Lab 5: Cyber-Physical Systems and Visualization

Due Friday October 26, 2018 @ 11:59pm PT on bCourses [Optional]

Recommended Recourses for Learning Web Design with HTML, CSS, and JS:

- W3Schools (http://w3schools.com) includes detailed reference materials for HTML and CSS and extensive tutorials and examples for HTML & CSS, JavaScript, SQL, and jQuery.
- Code Academy (https://codecademy.com) is a free website for learning to code through interactive lessons and includes lessons in HTML & CSS, JavaScript, jQuery, and Python. The "Make a Website" lesson (https://codecademy.com/skills/make-a-website) is particularly useful for beginners interested in web development.
- Code School (https://codeschool.com) is a free and paid website for learning to code through interactive lessons and includes lessons in HTML & CSS, JavaScript, jQuery, and iOS app development.

Part 1: Internet of Things Development Platform

Exercise 1: IoT Data Structure, Database, and API

Deliverables:

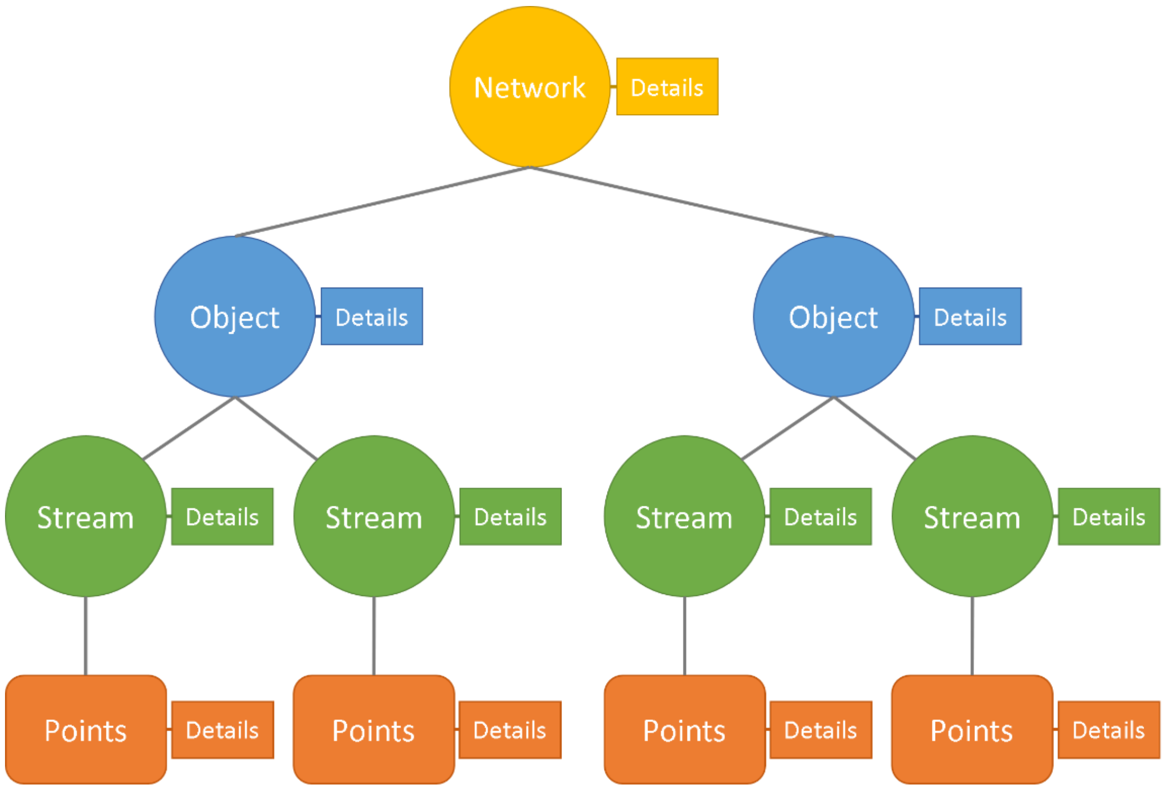
- None

Directions:

1. For this course, we will employ Wallflower.Pico (https://github.com/wallflowercc/wallflower-pico), a very basic Internet of Things platform with a RESTful Web API for storing and retrieving time series data (int, float, or string). This platform was originally developed for the CE 186 class and is intended to make it easy to collect data with Python and to create visualizations with HTML/CSS/JS. The Wallflower platform abstracts IoT devices using the data structure shown in the image below.

   With this data structure, devices (such as a thermostat, a light switch, or a watch) are referred to as objects. Information about a particular object, such as a user-friendly name, a device category, or a product description, is collectively referred to as the object details. These details provide the information necessary for other objects to understand and interact with the object.

Every object is comprised of a number of streams. For example, a thermostat (object) has a setpoint temperature (stream) and turns on and off (stream) a heating system according to a temperature measurement (stream). Essentially, streams describe the data that is sent to and from an object. This includes sensory data as well as user settings and commands. Just as with objects, information about a particular stream is collectively referred to as the stream details.



The actual values of a stream are referred to as the points. The points can include a historical record of all previous values or simply the current value of the stream. The points details provide information directly related to the values being stored, such as the data type (integer, string, etc.) and units (Celsius, hours, etc.).

Lastly, a network is simply a population of objects. Generally, a network will represent all the objects capable of communicating with one another (for example, the WiFi network within a house). The network details include characteristics of the network.

The Wallflower.Pico server employs a very basic implementation of the data structure above. Specifically, we are limited to a single network with the id 'local', the only object and stream details we can change are the names, and the only points details we can set is the data type.

2. In this exercise, we will generate data to store in the database. To begin, download the Pico server from https://github.com/wallflowercc/wallflower-pico and place the files in your Lab 5

directory. Open and run the **`wallflower_pico_server.py`** code. This will start a Flask server with a SQLite database.

3. Open http://127.0.0.1:5000/ in a browser to view the web interface.
4. Using the web interface, create an object with the id ‘`obj-lab5`’ and the name ‘`Lab 5 Test Object`’. When you click **Create**, you will see the **`HTTP PUT`** request that was sent from your browser to the server running on your computer and the JSON text that was returned by the server.
5. Next, add a stream to the ‘`obj-lab5`’ object with the id ‘`stm-lab5`’, the name ‘`Lab 5 Test Stream`’, and the point data type ‘Float’. Again, when you click **Create**, you will see the HTTP PUT request that was sent to the server and the JSON response.
6. We have now added an object and stream using the HTTP API. By clicking View in the left sidebar of the web interface, you will find that the object and stream have been added to the dropdown menu. The HTTP API consists of network, object, stream, and points endpoints which can be called using HTTP **`GET`**, **`PUT`**, **`POST`**, and **`DELETE`** requests. The endpoints of the API consist of `network_id`, `object_id`, and `stream_id` URL variables, as given below.

Network Endpoint: /network/<network_id>
Object Endpoint: /network/<network_id>/object/<object_id>
Stream Endpoint: /network/<network_id>/object/<object_id>/stream/<stream_id>
Points Endpoint: /network/<network_id>/object/<object_id>/stream/<stream_id>/points

7. Navigate to the Update page of the web interface and add several random points to the ‘`stm-lab5`’ stream. Note that points are added to a particular stream using a HTTP **`POST`** request to the points endpoint with the value and (optional) timestamp included in the query string. If a timestamp is not included in the HTTP request, the timestamp will be added by the server.
8. Navigate to the View page for the ‘`stm-lab5`’ stream. Use the Search Points form at the bottom of the page to send a HTTP GET request to the points endpoint. The query string can include a start time (ISO 8601 timestamp), end time (ISO 8601 timestamp), and a limit on the number of data points to return.
9. In a second Python console, open and run the **`wallflower_demo.py`** code that was included with the Pico server. This code uses the Python requests module to make HTTP calls to the API. The code will create an object and stream and then update the stream with random data. Use Ctrl+C to stop the code. Make sure you understand how the Python code implements the HTTP API. You will need to use the API in the next few exercises.

Exercise 2: Input Form

Deliverables:

- None (the code for exercise 2 will be included in the files submitted for exercise 3)

Directions:

1. In this exercise, we will recreate the NameServer from Lab 4 using the Pico server. Specifically, we will use JavaScript and jQuery to add a form to the web interface and allow users to enter their first and last names. When the form is submitted, the contents of the form will be sent to the database and stored as a JSON string.

This exercise will also cover some JS and jQuery basics. For a more extensive introduction to JS and jQuery, we recommend the resources listed above.

2. We will be extending the web interface of the Pico server by loading additional HTML using JS. To begin, create from scratch a file called **extend_dashboard_links.html** in the static directory of the Pico server. Edit the file to add the following HTML code.

```
        <li>
          <a href="#" id="sidebar-input" class="first-level"><i class="fa
fa-tags fa-fw"></i> Input</a>
        </li>
```

   The `<li>` element tag denotes an item in a list and the `<a>` tag denotes a link. Note that the link element has an id of "`sidebar-input`". The JS code running on the web interface uses this id to determine which page to display when a sidebar link is clicked. For example, the View link has an id of "sidebar-view".

3. With the Pico server running, navigate to http://127.0.0.1:5000/static/extend_dashboard_links.html in your browser. You should see a very simple webpage with a single link. Note that any files that are placed in the static directory are being hosted by the Flask app on which the Pico server is built.

4. Next, we will edit the **extend_dashboard.js** file in the static/js directory to load the **extend_dashboard_links.html** file into the web interface. Add the following code to the **extend_dashboard.js** file.

```
var ul = $('ul#side-menu');
$.ajax({
  url : '/static/extend_dashboard_links.html',
  type: "get",
  success : function(response){
    console.log("Load /static/extend_dashboard_links.html");
    ul.append(response);
  }
});
```

   In the code above, `$('ul#side-menu')` calls the jQuery selector function $( ) to select the `<ul>` element with the id of "`side-menu`". Since there is only one element in the HTML page that fits this description, the variable ul therefore refers to the sidebar of the web interface.

   The `$.ajax()` method is comparable to the Python requests module and allows us to make HTTP requests from the web browser (however, for security reasons, the `$.ajax()` method assumes you are making HTTP requests to the same domain as the webpage). In the code above, we pass the url of the HTML file and set the HTTP request type to **`GET`**. Lastly, since JS is an asynchronous language, we also pass in a function that the `$.ajax()` method will call if the HTML file is successfully received. The contents of the HTML file are then appended to the sidebar.

Go back to the Wallflower.Pico home page: http://127.0.0.1:5000/. Refresh this page (you may need to *force* refresh). Open your browser's JS console and verify that you see the "Load …" message. On the webpage itself, you should see a new sidebar menu item "Input". Do you see it?

HINT: How to open your browser's JS console?

- On Chrome
  - MAC: CMD + OPT + J
  - PC: CTRL + SHIFT + J
- On Firefox
  - MAC: CMD + SHIFT + J
  - PC: CTRL + SHIFT + J
- On Safari
  - MAC: CMD + OPT + C

5. Next, create a file called `extend_dashboard_pages.html` in the static directory and add the HTML code below.

```
  <div class="page hidden" id="page-input">
      <div id="page-wrapper">
          <div class="row">
              <div class="col-lg-12">
                  <h1 class="page-header">Input</h1>
              </div>
              <!-- /.col-lg-12 -->
          </div>
          <!-- /.row -->
          <div class="row">
              <div id="content-input" class="col-lg-8">


                <form id="form-input">
                    <fieldset class="form-group">
                        <label for="first-name">First Name</label>
                        <input type="text" class="form-control"
name="first-name" placeholder="" required="true">
                    </fieldset>
                    <fieldset class="form-group">
                        <label for="last-name">Last Name</label>
                        <input type="text" class="form-control"
name="last-name" placeholder="" required="true">
                    </fieldset>
                    <button type="submit" class="btn">Submit</button>
                </form>


              </div>
              <!-- /.col-lg-8 -->
              <div class="col-lg-2 note"></div>
              <!-- /.col-lg-2 -->
          </div>
          <!-- /.row -->
```

```
            </div>
            <!-- /#page-wrapper -->
        </div>
        <!-- /#page-input -->
```

The HTML above includes the basic structure shared by all the pages in the web interface as well as a form included within the “`content-input`” div. Note that the outermost div element has an id of “`page-input`”. This id allows the web interface to know which content to show or hide based on which sidebar link was clicked.

6. Add the following JS code to the extend_dashboard.js file to load the HTML into the web interface.

```
var wrapper = $('div#wrapper');
$.ajax({
  url : '/static/extend_dashboard_pages.html',
  type: "get",
  success : function(response){
    console.log("Load /static/extend_dashboard_pages.html");
    wrapper.append(response);

    // Form submit call goes here.
    $("form#form-input").submit( onInputFormSubmit );
  }
});
```

Refresh the Wallflower.Pico homepage. The web interface should now load with the additional Input link and page. Note that when the form is submitted, the JS code will call the `onInputFormSubmit` function, which we have not yet defined.

7. Now that we’ve added an Input page to add names from LAB4, let’s create an object and stream for “names”. Use the web interface to create an object with the id “`obj-names`” and object name “name”. Create a stream with the id “`stm-form-input`” and data type “String”. We will use this stream to store the user input collected by the form.
8. In this step, we will add some JS code that will run when the Input web interface form is submitted. First, define the `onInputFormSubmit` function and add to **`extend_dashboard.js`**.

```
/*
  Add functionality to the input page form
*/
function onInputFormSubmit(e){
  e.preventDefault();

  var object_id = "obj-names";
  var stream_id = "stm-form-input";


};
```

Within the `onInputFormSubmit` function, add the following code to collect the data in the input fields and store the text as key-value pairs in a JSON object (similar to a Python dictionary). Also, log the data to the JS console.

```
// Gather the data
// and remove any undefined keys
var data = {};
$('input',this).each( function(i, v){
  var input = $(v);
  data[input.attr("name")] = input.val();
});
delete data["undefined"];

console.log( data );
```

Refresh the WallFlower.Pico homepage in your browser. Go to the Input page. Open your browser's JS console. Input a First Name and Last Name into the form, e.g. "Harry" and "Potter". Verify in the JS console that when the form is submitted, the data is collected and stored as a JSON object. HINT: you should see text in JSON form that looks like…

```
{first-name: "Harry", last-name: "Potter}
```

As of now, we have not stored this data into the Wallflower.Pico database. Let's now add this functionality.

Finally, within the `onInputFormSubmit` function, create the points update query and use $.ajax() to make a HTTP POST request to the Pico server. The form data is converted to a JSON string and set as the "points-value". Add the following JS code to the end of your `onInputFormSubmit` function.

```
var url = '/networks/'+network_id+'/objects/';
url = url + object_id+'/streams/'+stream_id+'/points';
var query = {
  "points-value": JSON.stringify( data )
};

// Send the request to the Pico server
$.ajax({
  url : url+'?'+$.param(query),
  type: "post",
  success : function(response){
    var this_form = $("form#form-input");

    if( response['points-code'] == 200 ){
      console.log("Success");
      // Clear the form
      this_form.trigger("reset");
    }
    // Log the response to the console
```

```
        console.log(response);
      },
      error : function(jqXHR, textStatus, errorThrown){
        // Do nothing
      }
    });
```

Refresh the Wallflower.Pico web interface. Go to the Input page. Open your browser's JS console. Input "Harry" and "Potter" as before. View the response that is returned by the Pico server. Using JSON and the Pico server, we are now able to store and retrieve input data. Refresh the page and navigate to the View page for the "`stm-form-input`" stream to see a record of past inputs. If everything works properly, then the response will contain a point that includes the value:

```
"{\"first-name\":\"Harry\",\"last-name\":\"Potter\"}"
```

Exercise 3: Simple Report

Deliverables (4 pts):

- Submit the extend_dashboard_links.html, extend_dashboard_pages.html, and extend_dashboard.js files. Additionally, in your lab report, copy & paste the code into code boxes.

Directions:

1. If you have not already done so, open and run the wallflower_demo.py code that was included with the Pico server. This code employs the HTTP API to create an object and stream and then updates the stream with random data. Use Ctrl+C to stop the code.
2. Modify **`extend_dashboard_links.html`** to add another link called "Report". We will not give you code to copy & paste directly. You can emulate our past steps. Do not forget to change the id of `<a>` to "sidebar-report". Also, visit http://fontawesome.io/icons/ and choose a different icon (replace "fa-tags" with "fa-<new icon>" in <i>).
3. Modify **`extend_dashboard_pages.html`** to add another page for the link "Report". Begin by copying and pasting the HTML that we added in the previous exercise. Next, replace "page-input" and "content-input" with "page-report" and "content-report", respectively. Lastly, replace "Input" with "Report" in the <h1> element and delete the entire <form>…</form> element.
4. If you refresh the web interface, it should now include a link for Report which loads a (mostly empty) page entitled "Report". In the rest of this exercise, we will walk through the steps of adding a very basic plot to the Report page to display data generated by the **`wallflower_demo.py`** code. That is, we are going to web visualize our data!
5. Add the following JS function for requesting points from a particular stream to the **`extend_dashboard.js`** file. Note that we are now using `$.ajax()` to make requests to the server's API. Also, the inputs of the `getPoints` function includes a parameter named

`callback`. The `callback` parameter is itself a function which we call if data is successfully received from the server.

```
/*
  Add function to get points for report page
*/
function getPoints( the_network_id, the_object_id, the_stream_id, callback ){
  var query_data = {};
  var query_string = '?'+$.param(query_data);
  var url = '/networks/'+the_network_id+'/objects/'+the_object_id;
  url += '/streams/'+the_stream_id+'/points'+query_string;

  // Send the request to the server
  $.ajax({
    url : url,
    type: "get",
    success : function(response){
      console.log( response );

      if( response['points-code'] == 200 ){
        var num_points = response.points.length
        var most_recent_value = response.points[0].value
        console.log("Most recent value: "+most_recent_value);
        console.log("Number of points retrieved: "+num_points);
        callback( response.points );
      }
    },
    error : function(jqXHR, textStatus, errorThrown){
      console.log(jqXHR);
    }
  });
}
```

6. Next, we will add some code so that `getPoints` is called each time the Report link is clicked. The JS code included in the interface allows us to define a callback function that will be called each time any of the sidebar links are clicked. Add the following code to **`extend_dashboard.js`** to check if the Input or Report links were clicked. If Report was clicked, the `getPoints` function is called. If the points request is successful, the callback function is called.

```
// Call getPoints if Input or Report is selected
custom_sidebar_link_callback = function( select ){

  if (select == 'input') {

  }
  else if (select == 'report'){
    getPoints('local','test-object','test-stream', function(points){
      console.log( "The points request was successful!" );
    });
  }
}
```

Refresh your browser, open the JS console, and click on the Report link to verify that everything is working. In the JS console, you should see some feedback about a successful execution of the points request.

7. Next, we will add code that creates a plot using Highcharts and inserts it into the page. The plot uses a minimal set of options, defined by the `report_plot_options` variable. The full set of options for customizing the behavior of the plot can be found at http://api.highcharts.com/highcharts/chart.

```
/*
    Function to plot data points using Highcharts
*/
function loadPlot( points ){
  var plot = $('#content-report');
  // Check if plot has a Highcharts element
  if( plot.highcharts() === undefined ){
    // Create a Highcharts element
    plot.highcharts( report_plot_options );
  }

  // Iterate over points to place in Highcharts format
  var datapoints = [];
  for ( var i = 0; i < points.length; i++){
    var at_date = new Date(points[i].at);
    var at = at_date.getTime() - at_date.getTimezoneOffset()*60*1000;
    datapoints.unshift( [ at, points[i].value] );
  }

  // Update Highcharts plot
  if( plot.highcharts().series.length > 0 ){
    plot.highcharts().series[0].setData( datapoints );
  }else{
    plot.highcharts().addSeries({
      name: "Series Name Here",
      data: datapoints
    });
  }
}

var report_plot_options = {
    chart: {
        type: 'spline'
    },
    xAxis: {
      type: 'datetime',
      dateTimeLabelFormats: { // don't display the dummy year
          month: '%e. %b',
          year: '%b'
      },
    },
};
```

8. Finally, add `loadPlot( points );` to the `getPoints` callback function (just below `console.log( "The points request was successful!" );`). Refresh your browser and verify that the Report page now contains a plot.
9. To visualize time series data, we often want a plot to update as new points become available. This can be done by using `setInterval` to call the `getPoints` function every second. Update the `custom_sidebar_link_callback` function with the code below. Note that the timer is set to stop after 20 calls (i.e. after 20 seconds) and will reset each time Report is clicked.

   Refresh your browser and go to the Report page. In the JS console you should see periodic feedback from the `getPoints` request. Now run the **`wallflower_demo.py`** code to see the plot dynamically update.
   Note: If your browser is open to http://localhost:5000/, use http://127.0.0.1:5000/ instead.

```
// Call getPoints if Input or Report is selected
// ...added feature to dynamically update plot as new data becomes available
custom_sidebar_link_callback = function( select ){

  if (select == 'input') {

  }
  else if (select == 'report'){
    var plotCalls = 0;
    var plotTimer = setInterval( function(){
      getPoints('local','test-object','test-stream', function(points){
        console.log( "The points request was successful!" );
        loadPlot( points );
      });
      if( plotCalls > 20 ){
        console.log( 'Clear timer' );
        clearInterval( plotTimer );
      }else{
        plotCalls += 1;
      }
    }, 1000);
  }
}
```

TIP: You may see a huge time gap in the dynamic plot, between the first and current run of **`wallflower_demo.py`**. You may delete the `test-object’ using the web interface, and then re-run **`wallflower_demo.py`**. Now the plot will dynamically update with fresh data. If all works well, then you should see a very cool dynamic data visualization.

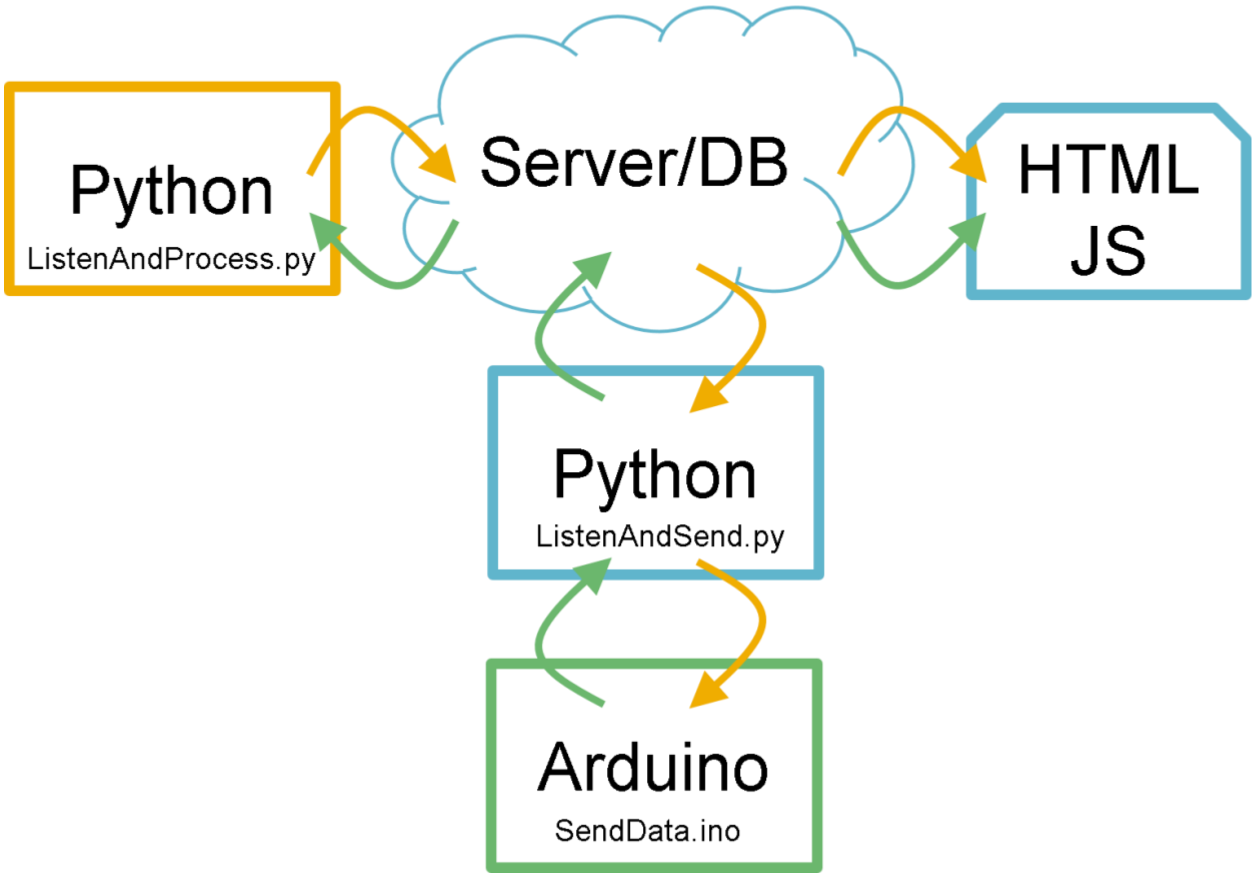
Part 2: Cyber-Physical System

Exercise 4: Putting it all together

Deliverables (4 pts):

- Submit all necessary files for implemented the cyber-physical system. Additionally, in your lab report, copy & paste code that you have written and/or modified into code boxes.

Directions:



Now it will all come together! Your hard work in labs 1 through 5 will culminate in this exercise, where you integrate everything. By the end, you will construct the first skeletal prototype of your CPS project! This will be amazing, you will see!

1. In this exercise, your assignment is to create a version of the cyber-physical system illustrated above. The colored arrows roughly illustrate the flow of data in the system. You are free to decide what your system does but you must meet the following requirements.
   a. Write an Arduino program called `SendData.ino` which produces at least 2 data streams (collectively represented in green). These data streams might be analog sensor

readings, digital switch/button input, a sine wave, etc. These data streams should be sent to a computer once every 10 seconds using a serial connection.
The Arduino program must also listen for at least 1 data stream (collectively represented in orange) and perform some actuation, such as toggling or fading an LED.

b. Write a Python program called `ListenAndSend.py`. This program listens to the serial port connected to the Arduino and reports the stream values (green) to the server/database (i.e. Pico server) using HTTP POST requests to the server's Web API. The program also listens to at least 1 data stream (orange) by making HTTP **GET** requests to the API once every 5 seconds and sends the values to the Arduino.

   Hint: You may find the `ProgramStructure.py` code helpful to get started.

   Hint: You will need to check the PySerial API Docs to figure out how to send data to the serial port (https://pythonhosted.org/pyserial/pyserial_api.html). Pay attention to how Python needs to encode data that is sent to the serial port.

   Hint: Be careful with data types.

c. Write a Python program called `ListenAndProcess.py`. This program listens to the Arduino streams (green) by making HTTP **GET** requests to the server's Web API once every 5 seconds. If the streams have updated since the last HTTP request, the program should process the values in some way (e.g. sum, product, mean, XOR, etc.) to produce at least 1 data stream (orange). This data stream is reported to the server/database using HTTP **POST** requests.

   Hint: You may find the `ProgramStructure.py` code helpful to get started.

d. Finally, you should visualize the data streams using an HTML/JS webpage. You may reuse the code from the previous exercise, but you must customize the user interface to fit your cyber-physical system (e.g add a table detailing the min/max/mean values observed, or a histogram to illustrate the distribution of the data points). At a minimum, you should change the Highchart plot options by adding a title, axis labels, etc.

   See http://api.highcharts.com/highcharts/chart for available options.