Lab – Cisco Coffee JavaScript Application (Optional)

Objectives

Part 1: Exploring a Web Application

Part 2: Viewing the Code of a JavaScript Web Application

Background

This activity focuses on a web application designed to monitor coffee fields. The Cisco Coffee JavaScript application is a simple example of programming code that can be used to report sensor status and provide alerts. Connected to simulated temperature, moisture, and light sensors, this web application monitors and displays information about the values read by the sensors.

Note: This application does not communicate with real sensors. It is designed to be a simulation and to provide a glimpse into using JavaScript for web programming. You will need to download the **Cisco Coffee JavaScript Files.zip** file in order to complete Part 2 of the lab.

Part 1: Exploring a Web Application

The choice to deliver this application across the web was due to the flexibility of the vast majority of network-ready devices that have a web browser installed. Computers, phones, and tablets should have no problems loading and running this web application on their respective web browsers.

When the application is loaded on a web browser, it begins to display the simulated readings captured by field sensors. You should be able to see temperature, light, and moisture values. The application also has a logging area (towards the bottom of the screen), which is used to display unusual events, such as storms. If the values captured by the sensors fall out of the predefined limits, this event is also tracked in the logging area.

Three buttons display (from left to right): Previous day Average, Next Day Average, and Show Real Data. The first and second buttons allow the farmer to look at the average temperature, light, and moisture levels from previous days. The application stores averages of up to seven days. The last button allows the farmer to toggle between simulated and real data.

Note: No real sensors are connected to this application. That is why when you click **Show Real Data** the display shows **No sensor data available**. Click **Show Simulated Data** to display the current day's values.

Part 2: Viewing the Code of a JavaScript Web Application

The application in this activity was written in JavaScript language because it is easy to learn. If you are interested in learning more about coding, the next steps should be useful to you.

Step 1: Open the code in a web browser window and in a text editor.

Web applications are (essentially) web pages and, therefore, it is always possible to look at the code used to create the page displayed on the browser window. You can view the source by right-clicking anywhere in the figure and choosing an option similar to **View Source** or **View Frame Source,** depending on the browser. Scroll down to the section in the code that starts with **Script**. Lines that began with a double forward slash (**//**) denote comments. The comments provide a brief explanation of the code. Alternatively, complete the following steps to download and open the web application in a text editor:

a. If you have not already done so, downloaded the **Cisco Coffee JavaScript Files.zip** file for the page where you downloaded these instructions.

b. Unzip the file, locate the folder, and open it. You should see two files: *Cisco_Coffee_JavaScript.html* and *pouring_coffee_grd.png*. The first file contains the code itself while the second file is the background image used in the application. The file containing the code can be opened using any text editor. Feel free to use your favorite text editor to open this file, but this activity uses Microsoft Notepad because it is already included in the Windows package.

Note: Avoid using Microsoft Word when creating or editing web pages. The formatting including in Word is known to create unwanted placement of images and text on the web page.

c. Select and right-click the *Cisco_Coffee_JavaScript.html* file. Select **Open With** > **Notepad**. You should now see a Notepad window displaying the web application code.

Note: Notepad is limited to only showing the code. If you would like to see additional features, such as different colors for different parts of the code as well as line numbers, download a free text editor like **Notepad++**.

Step 2: View the HTML section of code in the web application.

As you may have noticed, the file under examination has an .html extension, indicating that it is an HTML file. HTML is the main language used to build web pages. It allows the developer to organize the information to be displayed on the web page. While web browsers are somewhat forgiving regarding HTML syntax, HTML files have a well-defined structure, divided into sections. Common HTML sections are HTML, HEAD, and BODY. To mark these sections, HTML standards define HTML tags. HTML tags usually mark the beginning and end of a section, making anything in between part of the section. The standard also dictates that the greater than, less than (**<>)** characters are used to indicate a tag. For example, consider the HTML code below:

```
<HTML>
    <HEAD>
        <TITLE>HTML Example Page</TITLE>
    </HEAD>
    <BODY>
       <P>This text is to be formatted as a paragraph on the page.</P>
       <P>And here is another paragraph</P>
    </BODY>
</HTML>
```

The first line of the example marks the beginning of the HTML document. The second line opens the `HEAD` section. The page title is the text shown as the window name and the third line defines what the page title is. HTML standards say that the title should be placed under the `HEAD` section. Notice the `</TITLE>` tag at the end of the third line: the slash character (`/`), preceding the tag name, marks the end of a section. Because the `HTML Example Page` text is placed inside the `TITLE` section, web browsers display that text as the window name.

The `BODY` section is where the body of the page is placed. The example above creates a very simple page with only two paragraphs. Text placed inside a paragraph tag is displayed as paragraphs by web browsers. `<P>` and `</P>` mark a paragraph section. The `BODY` section is then closed by the addition of `</BODY>` in the fifth line.

Lastly, the `</HTML>` marks the end of the HTML section. Because the HTML section should contain the entire HTML page, this tag also marks the end of the HTML file.

Step 3: View the SCRIPT section of the code in the web application.

The HTML standard also defines many other tags. To use the web application efficiently, other tags are used. Another important tag is the `<SCRIPT>` tag. `<SCRIPT>` tags host client-side scripts, such as JavaScript. It is called the client-side script, because it is executed at the client's computer, not in the server. A script is required because HTML tags (alone) cannot be programmed to make decisions; their purpose is only to organize information. All of the application logic is placed in the `SCRIPT` section.

Web applications are usually based on functions. A function is a block of code written to perform a specific task. When it is defined, a function can be invoked from another part of the code when the task it performs is required. In a JavaScript application, functions are defined by following format:

```
function function_name (optional_variables_to_be_passed_into_function) {
      function code block (may span over several lines)
      function code block (may span over several lines)
      function code block (may span over several lines)
}
```

Look for the following functions in the application:

```
init(); prepareToggle(); resetExtremes(); fakeValues(); dayOfWeek(d);
showPastDaysAvgs();
```

Step 4: Describe each of the major functions in the code.

The application has its own workflow that gives the order in which its functions are invoked. Below is a summary of the application flow:

a. The *onload="init();"* parameter added to the BODY HTML tag towards the end of the code instructs the web browser to invoke the init() function as soon as it finishes loading the page. The *init()* function is located toward the top of the code right after the block of code that declares variables.

b. *init()* sets up some environment parameters, such as display size. It also specifies that the *fakeValues()* function executes every three seconds, and *resetExtremes()* function executes every 60 seconds.

c. *resetExtremes()* calculates random maximum and minimum values for temperature, moisture, and light. This is necessary to simulate the values during a normal day (no storms). After three seconds, *fakeValues()* is invoked (as defined inside *init()* above) to randomly choose a number between these extreme values. This technique is used to simulate the values moving within a very narrow range, thus, recreating the normal operation of real sensors.

d. By the time *fakeValues()* is invoked, the extreme values are already calculated. *fakeValues()* uses these values as the ceiling and floor for randomly choosing values.

e. Lines 338, 339, and 340 define the previous day, next day, and real/simulated data toggle buttons, respectively. These buttons are displayed on the web page and the JavaScript code calls these buttons by their ID: pDay, nDay, and toggleBTN.

f. Note: The code defines that the *showPastDaysAvgs(this.id)* function should be invoked when the previous day or next day buttons are clicked. The this.id parameter passes the identifier of the clicked button to the function, allowing it to take a different action depending on what button was clicked. When the Show Simulated/Show Real Data button is toggled, *prepareToggle()* is invoked.

g. *prepareToggle()* is invoked when the user wants to toggle between real and simulated data. To stop displaying simulated data and switch to real data, *prepareToggle()* stops invoking *fakeValues()*. Because this application is a simulation and no real sensors exist, *prepareToggle()* also replaces the simulated data shown on the screen by the No sensor data available. message. If the user wants to view simulated data again, *prepareToggle()* re-instates the periodic execution of fakeValues() and removes the No sensor data available message from the screen. *fakeValues()* once again populates that area with the results of its calculations.

h. *dayOfWeek()* is a support function used to calculate the last seven weekdays. This is important because the application must display the last seven days, independent of the day the user runs the application.

i. Lastly, *fakeValues()* is also responsible for populating the log area at the bottom of the application. Every time *fakeValues()* displays a value that falls off a predefined limit, it logs that event in the log area.

j. *resetExtremes()* also has an extra task: it simulates a storm every three minutes. It does so by setting extreme values far off the normal range of values. Because the extreme values are now too high and too low (ceilings and floors, respectively), the results calculated by *fakeValues()*invariably falls off the predefined limits. This will look like a storm just rolled in.

Step 5: Explore and manipulate the code on your own.

a. Using the previous description of the functions, examine the code and see if you can find additional places where the logic is used to generate the web application.

b. Feel free to make changes to the code and see the results. If you break the code and cannot correct it, use the code in the course to find and correct your errors.