Datava 4: WebSockets

Asa Farrer, Kyle Hughes, Henry Purdum, Hunter Sitki

Final Report
Field Session Summer 2021

I. Introduction

Datava Inc. is an SaaS business analytics startup, founded in 2006. Although they also specialize in employer training, our group was tasked with assisting the client with the back end of their data management and analysis system. During preliminary discussions, our client David introduced us to the WebSockets (WS) framework - which allows for a persistent connection between a server and browser, once a connection has been made. Our team was tasked with updating the existing HTTP and Ajax (Asynchronous JavaScript and XML) request system that exists within the Datava system with a WebSockets connection. Our group was given the option of using any WebSocket API available to us; though, it was recommended that we use Swoole or Ratchet, depending on our group's confidence with the PHP and JavaScript languages.

As two of the more experienced coders on the team were comfortable with the networking environment, our group opted to use the Ratchet library, as it was an intermediate-to-advanced level introduction to the WebSocket framework. Datava's product vision included:

- Replacing all Ajax-type connections with WebSockets
- SSL secure encryptions for all new connections
- The ability to 'push-down' information from the server to a user's browser

These were the stretch-goals that came from the client during our first week of discussion - as our group became more aware that not all of the team members were capable of writing code for the project, the project goals were narrowed down to installing a WebSocket library onto the server, and ensuring that all existing Ajax requests could be replaced with the new software.

Our group was able to meet these two basic requirements, more as a proof of concept than a working product. Our connections are opened without closing, and created every time an Ajax request would be made. This is not the optimal implementation for the WebSockets framework, which is designed to remain open as long as the user is connected to the server, to push/pull information based on user events or server response. Fortunately, the code is written in a way that implementing SSL should be an easy adjustment; however, persistent connectivity will require more work to include. This is because implementing persistent connections will require a function to be added that opens a WS as the user connects to the system, then the current override will need to be changed to only send/receive information to and from the user.

II. Requirements:

This project consists of several requirements to be met to deliver a good product. The first of these is to replace the existing Ajax and HTTP communication of the client's ESP with WebSockets to enable two-way communication. We need to be able to push the WebSocket information to and from the server without having to make a new request each time. This framework should also allow for future use as instant client-to-client communication. This project also requires that tests be run to ensure that the updates we are making do not break the client's existing code or cause other issues for the existing codebase.

The point of this backend overhaul project is to make communication more efficient and provide a future route for implementing two-way communication. One implementation requirement was to use either the Ratchet or Swoole frameworks to implement WebSockets for the Datava system. Any back-end functionality should be done using PHP and most front-end functionality should be using JavaScript, especially the ExtJS framework. There should also be a measurable performance improvement from the former system to the WebSockets system.

III. System Architecture:

A. Summary

The overall objective of this project was to implement a system to update and potentially optimize the way that the Datava web servers interact with clients' browsers. With this in mind, our project focused almost exclusively on the back-end components of the pre-existing application. This is not to say that no interaction or modification of front-end aspects was required, which it was. Instead, the majority of the front-end modifications were made away from the actual user interface and instead focused on the ExtJS implementation of Ajax requests. By focusing on and modifying this one class' behavior, we were able to essentially modify the behavior of the entire existing application while only needing to create and maintain front-end code in a single file.

B. Back-end and WebSockets

WebSockets provide client applications and browsers the ability to make a single request to a running server to establish a connection. Once this connection is established, the connection remains open and information can be freely pushed back and forth between the server and client without the need for extra requests or acknowledgements to be sent. In general, WebSockets are often used for applications that require or would benefit from near instantaneous communication with a server, such as web-based games or real time messaging, due to the fact that WebSockets can send and receive messages much faster than other methods of network communication.

The client's main objective for us was to change the existing Ajax request system, which simply specifies some parameters and a URL, which itself corresponds to a file within the application. The Ajax request would send the parameters, as well as a plethora of other meta information, to the specified file where a response is generated based on the inputs and echoed back to the Ajax request in the form of a JSON object. When a response is received, the Ajax request then executes a callback success function that does something meaningful, such as updating the UI or validating the user's session, with the response.

To mimic this kind of behavior with WebSockets, we needed to create a system that could interpret the parameters that would be passed through an Ajax request. The Ratchet WebSocket API that our team utilized requires two main components to function in the way that we required:

1) MessageComponentInterface

Our group referred to the MessageComponentInterface as the “socket” in WebSockets. This socket is a class that specifies the behavior of the server when a new connection is established, a message is received, a connection is terminated, or an error with a connection occurs. Each of those situations must be explicitly handled in their own corresponding function, such as onMessage(). For our implementation, a very generic implementation sufficed, save for the handling of messages from clients. In our onMessage() function, messages will come in the form of a URL-encoded string. Using built in PHP functions, this message is parsed, then converted into a PHP array. After the array is converted, it is passed to the file that is specified by the message and that file is dispatched. Once the file has concluded its execution, it will echo a response - in the form of an encoded JSON object - to stdout and the onMessage function will capture that output and send it back to the requesting client.

2) WebSocket Server

The WebSocket server provides the overall network functionality for WebSockets. A WebSocket server is not the same as the web server that the application interacts with… the difference being that the WebSocket server runs on the web server, and occupies a thread continuously. This is all done as the WebSocket is waiting for messages and handling connections. In production, this would look like a daemon running on a background thread on the web server. The construction of such a server is relatively simple, it only requires passing a MessageComponentInterface and IP address and port values into its constructor. From there, the server will run indefinitely until it encounters an error or is terminated manually.

C. Front-end and Ajax

In order to comprehensively modify the behavior of Ajax requests throughout the application, we chose to override the Ext.Ajax class, which is the ExtJS specific implementation of Ajax. Within this override, we were able to modify the behavior of the request function - this is the function that is used throughout the entire application.

To utilize the WebSocket interface that is running on the back-end, we used the built-in WebSocket functionality that JavaScript provides. This framework allows us to create a WebSocket object and attempt to connect to a running WebSocket server port. Once the connection between the front-end WebSocket object and the back-end server is established, messages can be sent back and forth between each other. This behavior is modelled in **Figure 1**, found below.

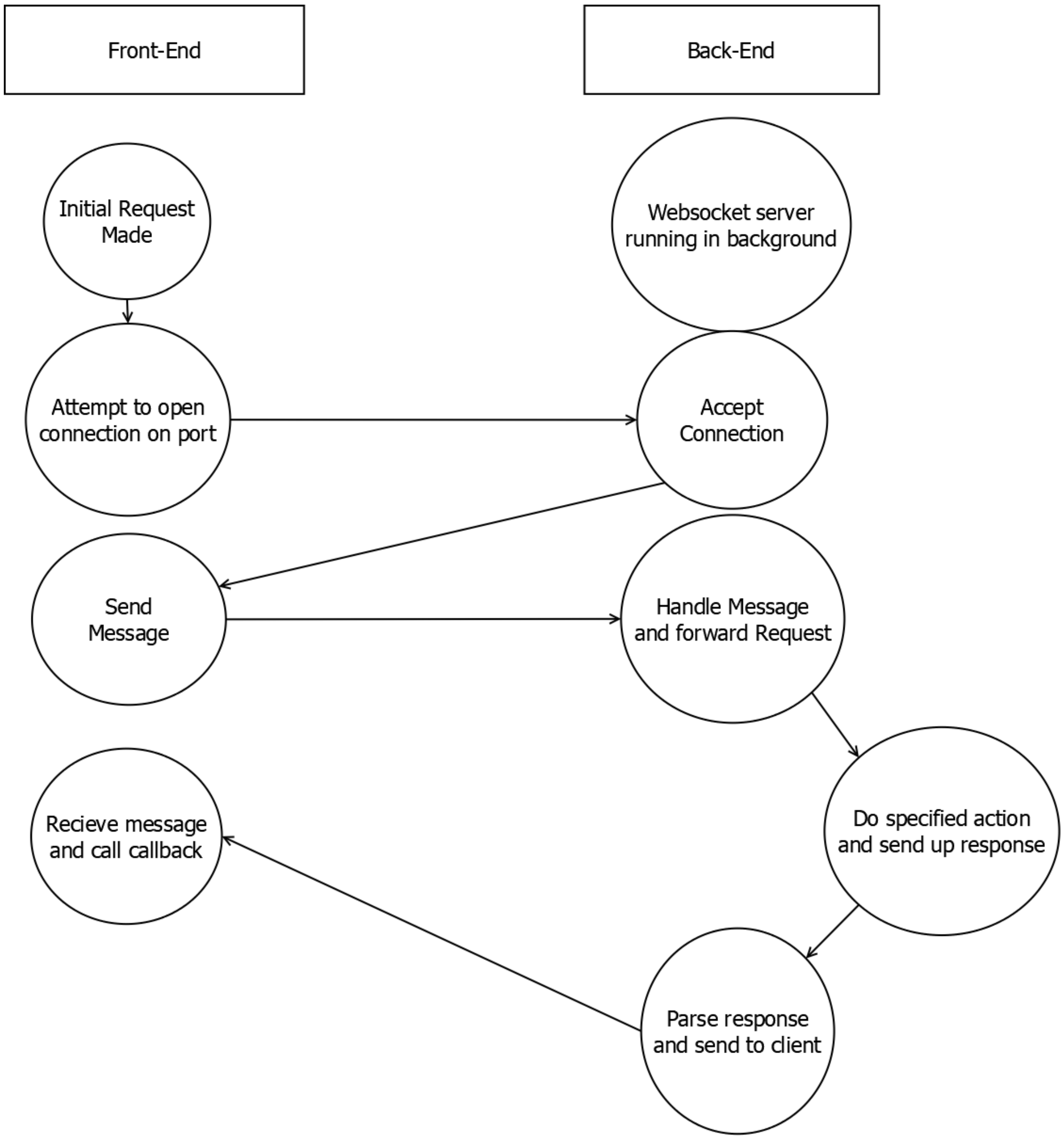
Front-End
Back-End
Initial Request Made
Websocket server running in background
Attempt to open connection on port
Accept Connection
Send Message
Handle Message and forward Request
Do specified action and send up response
Recieve message and call callback
Parse response and send to client

Figure 1: Simplified interaction between front and back ends.

IV. Technical Design:

A. JS Overrides

Development of our server application did not require immediately creating an extensive arrangement of files or frameworks. Instead, we were able to use some of the existing files as a test area for implementing a single WebSocket connection. As a result, we spent a lot of time connecting to that server from a few select lines in a particular JavaScript file.

As we got further into our development process it became clear that simply making a connection request to replace every existing Ajax related functionality would be extremely inefficient and time consuming. If our client had only a few applications requiring WebSockets, we could have hard coded each connection request. However the more modular practice of adding a functionality override for the Ajax requests that were already implemented proved the more elegant and efficient solution. Relocating this override functionality to it's own unique location of the development servers means that our presentation of our product to our client can be more streamlined and better organised. An example relative path from our main development directory on the server to our override implementation would be "./js/overrides/Ajax.js" where our implementation is stored in this Ajax.js script.

This implementation covers functionality for handling requests, responses, and sending/receiving messages with our WebSockets. Requests are handled by checking if the configuration parameter of said request defines a WebSocket or not. If it does, we create a WebSocket if one is not already open. We then set a few more configuration parameters, set the functionality of the WebSockets onmessage function and then send the message. Some of the Ajax functionality can be seen in **Figure 2**.

Responses from the web server are handled by converting the response text into a JSON object. This object is then passed into callback functions that are specified with the configuration parameter passed into the request. The standard Ajax functionality had default error and success callback functions so it was necessary to execute those functions if the configuration did explicitly specify any. In general, sending a message from the frontend is as simple as calling "websocket.send(message)". Our receiving end of the functionality requires a bit more complexity; we need to decode the JSON object that our message has been converted to mid-flight. Otherwise, functionality between the two parts of the system is fairly similar and straightforward.

B. Functionality of onMessage

Figure 3 does a great job of demonstrating the Finite State Automata at work behind our program. We start in the center of the diagram, awaiting connections and requests to send messages. When a new client tries to connect, we should assign them a unique ID and then establish the connection. When a message is received we should parse the URL for parameters, reroute those parameters to a global variable or outside file, await a response and then send an encoded JSON response containing the necessary parameters.

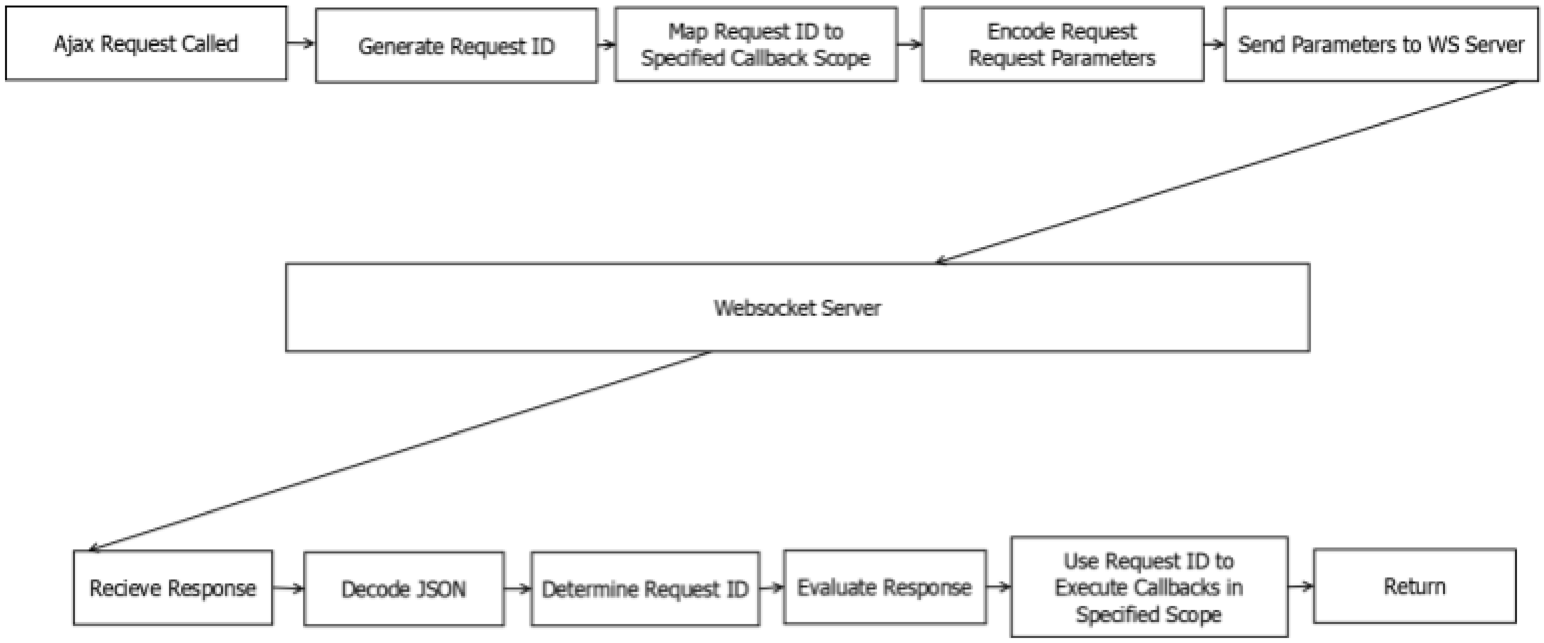
Ajax Request Called
Generate Request ID
Map Request ID to Specified Callback Scope
Encode Request Request Parameters
Send Parameters to WS Server
Websocket Server
Recieve Response
Decode JSON
Determine Request ID
Evaluate Response
Use Request ID to Execute Callbacks in Specified Scope
Return

Figure 2: Overriden Ajax Request Functionality

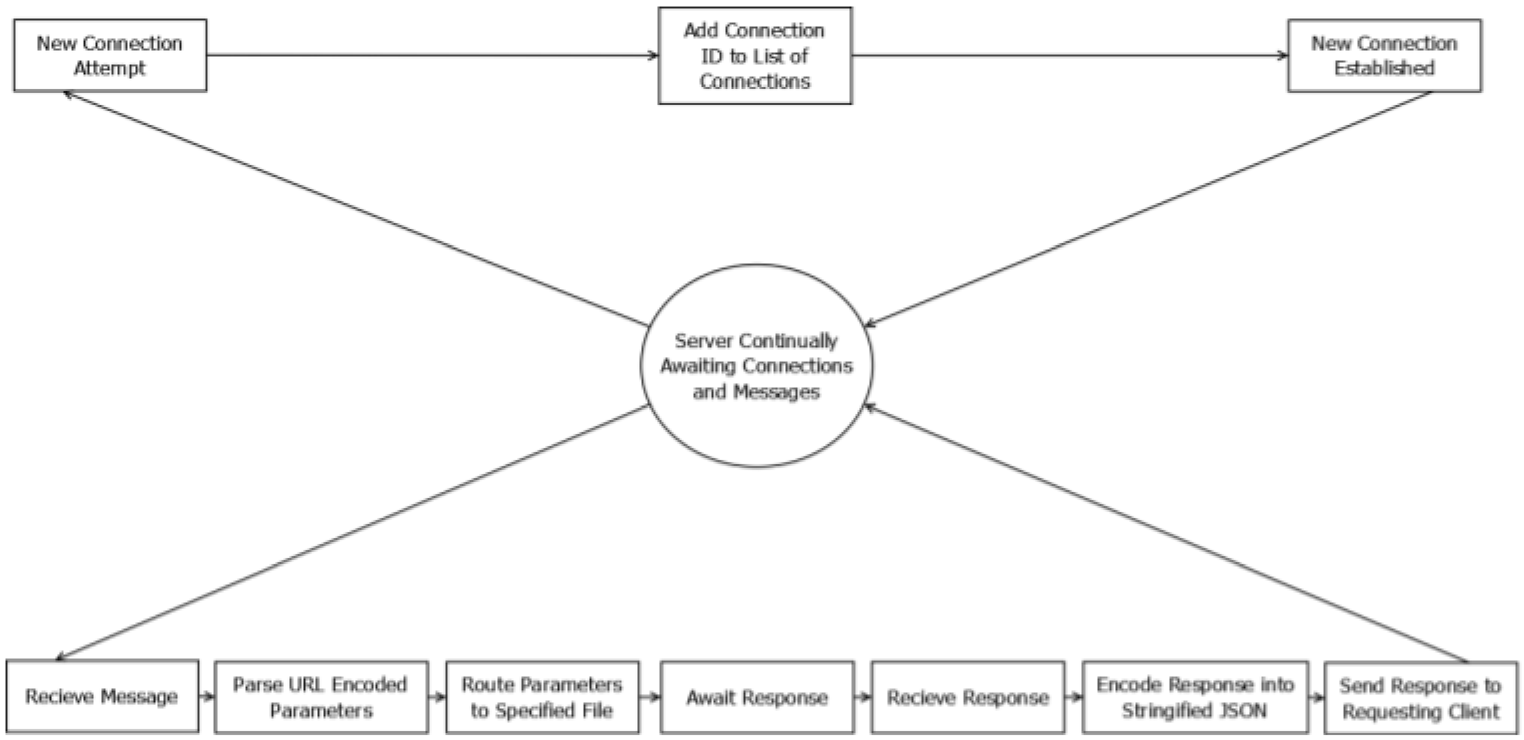
New Connection Attempt
Add Connection ID to List of Connections
New Connection Established
Server Continually Awaiting Connections and Messages
Recieve Message
Parse URL Encoded Parameters
Route Parameters to Specified File
Await Response
Recieve Response
Encode Response into Stringified JSON
Send Response to Requesting Client

Figure 3: Websocket Server Flow

V. Quality Assurance:

A. Unit Testing

Smaller scale unit testing was beneficial in guaranteeing that the behavior of small mechanisms within the application output the same results even after we had significantly modified the underlying machinery. When first beginning our implementation, we focused on very small portions of the application and used those as somewhat of an experiment to try and get a better grasp on the technology and techniques. As we moved forward and began to create modifications that would impact the entire application, these smaller tests and portions of code were exceptionally valuable when trying to determine the impact our changes had on the entire system.

B. Integration Testing

The client already had a relatively large suite of tests before we began working on the project. Since our project focused less on creating new functionality and more on optimizing some existing functions, the behavior of the application, in general, would not be expected to change much due to our modifications. With this in mind, ensuring that the changes we made did not significantly impact the core functionality of the application was critical and being able to utilize the provided tests was helpful.

C. Code Metrics

We utilize built-in JavaScript timing functions to record the time it takes to execute both an Ajax request and our new WebSockets request. We allow these timers to execute 20 times and then are able to get an average for the runtime of each framework. Depending on the nature of the function these timing metrics can show either a significant performance improvement or little to no change. Some functions have to access the same databases and depending on how much they use it, this can be a bottleneck for performance no matter what the request framework.

VI. Results:

As our group's task was to deliver a WebSockets framework using JavaScript and PHP, our original plan was to create a mechanism that allowed for the emulation of Ajax requests using WebSockets. This plan was more or less successful - the existing Ajax request function is overridden by a WS connection. This new WebSocket is created when the first request of a user's session is made. This single WebSocket connection remains open for the duration of the user's session and is closed only when the user exits the application or refreshes the page, which essentially restarts the session. With this persistent connection between the user's client and the server, our new framework resulted in improved connection speeds - in fact, our product delivers a ten-times speedup when compared to standard Ajax requests, on average.

The team decided that overriding each request would yield the highest likelihood of completion within the five week window. We made this decision because creating a function override made the code much cleaner, and allowed our code to run with more abstraction than if we had gone file-to-file and replaced every call to an Ajax request within Datava's system. As it stands, there are still some issues with the way that the WS mechanism interacts with other back-end components. In order for the system to work most efficiently for the client, these issues will need to be analysed in more significant detail than our group will be able to achieve by the end of the five-week sprint. Our current implementation does not use encryption of any sort, and will likely require some significant security updates in the future.

As a back-end project, we were able to manually test that requests made with WebSockets did not break the rest of the ESP from functioning by clicking through most functions and event-drivers as we came across them. We were also able to write code performance tests in the form of averaging the runtime of many calls to certain functions. These demonstrate that our WebSockets implementation does have the desired effect of increased performance in general. As our project is built on existing system code, it can be expanded by the company to incorporate security, persistent connections, and server-side push for two way communication. Overall, our group is satisfied with the proof of concept we developed and believe that we have demonstrated some promising results with regards WebSockets and their capabilities for Datava.

VII. Future Work:

While we are content with the results our group was able to produce, there are some features that would almost certainly move our framework forward and provide some added security and usability benefits. Some of these features are listed below.

A. Client to client communication

Two-way communication is something that will be left for future work. This would allow for instant updates when multiple users are editing the same form. This would resemble something like Google Docs which would help alleviate some conflicts from their own clients' using the ESP. While we did not have time to include this feature, the foundations for it are included within our current implementation. Within our MessageComponentInterface, a list of connected clients is maintained and it would not be too difficult to keep track of connections within a certain context and include extra information on certain requests to maintain a collection of user's within a specific application and who should be able to interact with them.

B. Implement SSL certificates and full WSS connections

Despite several different attempts, our group was unable to connect the application's client to our WebSocket server when it was using a WebSocket Secure connection protocol. Since there is a potential for large amounts of sensitive information to be sent over the websocket, this will likely be a critical feature to implement before this can go into production. Based on our understanding of the Ratchet API, it does not seem like a significant amount of work will be required, especially for someone who is more familiar with HTTPS connections.

C. Improve WebSocket server infrastructure

One feature that is not necessarily required but would undoubtedly help future scaling would be to improve the way in which Datava developers interact with the WebSocket server. At the moment, the back-end WebSocket server must be instantiated and run from the command line. Upon calling the file that does this, the developer will lose access to that instance of the terminal because the server will indefinitely occupy it. An automated way to start the server would be preferred. Additionally, all error messages are routed to the console, but a server-specific error log would be very useful for tracking any issues.

VIII. Appendices

A. Performance Testing Results:

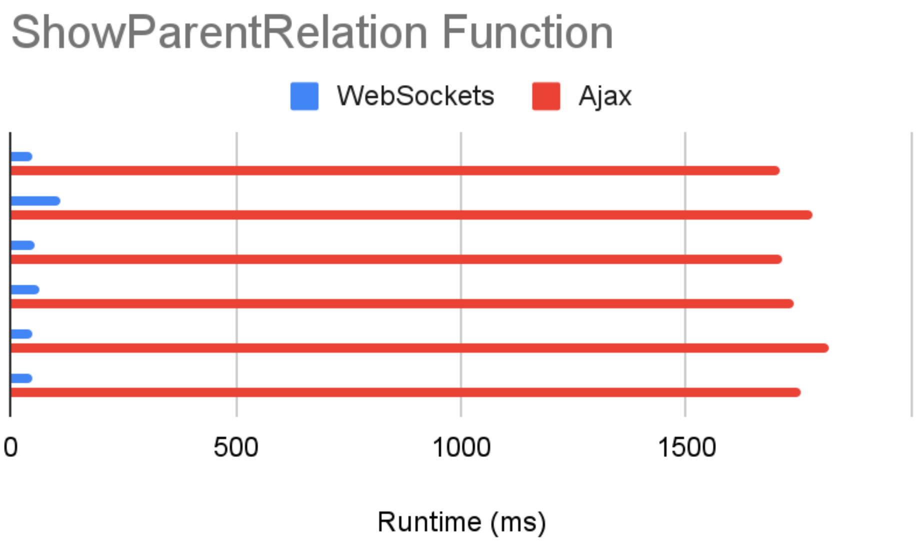

Figure 4: ShowParentRelation Function Timing Results

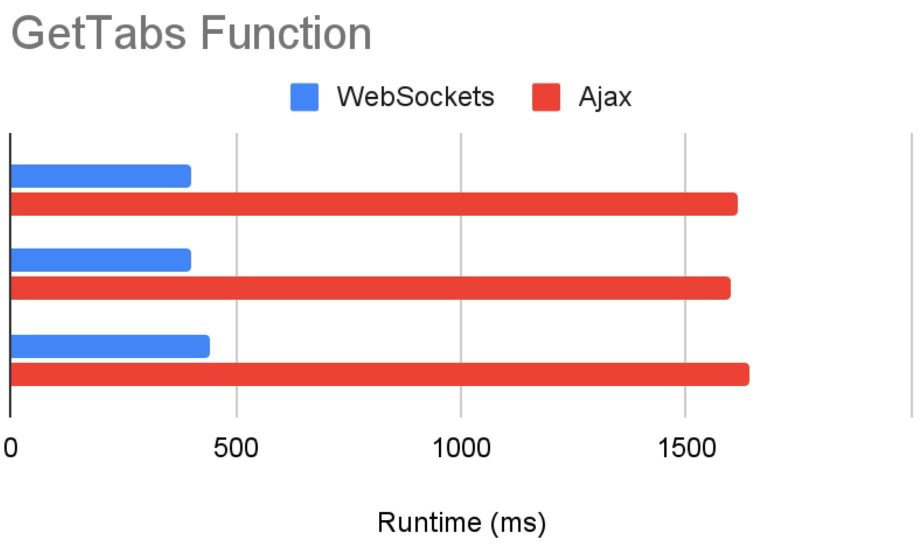

Figure 5: GetTabs Function Timing Results

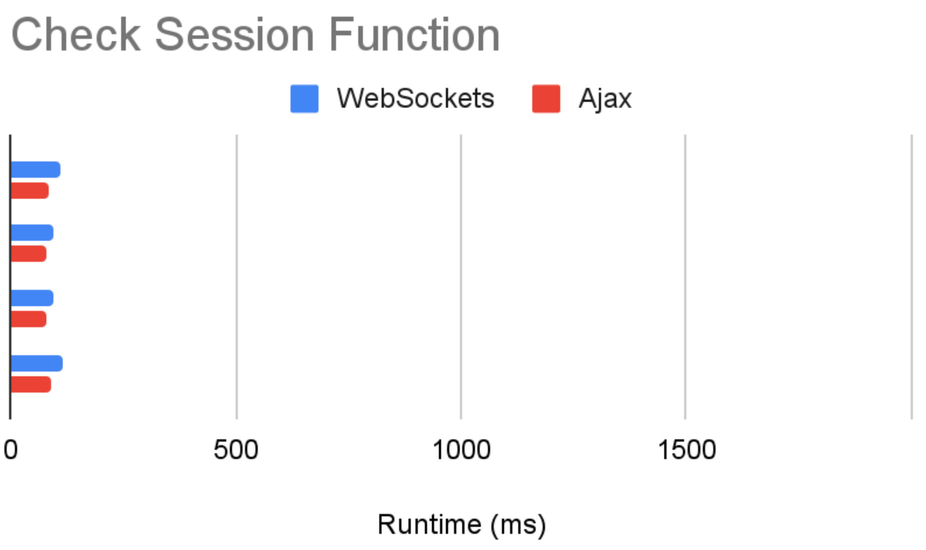

Figure 6: Check Session Function Timing Results

Figures 4-6 demonstrate any performance speedups our project yielded. Each red bar shows how long the runtime took to complete using the previous Ajax framework compared to the new implementation with WebSockets shown in blue. Each of the sets of bars is an average of 20 tests for each framework. Multiply the number of total bars by 20 to know how many total timing tests were performed. These results demonstrate that when faced with a function that takes a lot of Ajax requests, that time is cut significantly with a WebSockets implementation. More tests can be run in the future for further demonstration of performance increase, but these give a good visualization of the performance benefits of WebSockets.