Helpful Tips

Objectives:

- Establish good developer/debugging habits

As you proceed with the rest of the AJAX assignments, keep the following tips in mind:

. **Put all the key information in a <form>.** For any AJAX request done internally (meaning, to your own server), you're almost always sending several pieces of information to another url. Therefore, instead of having your data scattered across your web application in different elements, have it all contained in a form. This will save you countless hours of debugging in the future.

. **Make the <form> submit manually without AJAX first and make sure the HTTP response is valid and generates no errors.**Debugging AJAX can be tricky, so break the process into smaller pieces. First, make sure the HTTP request generates the result you want. In a lot of cases, for internal AJAX calls, the HTTP response will be in the form of html (although sometimes you may want to pass back JSON, which we will discuss later). Make sure you are able to generate the desired response before adding AJAX to your list of concerns.

. **Carefully consider whether you should use GET or POST.** You already know how to post data to your server. However, if you use a GET request, the data will be in the url as a **query string**. Query strings are easily recognized by the question mark in the url. Different key-value pairs are separated by ampersands. For example, if we use a get request to submit a form to /users with two inputs, name and color, we may produce the url `localhost:5000/users?name=kermit&color=green` . On the server side, our logic for the /users route will pull the information out of the url with the `request.args.get(key)` method.

```
@app.route('/users')
def data_from_query_string():
    print(request.args.get('name'))     # outputs kermit
    print(request.args.get('color'))    # outputs green
    # any other logic goes here
```

. **Once the HTTP request is working the way you expect, use the `.ajax()` method to have the JavaScript interpreter send that request.**

1. Use `.serialize()` to group all the key information in the form.
2. Since the form should also contain the url where it should be submitted, your AJAX call should often follow the format shown below. If your AJAX call doesn't look simple like this, you're not doing AJAX the right way (in our opinion).

```
$('#form1').submit(function(){
  $.ajax({
    method: "POST",     // using a GET request would put your form data in the url as query
 strings
    url: $(this).attr('action'), // 'this' refers to #form1, the element that triggered the
 function
    data: $(this).serialize()
  })
  .done(function(response) {
    // your code on what to do once the http response is received
  })
  .fail(function(response) {
    // optional code on what to do if the http request fails
  })
  .always(function(data){
    // optional code on what should be done regardless of whether the http request is succes
sful or not
  })
  return false; // return false so the form is not submitted normally
});
```

. **Use jQuery to update part of the HTML DOM using the `.html()` method.** This is straight forward if the response consists of HTML. If the http response returns JSON, you will most likely need to create a new HTML string from the data (using loops, if/else statements, etc).

. **Use your browser's (Google Chrome, of course) inspector heavily.** The very first thing to check when debugging JavaScript is the console tab. Any errors in your JavaScript syntax will be displayed in the console, so if you read them, you may save hours of time hunting for a missing curly bracket. To produce your own messages in the console, remember to use `console.log()` and use it frequently! It is easy to lose track of the order of operations when working with asynchronous code. To get started, be sure to include logging to the console in the following places:

1. First thing in `document.ready()` to make sure jQuery is loading properly
2. Right after the `.submit()` handler for your form to be sure the event listener is working
3. Within the `.done()`, `.fail()`, and `.always()` methods chained after `.ajax()`
4. Whenever creating variables within your JavaScript code to make sure you have what you think you have. Check the data type as well. Often we think we have a number when we actually have a string, or we think we have an object when we actually have null.

. **Check the network tab.** The network tab in your browser's inspector allows us to monitor the http requests and responses sent and received by our web page. Make sure your page is connecting to the file where you are submitting your form, and make sure there are no errors. Click on the response/preview tab to check whether the URL is returning data. If your JavaScript/jQuery code is working correctly, but your Python code is awry, using the network tab will allow you to see your Python errors without turning AJAX off. This is a great way to lessen the scope of potential errors!

. **Understanding AJAX is essential to master frontend frameworks.** Later on, in the MEAN stack, we will be covering Angular. Many students also choose to learn React, AngularJS, and/or Vue.js. Developers skilled with frontend frameworks are in high demand, and frontend frameworks revolve around asynchronous programming.