JavaScript-PHP Boomerang

When developing a webpage, it is essential to have the front-end (the website) communicate with the back-end (the server). This demo describes one way to take in user interactions and then update the page accordingly.

I call this functionality the JavaScript-PHP Boomerang based on the sending and returning of the data. I put this simple, interactive demonstration together to explain how this works.

BACKGROUND

On this website, the code is divided into different modules, each focusing on different functional areas: content, communications, error tracking, writing tools, etc.. Each code module consists of three files: one JavaScript file, and two PHP files. The JavaScript file handles UI interaction and content updates. One of the PHP files works like a switchboard operator, while the other one is a library of functions.

The functionality is how a majority of my website works using only JavaScript and PHP. There are other frameworks, especially with NodeJS to handle this type of functionality, but this method has worked for me on a web host that does not support frameworks like NodeJS.

This demo does requires the JQuery CDN:
https://releases.jquery.com/

HOW THE BOOMERANG WORKS

This process is not a shallow request and response. The request travels further, is processed, and then returns home. Like a boomerang, data is thrown to the back-end, where it is fiddled with and formatted, and then returns to the user.

First, the UI awaits an action, either using JavaScript listeners or inline actions, like OnClick. This action collects any necessary data from the UI (in this case, using JQuery), bundles it with an action, and sends the data to the first PHP file using an AJAX call.

The PHP operator starts with declaring the variables. The data sent from AJAX is converted into variables. For all other data not included with the request, these variables are assigned a default value. One variable holds the desired action. A switch statement uses this action variable to call a function in the PHP library.

Once the function returns a reply, the operator echos this response back to AJAX waiting in the JavaScript file for a response. Usually, the response triggers an update to the UI letting the user know the boomerang has returned home.

See Figure 1 below for a simplified visualization of the the data flow.

HOW THE DEMO WORKS

This demo includes two interactive UI elements, an element for the response, and a header. The interactive elements are a button labeled “Press Me” and a checkbox labeled “Reverse?”

The button is assigned one JavaScript function with the onClick listener. In this function, the value of the checkbox is determined using a JQuery finder by the element’s ID. This value and an action to request for a new phrase are sent to the PHP operator. Two variables are set to the incoming parameter values, while all others are set to their default values. One of the variables is used in the switch statement. The case in this statement requests a new phrase with a flag set to have the response reversed. In the PHP library, a response is randomized and reversed, then returned to the PHP operator. All switch cases set a feedback variable, which is returned to the AJAX-JavaScript call via echo statement. Upon the successful return to the JavaScript file, the result is stored in the HTML value of the response element.

The checkbox is assigned a similar function using the onChange listener. In this function, both the response string and the value of the checkbox are sent to the PHP operator with a different action. Three variables take in the incoming parameter values, while any others are set to their default values. This time, the switch statement triggers a case which only uses the string reversing function. The result is returned to the operator, where it is echoed back to the AJAX-JavaScript call, and the response element is updated.

In the code of this demo, the switch statement includes a default case which responds with a message indicating the operator does not know what to do with the desired action. Also, an extra case statement has been included, but will not be triggered until its action is sent to the operator. Similar to this third, unused case, additional variables test if their parameters have been sent, and when they are not, its associated variable is set to a default value.

Here is the link to try this demonstration:
https://www.checkeredscissors.com/pages/portfolio/web/php/demo1/demo.php

OTHER CONSIDERATION(S)

One improvement to consider for this functionality is to improve the AJAX calls with possible failure scenarios. A majority of the AJAX calls in my code request content updates. Most failure scenarios are logged by the backend and alternate, generic responses are echoed back to the AJAX call. To improve this, instead of echoing back an HTML formatted response, this value should be packed into a JSON encoded response with an appropriate status. Because I have been coding this way for quite a while without including failure scenarios, this is an epic sized enhancement to make to the code. However, I have filed myself a ticket to make this and other best practice improvements to the code.

Another consideration is to update the website’s hosting package to support frameworks that can perform similar UI updates in improved ways. During the bootcamp, I enjoyed playing with React and other NodeJS frameworks. At the moment, the cost of making such a hosting change is not feasible.

One last consideration is to bypass the PHP operator with individual PHP files containing the narrower selection of parameter requestors and functionality. If I recall correctly, I decided not to go this route, because maintaining two PHP files seemed simpler than many PHP files. There is quite a bit of shared functionality in the PHP libraries, which would still need to be included. It boils down to the different between one-to-one versus many-to-one.

DATA FLOW VISUALIZATION

Figure 1 - Data flow visualization: