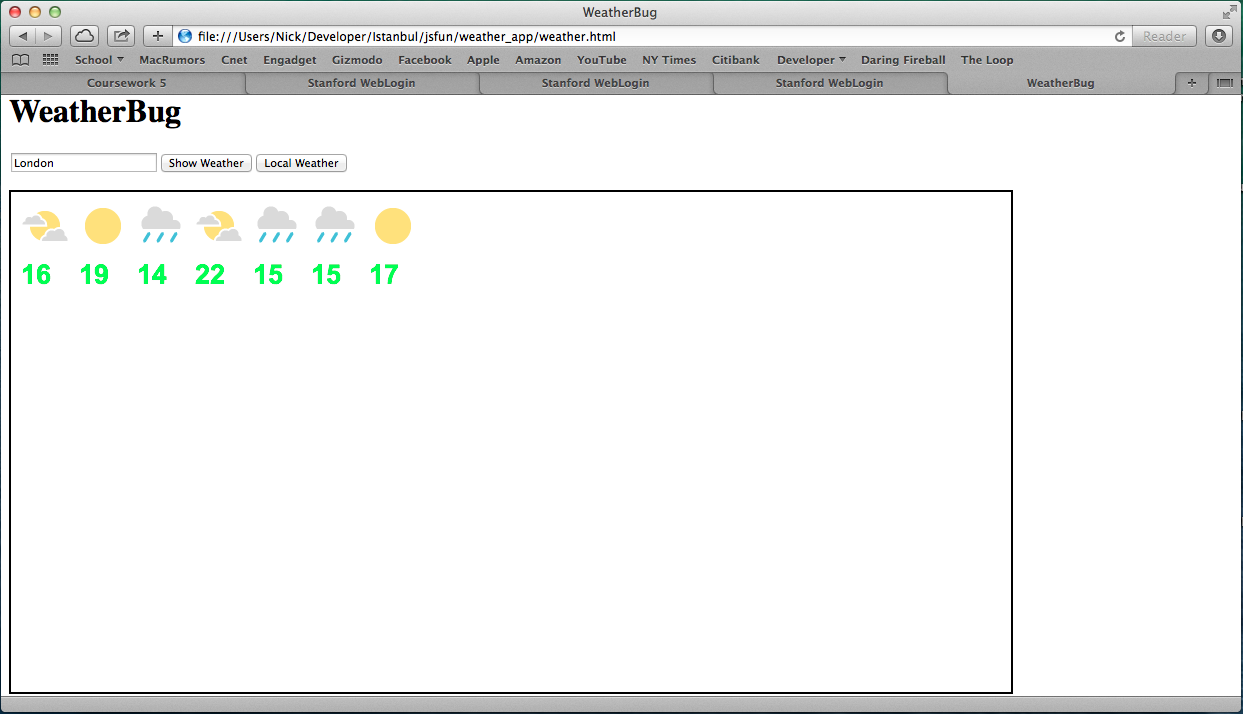
Stanford CS in Istanbul Summer 2014

**JavaScript: WeatherBug**

****

**The WeatherBug App**

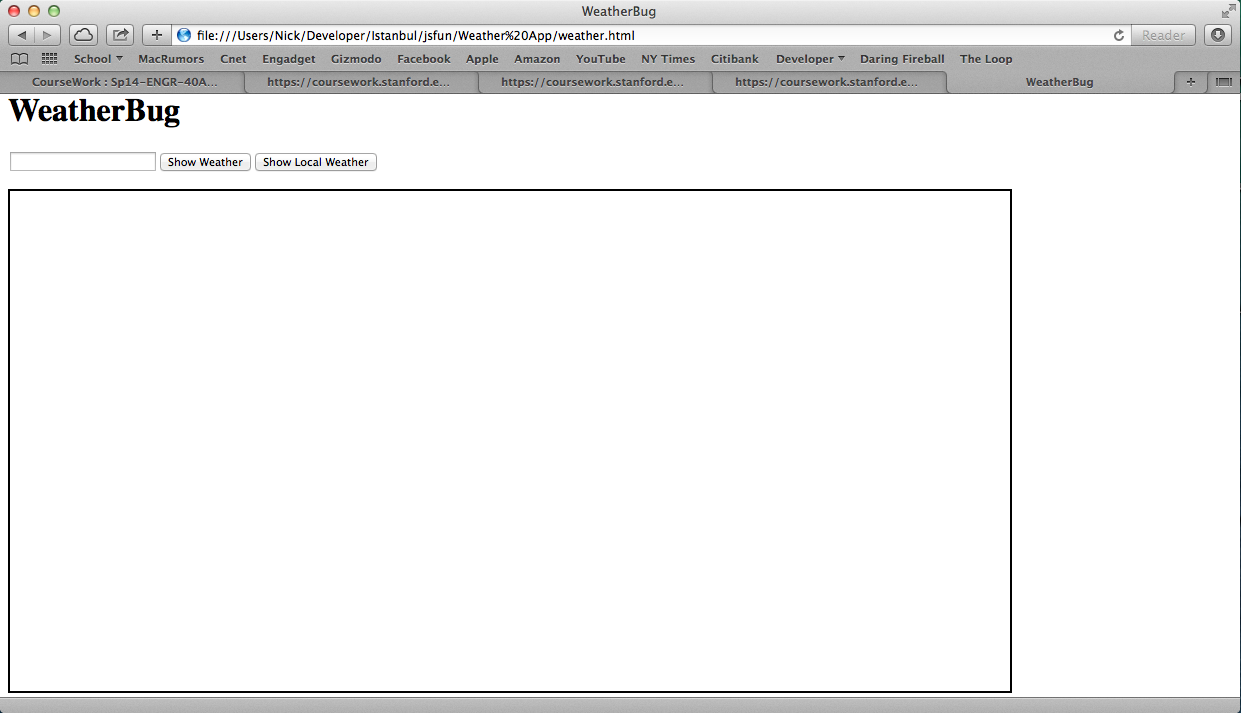
If you’ve ever used weather.com or another weather site to look up the weather, you know how useful (and cool!) it is to be able to look up the weather online for both your current location and any location around the world. And, as it turns out, it’s also a great way to learn about and practice JavaScript callbacks! In this assignment, you’re going to create a webpage that, using JavaScript, fetches and displays weather for any place the user types in, or the user’s current location.

**How’s this going to work?**

For this assignment, we provide you with some starter code and starter files to help you out. The assignment comes with multiple files, but the only 2 that you’ll need to look at are **weather.js** and **graphicsapp.js**. **weather.js** is where you’ll be writing your JavaScript – you’ll notice that the only thing in it (so far!) is a function called **run()** that, just like in Java, is triggered when the webpage loads. You’ll need to fill in the rest! We also provide some helpful functions in **graphicsapp.js** that you’ll need to use when writing your code, like for adding buttons, text fields, and graphics, and fetching weather data.

We suggest you break this down into milestones, and get each milestone working before moving on. We’ve outlined the milestones below, along with the relevant functions in **graphicsapp.js** that you’ll need to use for that milestone. If you break it up, WeatherBug is much easier to manage.

**Milestone 1: Interactors and the GraphicsApp**

The first task you should tackle is getting your program’s interactors – buttons, text fields, and the canvas – all set up. Once set up, your program should look something like this:

Fortunately, we’ve provided some helpful methods to do this:

|  |  |
| --- | --- |
| **addTitle(*title*)** | Adds a title to the top of the screen, and also sets the page title (the name displayed on this page’s tab if you have multiple tabs open). |
| **addTextField()** | Creates and adds a new text field to the top of the screen. Returns the text field object so you can access it later. |
| **addButton(*buttonName*, *callbackFn*)** | Creates and adds a new button to the top of the screen displaying the text you provide in ***buttonName***. You must also pass in a callback function ***callbackFn*** that will be called whenever the user clicks this button. |
| **addCanvas(*width*, *height*)** | Creates and adds a new canvas with the given dimensions to the center of the screen. |

However, there’s one more thing you need to know before you jump into this milestone. All the functions we provide are accessible through something called a GraphicsApp. It’s a type of object we’ve defined for you (like a **GObject**). In order to use any of our functions, you’ll need to create a new **GraphicsApp** object first. Creating one is easy – just write

**var app = new GraphicsApp();**

You should put this line of code at the top of your file, outside of your **run()** function, so that your **app** variable is an instance variable and accessible inside all of the callbacks you will write. Now, calling one of our functions is as simple as saying

**app.addTitle(“WeatherBug”);**

and, voila! A title will appear at the top of the page.

To set up your buttons, you’ll need to create 2 new functions to pass as callbacks to **addButton()**. Remember that when you want to pass a function as a callback, you type the name of the function *without parentheses*. For example, to pass the following function

**function doSomething() {**

**// does something!**

**}**

you would write something like

**app.addButton(“My Button”, doSomething);**

Now, the **doSomething()** function will be called every time the “My Button” button is pressed. You need to add two buttons (and their required callbacks) – one that says “Show Weather” and another that says “Show Local Weather”.

Setting up the canvas is easy – just call **addCanvas()** and pass in the size of the canvas you want (we recommend width = 1000 and height = 500 but you can choose any size you’d like).

Finally, add the text field. **addTextField()** is special in that it *returns the text field that is added to the screen*. Why? So you can access the text that the user types in! Be careful, though – you should only call **addTextField()** in **run()**! That’s because it’s not safe to add user interface elements until the page is fully loaded.

**Milestone 2: Fetching Weather By Query**

Now that you have your interactors set up, it’s time to tackle the first of the two buttons – fetching weather for a query the user types in. You can get the text the user typed in the text field from the **value** property of your text field variable. Otherwise, there are only 2 **GraphicsApp** functions you’ll need for this milestone:

|  |  |
| --- | --- |
| **fetchWeatherForQuery(*query*, *numDays*, *successFn*, *errorFn*)** | Makes an API call to fetch weather data for the given query string. You must also specify the number of days you want the forecast for. Finally, you must pass in 2 callbacks – one that is executed if the API call is successful, the other if it fails. The ***errorFn*** doesn’t take any parameters, but the ***successFn*** must take one parameter – the data that is returned. |
| **displayErrorMessage(*message*)** | Displays an error message onscreen displaying the given text. Note that this error message will remain onscreen until you clear it – to do this, you should call **displayErrorMessage()** and pass the empty string (“”) as the message. |

Remember, when fetching data from the web, the request could always fail for a variety of reasons. That’s why you need to pass a second callback when fetching weather data. If the API call does fail, you should display an error message to the user. Don’t worry about filling in your success callback just yet – we’ll get to that soon.

**Milestone 3: Fetching Local Weather**

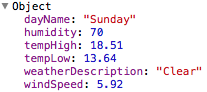
Now let’s tackle the second button – fetching weather for the user’s current location. Instead of getting the text the user entered, you should instead fetch the user’s current location coordinates (latitude and longitude). These functions will come in handy for this milestone:

|  |  |
| --- | --- |
| **getCurrentLocation(*callbackFn*)** | Fetches the user’s current location (in latitude/longitude coordinates). Since this may take some time to calculate, you must pass in a callback function that will be executed when the user’s location is determined. This function should take 2 parameters: the user’s latitude and longitude. |
| **fetchWeatherForCoordinates(*latitude*, *longitude*, *numDays*, *successFn*, *errorFn*)** | Makes an API call to fetch weather data for the given coordinates. You must also specify the number of days you want the forecast for. Finally, you must pass in 2 callbacks – one that is executed if the API call is successful, the other if it fails. The ***errorFn*** doesn’t take any parameters, but the ***successFn*** must take one parameter – the data that is returned. |
| **displayErrorMessage(*message*)** | Displays an error message onscreen displaying the given text. Note that this error message will remain onscreen until you clear it – to do this, you should call **displayErrorMessage()** and pass the empty string (“”) as the message. |

Again, remember that, when fetching data from the web, the request could always fail for a variety of reasons. That’s why you need to pass a second callback when fetching weather data. If the API call does fail, you should display an error message to the user. Don’t worry about filling in the success function just yet – we’ll get to that soon. ***Hint:***you’ll notice that the two **fetchWeather** functions from milestones 2 & 3 require the same type of callbacks. Is there a way to take advantage of that in your code?

**Milestone 4: Understanding the Weather Data**

Now that you have all of your callbacks in place, let’s take a look at the type of data that’s returned from the API call:



This object represents one day of weather. You’ll notice that this object has 6 properties that you can use in your weather app. Your success callback will be passed an array of these day objects – one for each day of weather you requested (so if you requested 7 days of weather, you’ll have 7 objects in your array). Lets say I wanted to get the temperature high for the 3rd day of the forecast – how would I do that? Like this:

**data[2].tempHigh**

In this case, **data** is the name of your success callback’s parameter. As a reminder, you can get the length of a JavaScript array by saying **data.length**.

If, instead of a temperature, we wanted to get the **weatherDescription** field for the same day, we say

**data[2].weatherDescription**

**Milestone 5: Graphics**

Programming graphics in javascript should remind you of programming graphics in Java – the two are quite similar! Still, they differ in some significant ways, so ask questions if you're not sure how to do something. Keep the JavaScript Graphics handout handy, as it has lots of functions you'll need to use to implement this part successfully!

*Cool Stuff: Printing Temperatures*

First, following the example above, once we have a **data** object returned from our weather service, we can get the high temperature for the first day in the object like so:

**data[0].tempHigh**

If we want to print this number to our canvas, we need to draw text on the canvas. How did we draw text on the canvas with Java? We used a Glabel! The procedure is very similar here:

**var label = new Glabel(“Hello World!”);**

Following what you remember from Graphics in Java, create a Glabel for the temperature of one day of the week and show it on the canvas.

Note one tricky difference with javascript: instead of calling

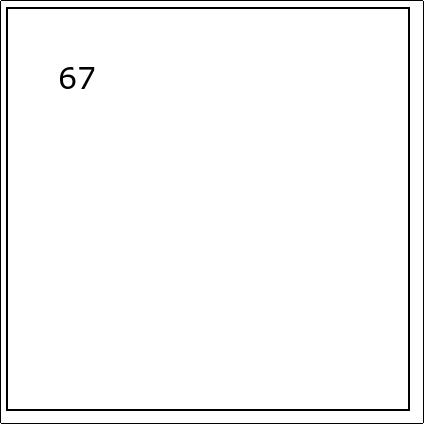
**label.setLocation(x,y)**

we set the **x** and **y** fields of the **position** field of the label, for example:

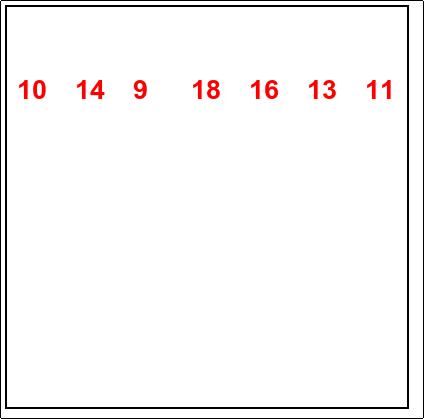
**label.position.x = 50;**

**label.position.y = 50;**

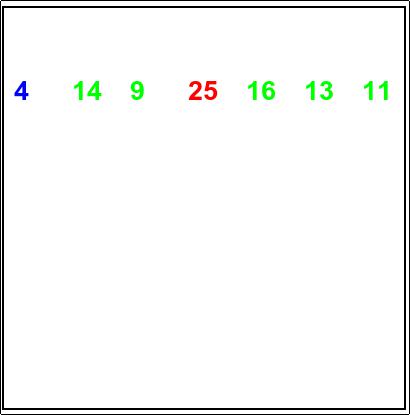
You should get something similar to the below:



Nice Work! Now that you know how to print one temperature, print a whole list of temperatures and make the labels red. Refer to the API table at the end of this section to learn what the framework provides. You should get something similar to the image below:



Now, color your labels by the weather they represent – cold, nice, or hot. To do so, you might consider writing a function that takes a temperature as a parameter and returns a color appropriate for that temperature. You should get something a bit like this:



*Hot Stuff: Weather Conditions*

You know how weather.com always shows a cartoon of what the weather will look like? Well, we can have that too. Remember Gimages from Java? Well, they're back! First, let's create one Gimage and add it to the canvas, about 60 pixels above our text:

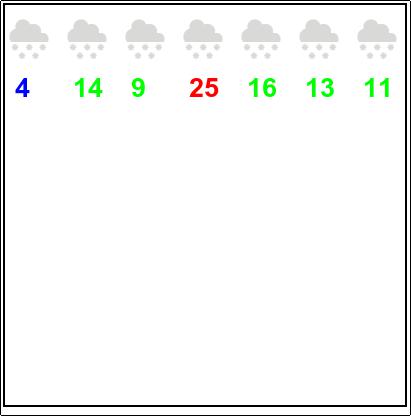
**var image = new GImage("snow.png");**

**image.position.x = TEXT\_X;**

**image.position.y = IMAGE\_Y - 60;**

**add(image);**

Once we've done that, lets add the image above all of the labels!



Great!

The only problem is, you may have noticed it doesn't snow every day! Remember from milestone 4 that you can grab a day's weather condition using

**data[n].weatherDescription**

Using this information, write a function to take in a weather description and returns an image name corresponding to that condition. Note that possible descriptions are “Clear”, “Snow”, “Rain”, or “Clouds”. The image files you're given and their corresponding filenames are given in the table on the next page.

Also note one new thing in javascript: to compare whether two strings are the same, we write:

**“hello there!” === myString**

|  |  |  |  |
| --- | --- | --- | --- |
| A description... | A description... | A description... | A description... |
| “snow.png” | “rain.png” | “sunny.png” | “partly\_cloudy.png” |

One you have this function, let's put it all together! Use your function to display the correct image corresponding to the day's weather description! Your result should look a bit like ours, below. Try some interesting queries to get different weather conditions and temperatures.



AWESOME! YOU FINISHED WEATHERBUG!! If you're not yet feeling weathered, there's lots of cool stuff to add to WeatherBug. Just ask us for ideas or come up with your own!