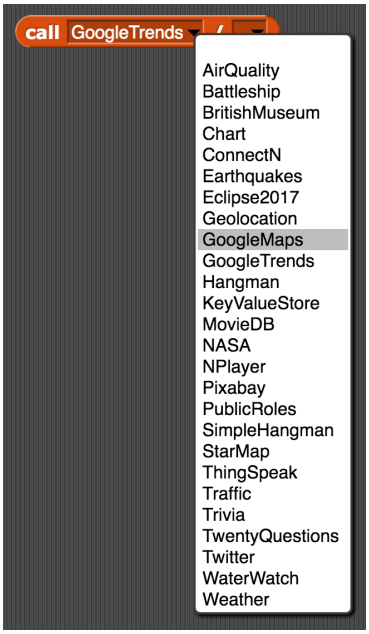
NetsBlox Lesson 1: Introduction to Remote Procedure Calls
Weather App

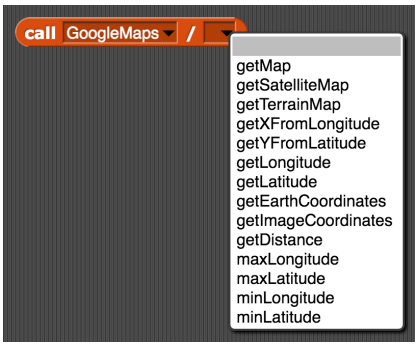
One of the core networking/distributed computing concepts of NetsBlox is Remote Procedure Calls (RPC). An RPC is similar to a custom block: it is a functionality that can be invoked with input values. An RPC is a typically reporter, so it returns some result. The major difference is that an RPC runs remotely, specifically on the NetsBlox server in the cloud.

NetsBlox uses RPCs to provide access to interesting data sources and services already available on the web. Related RPCs are group together into Services. NetsBlox has many of them including Google Maps, Weather, Earthquake, Geolocation, etc.

To use RPCs is very easy. Simply go to the Network tab on the palette and use the `call` block. It has two pull down menus. The first selects the Service from the list of all available ones:



The second one automatically changes based on which service is selected and lists the available RPCs of the given service. For example, here are the RPCs of the Google maps service:



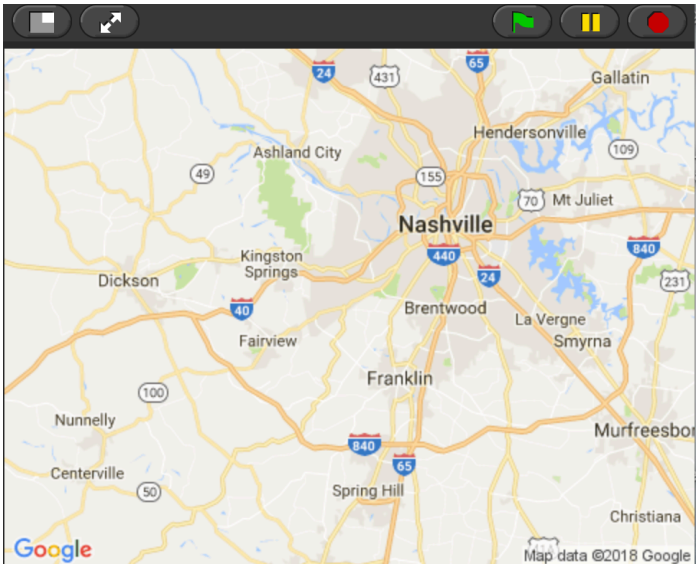
Once an RPC is selected, the block reconfigures to show all its input arguments:

Unlike custom blocks, the name of the input is shown, so RPCs are very intuitive to use. For example, the following call will set the costume of the sprite to a map of the Tennessee area:

We can simply drag in the RPC call block into the switch custom block. The first two parameters are the approximate latitude and longitude for the center of TN, the next two (400, 150) are the width and height of the desired image in stage coordinates and the final input is the zoom level which should be an integer between 1 and 25.

The most common use of the GoogleMaps service is putting it in a script of the stage and using inputs like these:

The sensing tab has the blue blocks above: if the user allows the browser to use his or her location, the **`my latitude`** and **`my longitude`** blocks will return the current location, while the **`stage width`** and **`stage height`** blocks provide the size of the stage. Here is how my stage looks like after calling the RPC shown above:



Step 1: Creating an interactive map background on the stage

The first version of our stage script looks like this:

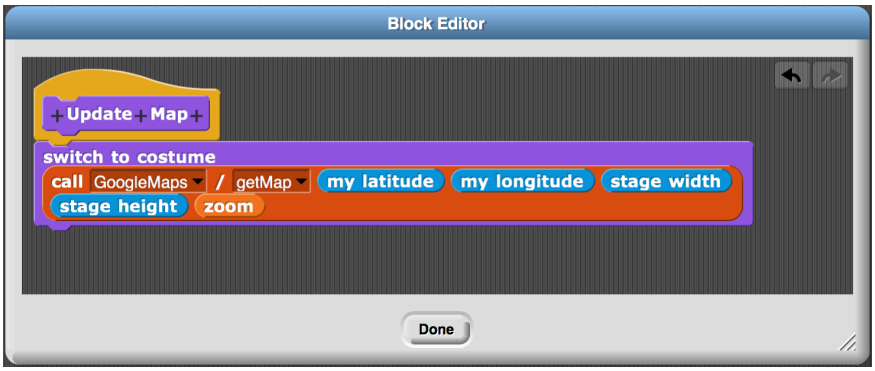
When we click the green flag, the background will change to a map of our current location. The browser may ask for permission to use the current location. On some machines, this feature may be disabled. In that case, no map will show up. If that happens remove the **`my latitude`** and **`my longitude`** and manually type in the desired coordinates.

The next step is to add zooming capability. This can be achieved by introducing a new variable which we should name conveniently *zoom*. This needs to be initialized and used in the getMap RPC call in place of the originally hardcoded value (9 above).

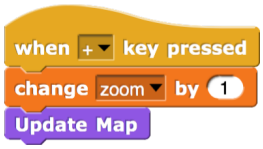
The new version is shown below:

Now we are ready to implement zooming. We will use the + key for zooming in. Using the **`When + key pressed`** hat block, we increment the value of zoom and we need to call the getMap RPC again to get the new map image:

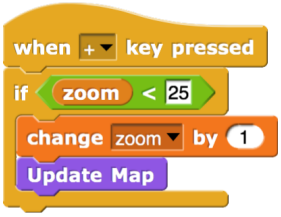
Since the last block is slightly complicated and it is the exact same block used in two places already (and in several more later on), it makes sense to create a custom block called Update Map. In the gray Custom tab, click the gray make a Block button. Provide the name Update Map and select the the purple color corresponding to the Looks block type. Here is how the code should look like:

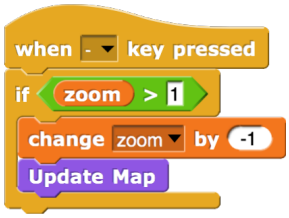


Let's not forget to update the scripts too:



To be precise, we really should make sure that we do not set the zoom value too big. And if-statement does the trick. Finally, the zooming out with the – key should be easy enough to add. Here is how our zoomable NetsBlox mapping app looks like:





Optional Step: Add panning

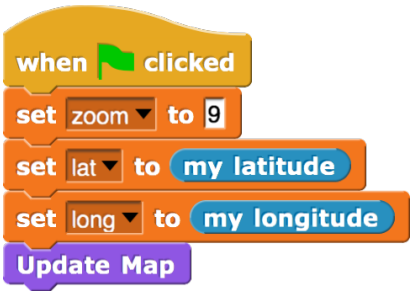
Panning is slightly more difficult than zooming from a conceptual point of view. The difficulty stems from the fact that we need to pan by a different amount depending on the zooming level if we think in terms of latitude and longitude or actual distances. For example, when I am zoomed in to the neighborhood, I would like to pan by a few hundred yards, but at the state level it might hundreds of miles that is needed. The trick is actually quite simple: we need to consider stage coordinates. When we want to pan, it makes sense to move by a half stage. That way half of the same map is still shown. That is, when we pan right, the right half of the old map will be shown on the left half of the stage and a new area is displayed on the right. It does not matter what zoom level we are at, we always move by half a stage.

So, when we pan right (east), the new center of the map will need to be what used to be the right most x coordinate. The y coordinate won't change at all.

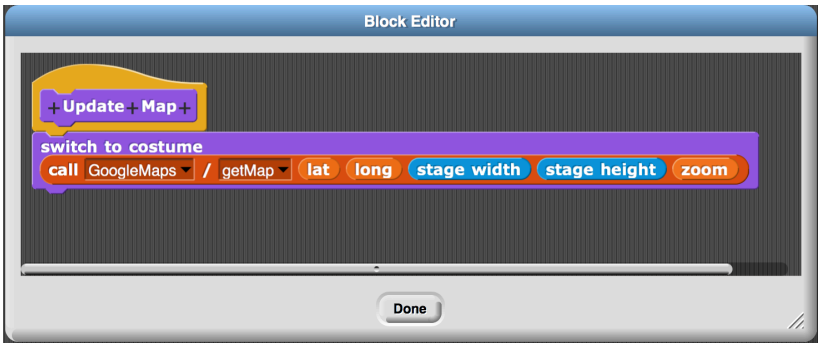
However, there is another complication. The GoogleMaps Service getMap RPC is expecting latitudes and longitudes for the location of the desired map and not stage coordinates. Fortunately, the GoogleMaps service provides coordinate translation RPCs to go from latitude to y, and longitude to x and vice versa.

The final hurdle is realizing that the **`my latitude`** and **`my longitude`** values are no longer suitable since panning will change the current map latitude and longitude values. So, we need two new variables that keep track of the map coordinates. We need to initialize them to my latitude and my longitude respectively, but we need to use these new variables inside the Update Map custom block.

Here is the initialization of the new variables:



And here is the updated Update Map custom block:



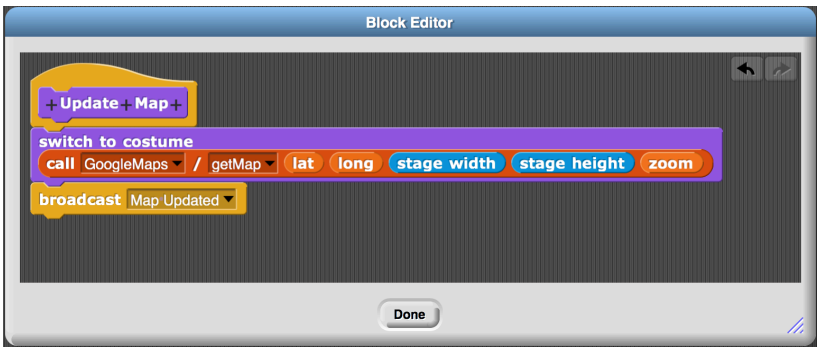
And finally, the panning right (east) code:

We set the new longitude value to the what is currently the right-hand edge of the stage. Its coordinate is half of the stage width (remember the default range for the stage x coordinate is -240 to 240, so that default stage width is 480, therefore, the right edge is 240, half of the width. We use the **`stage width`** variable because the user can change the stage size from the settings menu (cogwheel), so it may not be 480.

But again, that value is a stage coordinate, so we use the getLongitude RPC that expects an x coordinate and returns the corresponding longitude value on the current map.

Adding panning for west, north and south is straightforward based on the code above.

To be able to notify the rest of the program when the map changes and when the user clicks on the map, we broadcast two new events: Map Updated and Map Clicked. The former is called from the custom block, as any time there is a change in the map background the sprites of the applications may need to do something.



While the latter uses a new script with the **`When I am Clicked`** hat block. The final code for the stage implementing a fully interactive map of the world is shown below:

A potential improvement could be limiting the possible values for the latitude because too close to the poles, the behavior of the map may be unusual due to the projection of the surface of a spherical object (Earth) to a plane. That is left as an exercise for the reader.

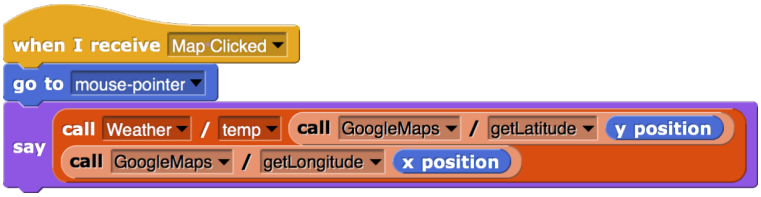
Step 2: Displaying current weather conditions

Our goal is to display current conditions wherever the user clicks on the map. We will use a sprite to do that. It will need to handle the event Map Clicked sent by the stage program. The first thing the sprite should do is jump to wherever the user clicked. Here is the beginning of our sprite script:

The Service we are going to use is called Weather and among its RPCs is one called temp for returning the temperature.

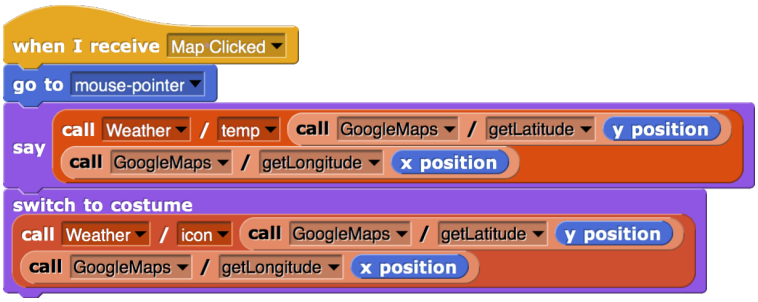
The RPC returns a number representing the current temperature at the given position in Fahrenheit. Not surprisingly, the RPC expects the position in Earth coordinates. But all we have is the x and y stage coordinates of the sprite. Fortunately, the GoogleMaps service provides coordinate translation RPCs to go from latitude to y, and longitude to x and vice versa. (The optional panning task described previously used the same RPCs). Here they are:

As you can see, they are expecting a stage x and y coordinate and return the corresponding longitude and latitude on the current map. Now we are ready to display the temperature on the map:



Notice how it is perfectly fine to embed RPC calls into places that expect a value like the lat and long inputs for the temp RPC call. RPCs are reporters, the getLatitude call reports a number, a latitude value. That is exactly what the temp RPC is expecting. When executing this command, NetsBlox will simply call the getLatitude and getLongitude RPCs first. Once they reported the resulting values, NetsBlxo will call the temp RPC with those inputs. When the temperature is reported back, NetsBlox will execute the say block to display the number next to the sprite.

The temperature service has another RPC that returns a small weather icons representing current conditions. Let's use it to change the custom of our sprite!



The new block is very similar to the one above. Notice that we are using the exact same RPC calls twice here: the coordinate transformations are carried out twice. Getting data from the network (that is what RPCs are doing essentially) is typically slower than doing something locally. There is always some network latency involved as multiple computers need to communicate with each other to carry out our request. In our case, the RPC call first goes to the NetsBlox server running somewhere in an Amazon data center. In turn, the NetsBlox server issues a request to the Open WeatherMap service running somewhere else. It may be the same data center or it may be thousands of miles away. That computer will get the data we requested, report it back to the NetsBlox server, which report it back to our computer. It is an amazing feat that all this happens and happens very fast. Nevertheless, sometimes there is a noticeable delay. Sometimes it may take up to a few seconds depending on network conditions and the load on the various servers involved. (Recall how sometimes a Netflix show takes time to start or even pause mid vide for "buffering." That is the same phenomenon.)

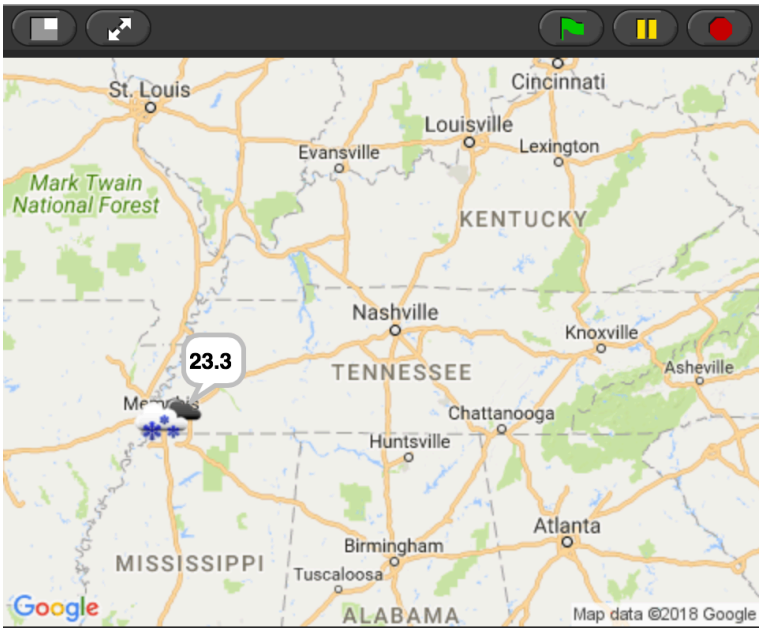
Long story short, we should avoid making unnecessary RPC calls. In our case, we can simply carry our the coordinate translation once, save the value in a variable and use it. Since we are dealing with latitude and longitude values, lat and long would be natural variable names. However, we used those exact same names in the stage code. If we did not make them "for this sprite only" using the same names may cause problems. Also, conceptually, the lat and long variables in the stage correspond to the center of the map, while here we want to store the sprite's location. So, it is best to use different names.

There is also a nice way in NetsBlox to create temporary variables that are only need locally in the given script. In our case, we only need these variables right here, so we'll use this feature. We drag in the block below from the variables tab:

By clicking the arrow keys, we can add or remove additional variables. By clicking the variable itself (e.g., a above), we can rename it. Here is the newest version of our weather sprite script:

First we declare that we are using two new script variables (variables that are only visible in the given script) called sprite lat and sprite long. Then we set their values using the GoogleMaps coordinate translation RPCs. Finally, we use these latitude and longitude values stored in these variable to call the temp and icon RPCs of the Weather service.

When you run this program and click somewhere on the map, you should see something similar to this:



As you can see, it is quite cold and snowing in Memphis when I am writing this. A rare event!

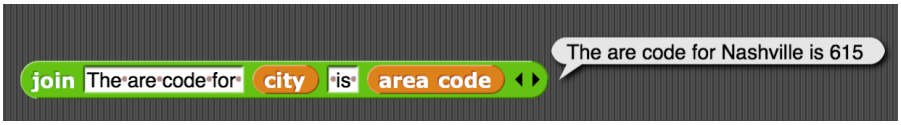
Now we have a fully functional weather application that displays current conditions anywhere on Earth. Pretty cool!

Optional Step: Making improvements

There are a number of ways we can improve our initial weather app. Let's start with an easy one!

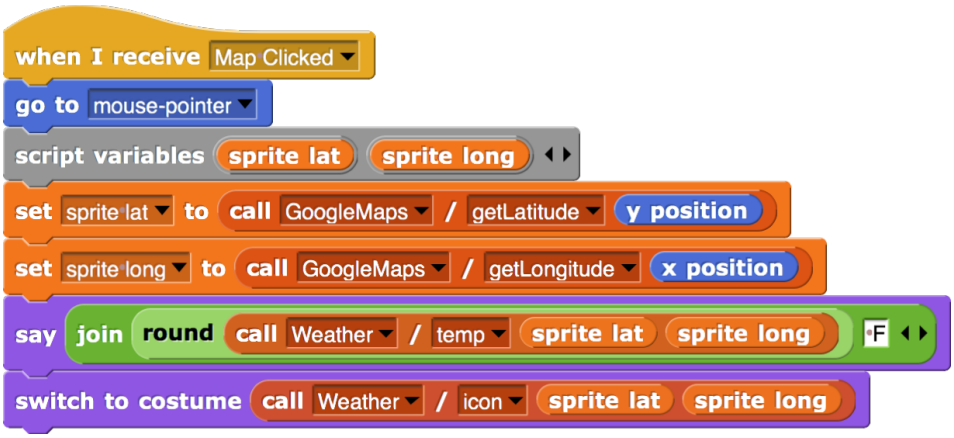
Improvement #1:

Most of our international users are used to dealing in Celsius and not Fahrenheit. The least we can do is indicate that the numbers we are display are in fact in Fahrenheit. That is easy enough: we will simply use the join command under the Operators category. The join block allows us to create text from various parts:



As you can see, we can use variables or reporters like RPCs etc. Note that spaces are highlighted by little red dots.

So, our modified weather script looks like this:

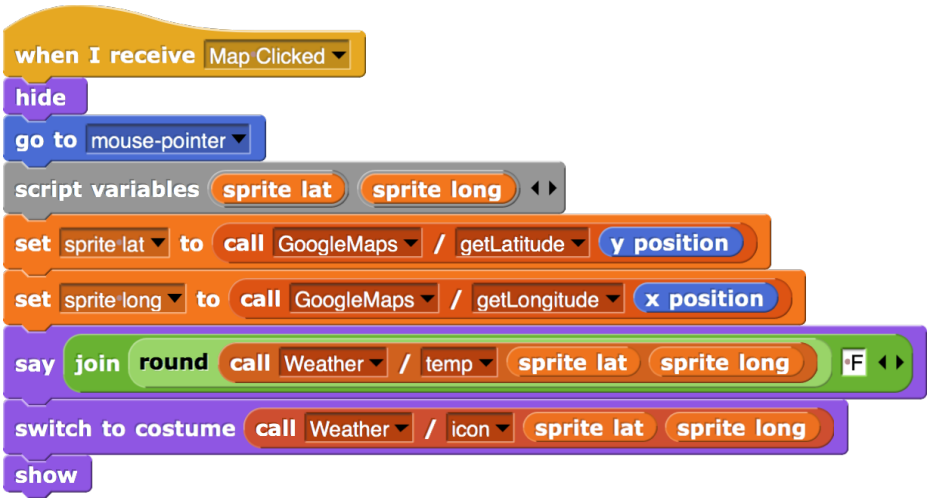


Notice that we also added rounding so that we display full degrees only.

Improvement #2:

As you play with the app and keep clicking around, you may have noticed that the sprite jumps to the new place while still displaying the icons and the temperature value for the old one for a short while. This is again due to network latency: it takes a second to get the weather data and our sprite will keep its old look in the meantime.

There is an easy enough fix for this: simply hide the sprite when the map is clicked and only show it once the new data is available. Here it is:



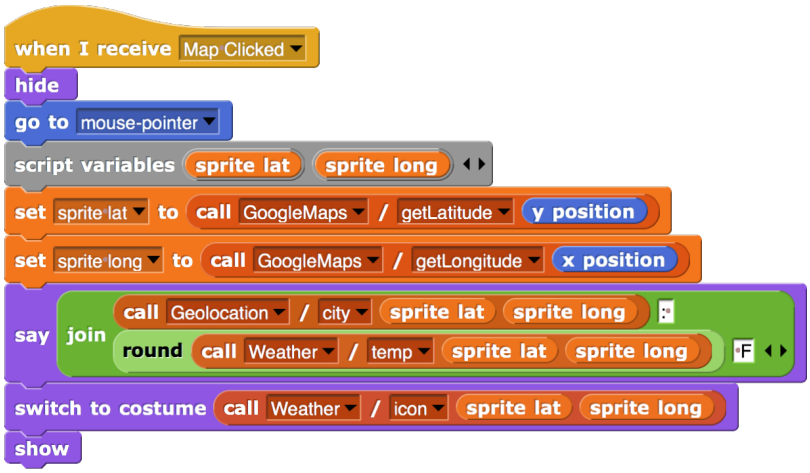
Improvement #3:

An even worse problem is if we zoom or pan the map, the sprite will remain in the old position on the stage and show weather data that is now at an incorrect place! How can we solve that? Remember the event the stage broadcasts when the map has changed? It was called Map Update. Since we changed the map, it makes sense to simply hide the sprite until the user clicks on a new position. Yet another easy fix:

Improvement #4:

The map only shows the major cities in the given area. If we clock on the map somewhere where there is no city name shown, we still get the weather data. But it would be a nice feature if we also saw the name of the city near that location. Fortunately, the geolocation service has and RPC for that:

We will simply use this RPC and display what it returns along with the temperature. Here is our final script for the sprite:



And here is the app in action: