and setItem(), that you can use. The local storage can only store strings. Therefore, you need to convert the data to string format before storing it in the local storage. For more information: <https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage>

7. Files to Submit

In your course homework page, you must update the Homework Assignment #8 link to refer to your new landing webpage for this exercise. Additionally, you need to provide **an additional link** to the URL of the GAE/AWS/Azure service where the AJAX call is made with sample parameter values (i.e., a valid sample query, with lat/long. See section 6.6).

Combine and submit all your front-end and back-end files (HTML, JS, CSS, TS) electronically as a **SINGLE ZIP FILE** to the DEN D2L dropbox folder so that it can be compared to all other students’ code. **Please do not ZIP a FOLDER containing the files.** Please do not include library, node-js modules, any config files, any angular-cli build files, or any images that we provided or that are included in any library, or any code generated by the tools.

****IMPORTANT**:**

All videos are part of the homework description. All discussions and explanations in Piazza related to this homework are part of the homework description and will be accounted into grading. So please review all Piazza threads before finishing the assignment.