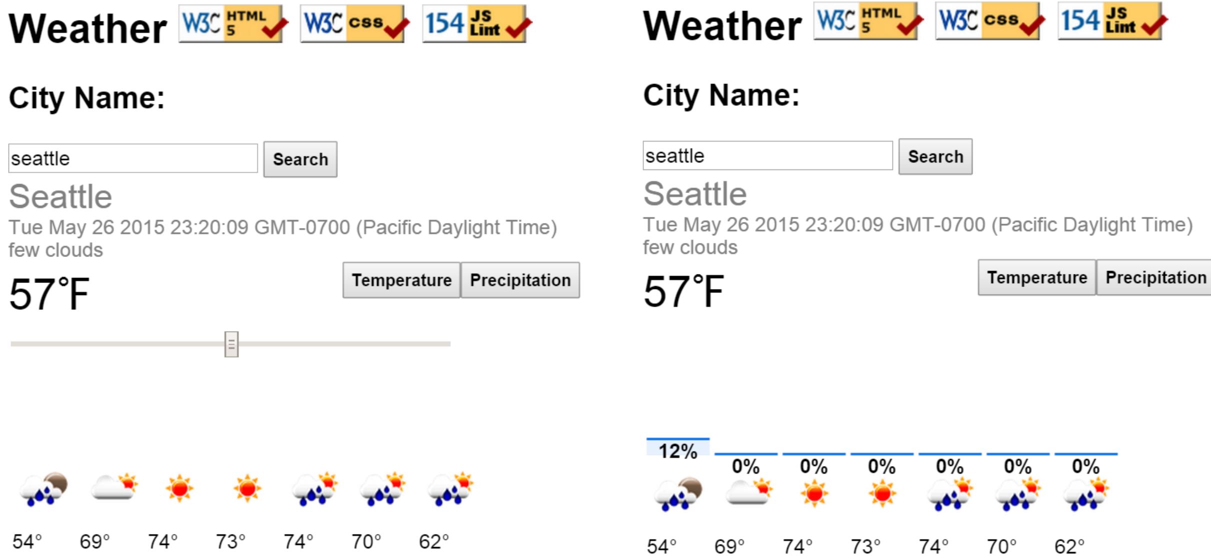
University of Washington, CSE 154
Homework Assignment 5: Weather

Thanks to Marty Stepp and Jessica Miller for pieces of this assignment write up

This assignment is about using **Ajax** to fetch data in text, XML, and JSON formats.



When you Google "weather" you will see some information much like the screenshots above. Multiple places keep track of weather data like this including OpenWeatherMap. OpenWeatherMap provides data in both XML and JSON formats. Your task for this assignment is to write JavaScript code for a web page to display this weather data similarly to how Google does.

We will provide you with the HTML (**weather.html**) and CSS (**weather.css**) code to use. Turn in the following files:

- **weather.js**, the JavaScript code for your weather web page's behavior
- **weather.css**, the CSS styles for your web page's appearance *(you don't need to modify it, but you can if you like)*
- **weather.html**, the weather web page *(you don't need to modify it, but you can if you like)*

You are allowed to modify weather.html and **weather.css** but are not required to do so. You can change the page's general appearance, so long as it meets the requirements in this spec. We encourage you to be creative and have fun.

This program uses Ajax to fetch data from the Webster server. Ajax can only connect to a web server from a page located on that same server. This means that **you must upload your page to Webster to test it**. If you try to fetch data from Webster while viewing the page from your local hard drive, the Ajax request will fail.

Data:

Your program reads data from a web service at **https://webster.cs.washington.edu/cse154/weather.php** . You may assume that all data sent to your program from **weather.php** is valid and follows the formats below.

This web service accepts three different types of queries, specified using a query string with a parameter named `mode`. Each type of query produces output in text, XML, or JSON format. *(You can test queries by typing in their URLs in your web browser's address bar and seeing the result.)* If you submit an invalid query, such as one missing a necessary parameter, your request will return an HTTP error code of 400 (Invalid Request) rather than the default 200.

1. list: The first query mode is `cities`, which outputs plain text containing all city names on file in a plain text format, with each on its own line. The following query would return the results below (abbreviated):

https://webster.cs.washington.edu/cse154/weather.php?mode=cities

```
Abidjan
Accra
Adana   ...
```

The provided web service generally returns the names in alphabetical order, but your code should not rely on this. Just display the names in whatever order the service returns them, and don't make assumptions about the order.

2. current day: The second query mode is `oneday`, which outputs information about the weather for the current day in XML format. In addition to the always-required `mode` query parameter, the `oneday` query requires a second parameter named `city`. The following query returns the results below (abbreviated):

https://webster.cs.washington.edu/cse154/weather.php?mode=oneday&city=seattle

```
<weatherdata>
    <location>
          <name>Seattle</name>
          <country>US</country>
    </location>
    <forcast>
          <time>
                <symbol description="sky is clear"/>
                <temperature>47.62</temperature>
                <clouds chance="0"/>
          </time>
          <time>
                <symbol description="sky is clear"/>
                <temperature>47.93</temperature>
                <clouds chance="0"/>
          </time>
   ...
```

The above is a partial version of the data that you will get back. The real data will have several more `time` tag sections.

If the city you pass doesn't have any data (such as "seeeattle"), your request will return an HTTP error code of 410 (Gone).

3. week: The third query mode is `week`, which returns JSON data about that city's forecasted weather for the next 7 days. The `week` query requires an additional `city` parameter, just like the last query.

https://webster.cs.washington.edu/cse154/weather.php?mode=week&city=seattle

```
{"city":"Seattle",
 "country":"US",
 "weather":[{"icon":"10d","temperature":40.44},
            {"icon":"10a","temperature":41.14},
            {"icon":"10d","temperature":40.14},
  ...
```

If you submit a query for an unknown city, your request will return an HTTP error code of 410 (Gone). The web service returns data starting with the current weather and increasing one day at a time. You may assume that the days are in ascending order by date.

Appearance and Behavior:

All style or appearance aspects not mentioned in the provided CSS file are subject to the preference of the web browser. The screenshots in this document were taken in Chrome on Windows, which may differ from your system.

The HTML page given to you shows a heading of "City Name:" followed by an `input` element with `id` of `citiesinput`. This is followed by a `datalist` with an `id` of `cities`. When the page loads, the box is empty and disabled. Using Ajax, the page should fetch the list of all cities from the web service as described previously, fill the `datalist` with an `option` for each name, and then enable input. More information about `datalist`s can be found on the homework page of the web site.

The rest of the page is inside of a large `div` with the `id` of `resultsarea`. Initially this `div` is hidden, but when the user clicks Search after typing a name in the `input` box, you should make this area appear. *(You can make an area of the page appear/disappear by setting the `.style.display` property on its HTML DOM object.)*

Nothing should happen when the user types a name into the `input` box; wait until they click the Search button to search. You may assume that the user doesn't click Search again until the current search is done downloading.

City data: When the user chooses a city and clicks Search, that city's information should be fetched using Ajax and injected into the page into the `resultsarea`. Specifically, the city name, the date and the description of the weather should be each in their own paragraphs inserted into the `location` div. The city name should have the capitalization it has in the XML and a class of `title`. The current temperature should come next followed by `℉` inserted into the `currentTemp` div. The date should be the current time. You can get this by printing the result of `Date().`

Precipitation data: There is an HTML table with the id of `graph`, initially empty:

```
<table id="graph"></table>  <!-- city percipitation data should be inserted here -->
```

If a city is typed in the input box, you should use the DOM to fill the table with one row of data: a vertical bar representing that city's likelihood of rainfall for that time in the data. For example, the table after searching for Seattle might look like this:

```
<table id="graph">
    <tr><td><div>0%</div></td>          <!-- no chance of rain -->
      <td><div>40%</div></td>           <!-- 40% chance of rain -->
...   <td><div>18%</div></td></tr>      <!-- 18% chance of rain -->
</table>
```

The precipitation bars are drawn as `div`s inside each table cell `td` of one row of the table. Each ranking bar's height should be one pixel for each percentage it is likely to rain. For example, if it was 18% likely to rain the bar would be 18 pixels tall.

Within each bar appears the percentage number for that time. Some lower percentages have numbers that drop below the bottom of the graph; this is expected and you don't need to treat this as a special case.

Precipitation data should initially be computed but it should be hidden until the user clicks the button with the text "Precipitation" on it.

Temperature data: The page contains a `div` with the `id temps`. This contains a slider. When the user clicks the search button and receives a response back, this slider should appear. It should change the temperature displayed in the upper left when moved. This slider should disappear if the Precipitation button is clicked and reappear if the Temperature button is clicked. The temperature displayed should be rounded to the nearest whole number.

The temperature the slider shows represents the temperature for the current day. At the furthest left position it should show the current temperature. One position to the right it should show the prediction for the next 3 hour block of time. You can get all of the XML temperature tags and show the value of the first at the first position on the slider, the second at the second position and so on. You should not make a new Ajax request every time the slider is moved. Instead, store the data you receive back and use it when the slider moves.

Forecast data: The HTML page contains a `table` with the id `forecast`. When the search button is clicked make an Ajax request for the week as described above. You will need to add two rows to the table. The first will contain the images of the weather. These images can be found at:

`https://openweathermap.org/img/w/<icon value from JSON>.png`

The second row will contain the temperature (rounded) for each day followed by a degree sign (escaped as: `°`).

Missing data: Some cities do not have any data. In such a case, the **weather.php** query will return an error code of 410. You should handle this case by displaying a message indicating that there was no data for the chosen city. There is already a `div` in the page with the `id` of `nodata` that contains such a message, but it is hidden by default, so you must show it in such cases.

You can check what kind of Ajax error occurred by examining the `ajax.status` field in your failure handler.

Subsequent Searches: The user can use the page to make multiple searches. When making a search, any data from a previous search should be cleared from the screen. You can remove all HTML content inside an area of the page by setting its `innerHTML` to an empty string. For example, to clear out the `div` with the `id` of `example`:

```
document.getElementById("example").innerHTML = "";     // clear out any child elements
```

Be careful to test your page with several searches in a row. For example, if one search has no data you will have shown the `nodata` message. But on the next search, you should hide this message.

Loading Feedback: In each section of the page where data is shown, there is a small `div` with an animated "loading" GIF image that should be displayed while the data is being fetched from the server. For example:

```
<div class="loading" id="loadinggraph"> <img src="loading.gif" alt="icon" /> Searching...</div>
```

This HTML is already in the provided page, but it is your job to make it appear or disappear at appropriate times. There is a loading area with the `id` of `loadingnames` to the right of the Search button. It should be visible until the page is done loading the list of city names from the server, then it should disappear for the remainder of the page view.

There are three other loading areas with `id`s of `loadinglocation`, `loadinggraph`, and `loadingforecast`. All three should be shown when the user clicks Search to search for data about a name. When the data for one of the Ajax requests arrives from the server and you are finished processing that data, hide the corresponding loading `div`. The page should work for multiple requests; show/hide these loading `div`s properly on each subsequent search as well.

If there is an error with any Ajax request, hide all of the loading areas on the page.

*(The provided **weather.php** web service delays itself by 1-2 seconds to help you test your loading behavior. If you want to test other delays, pass an optional parameter `delay` to the service for the number of seconds you want it to pause before returning its data.)*

Error Handling: If an error occurs during any Ajax request, other than the expected HTTP 410 error when a name's ranking is not found, your program should show a descriptive error message about what went wrong. For full credit, your error message should not be an `alert`; it must be injected into the HTML page. The exact format of the error message is up to you, but it should at least include some descriptive error text about what went wrong. You can inject any error messages into the provided HTML page into the `div` with `id` of `errors`:

```
<div id="errors"></div>   <!-- an empty div for inserting any error text -->
```

You should also hide all "loading" animation images on the page if any error occurs. In order to test your error-handling, try temporarily changing the URL of the web service in your code to a bogus file name such as **notfound.php**. This will trigger an HTTP 404 File Not Found error on every request.

Implementation and Grading:

For full credit, your HTML/CSS code must pass the **W3C validators**. Your JavaScript code should pass our **JSLint** tool with no errors. Your .js file must run in **JavaScript strict mode** by putting `"use strict";` at the top.

Follow the course **style guide**. Separate content (HTML), presentation (CSS), and behavior (JS). As much as possible, your JS code should **use styles/classes from the CSS** rather than manually setting each `style` property in the JS. In particular, no CSS styles should be set in JS other than heights of the precipitation bar `div`s drawn in the graph, or showing/hiding various elements on the page by setting their `display` property. Use **unobtrusive JavaScript**, so that no JavaScript code, `onclick` handlers, etc. are embedded into the HTML.

Follow reasonable style guidelines similar to a CSE 14x programming assignment. In particular, avoid redundant code, and use parameters and return values properly. Make extra effort to minimize **redundant code**. Capture common operations as functions to keep code size and complexity from growing.

Use the **"module pattern"** shown in class to wrap your code in an anonymous function. **No global variables or symbols are allowed**, however one "module-global" var is allowed on this assignment; values should be declared at the most local scope possible. If a constant value is used, you may declare it as a module-global "constant" variable `IN_UPPER_CASE`.

Fetch data using **Ajax**. Be careful to avoid redundancy in your Ajax code; if you do similar Ajax operations in many places, make a helper function(s). Process XML data using the XML DOM. Process JSON data using `JSON.parse`.

Your JavaScript code should have adequate **commenting**. The top of your file should have a descriptive comment header describing the assignment, and each function and complex section of code should be documented. If you make requests, comment about what you are requesting and what your code will do with the data that is returned.

Format your code similarly to the examples from class. Properly use whitespace and indentation. Use good variable and function names. Avoid lines of code more than 100 characters wide.

Do not place a solution to this assignment on a public web site. Upload your files to the **Webster** server at:

https://webster.cs.washington.edu/students/*your_uwnetid*/hw5/weather.html

Copyright © Allison Obourn, licensed under Creative Commons Attribution 2.5 License. All rights reserved.