A Web-App for Analysis of Honey Bee Hive Data

Gurney Buchanan
Department of Computer Science
Appalachian State University
Boone, NC, USA 28608
tashakkorir@appstate.edu

Rahman Tashakkori
Department of Computer Science
Appalachian State Univeristy
Boone, NC, USA 28608
buchanangb@appstate.edu

***Abstract*— The significant drop in honey bee population in recent years has been attributed to Colony Collapse Disorder (CCD). There have been many efforts to monitor honey bee hives and learn about their health and behavior. As part of early efforts for monitoring hives, our research group developed the Beemon system which obtains audio and video recordings as well as humidity and temperature data at beehives for further analysis. In order to make the analysis more efficient and widely accessible, we recently developed a web application to allow easier access to all the Beemon data and provide several tools for analysis. This paper provides details on the design and development of the web application used to make Beemon data available via web-based visualization tools for more effective analysis. Our web application was built using the MEAN (MongoDB, Express, Angular, Node.js) web stack which allowed JavaScript to be used on both the server- and client-sides. This system was built modularly, so the data visualization system was added on as a new server- and client-side module. The system queries data quickly from MongoDB, aggregate views and sends the large blocks of serializable data to the client using Socket.io. The client can interact with our server's data-providing endpoints and flexibly use any client-side data visualization framework. Our system employs C3.js, a D3-based data visualization platform, which includes base charts, to render visualizations of transmitted data.**

Keywords—Beemon, honey bee data, honey bee, Angular, Node.js, C3.js, JavaScript, Web-App, bee data visualization

I. INTRODUCTION

The number of managed bee colonies in the U.S. has steadily declined through the second half of the 20th century, and, in the early 2000s. Some beekeepers began reporting large numbers of bee die-offs to unknown causes, a phenomenon which is now known as Colony Collapse Disorder (CCD) [1]. Much of the honey bee related research conducted since the discovery of CCD has sought to determine the cause of the decline in the honey bee population [2]. Some of these research projects have attempted to bring automation to the study of honey bees [3-5], but there is still room for improvement by leveraging technological resources that were not available even a decade ago. In recent years, many efforts have sought to develop web based applications for monitoring natural systems. Our research project observes honey bee hives to determine their level of health and strength. A healthy hive has a strong number of diverse bees and demonstrates a steady traffic entering and leaving the hive. Also, a hive generates several different sounds and audio frequencies depending on its health and safety. The Beemon system, developed in our Visual and Image Processing (VIP) lab, provides an opportunity to reliably collect data from honey bee hives [6, 7]. This paper provides details on a web application created to provide easy, widespread access to the collected data. The web application will also include several tools for efficient analysis.

Several projects have attempted to solve the problem of data visualization and analysis on the web, including a number focused on natural systems. Huang et al. [8] developed server-side and client-side hybrid approaches that take advantage of a strong server-side solution. This allows the client to be served a universal HTML source that is compatible with all browsers and complies with web standards with minimal effort. Furthermore, such a system allows all administration, processing, and data to be centralized at the server. Server-side solutions are built using more mature and simplistic technology, utilizing a server to render a visualization and serve it to the client. However, a server-side solution comes with disadvantages, namely it offers a far less interactive and user-friendly solution and results in a large number of server requests. Huang et al. suggest that a client-side solution implemented using ActiveX or Java offers a more interactive experience for the user while offloading performance demands to the client. The client is sent data for processing and visualization, removing processing tasks from the server. Client-side solutions are typically less demanding on the client's persistent internet connection and are less demanding of the server. Client-side solutions have a major downside as they require downloading client-side software, or at least JavaScript source files for modern web applications. In the case of JavaScript web applications, some form of client-side code is released to the client for open access, whereas a server-heavy solution prevents the release of proprietary software to the public.

Thorvaldsdóttir et. al. [9] developed a visualization program that offers the option to load data from a remote source. The software utility handles large and diverse data sets, allowing for different scaled views to focus on small subsets of data or view the data as a whole, and allow the user a means of comparative analysis. This software architecture takes a layered approach to allow for various functionality such as access to different local and remote data sets and generating different views for data sets. The architecture contains a steaming layer which provides random access to different parts of a data set, both remote and local. The data layer handles the ingestion and management of imported data sets from different supported formats, and the application layer provides the user interface. This architecture is reminiscent of the MVC architecture. As part of the application layer, the user is allowed to view highlighted features and focus on certain portions of the desired data set. Their use of color to highlight important features and allow for quick navigation is of notable importance.

Wood et. al. [10] present a web-based system for sharing and visualizing neuroimaging data. This data can be queried,

viewed, and downloaded. They discuss a specific interface for query building that has the user build a query from certain building block representing data sets and operations. This interface presents an interesting potential for data filtering, grouping, and analysis as it allows the user freedom to control what data is retrieved and how the data is analyzed. This method can be further extended with new innovations in real-time data synchronization between the server and the client. Their architecture involves an Apache server which serves from processed PHP scripts in order to authenticate the user and set up their session. These webpages then feed into an HTTP interface to a Node.js server to retrieve available query elements and query results. Interactions with JSON data, the complexities involved, and the methods for sending large data sets through a serialized connection are also discussed. These methods have evolved since their paper was written, but the principles remain the same.

Ma et. al. [11] introduced a monitoring system for Internet of Things (IoT) devices build on top of WebSockets in Socket.io, a Node.js server, a MongoDB database, and a web portal for data access which uses Chart.js for data visualization. They propose the use of WebSockets through Socket.IO as the single means of bi-directional communication between a server, a device, and a client. In this model, the server both ingests data from IoT devices into the database and serves data to a client from the database. This architecture allows for the user to access data from the device directly and also control the IoT device directly through the webserver. The client has access to real-time and historical data that can be visualized using packages such as Chart.js. Chart.js utilizes the HTML5 canvas and JavaScript to create interactive charts from large data sets served to the client via a WebSocket. Another feature of note in the proposed architecture is the NOSQL database solution - MongoDB, which was selected for its balance of read/write performance and its ability to easily handle large data sets. *Ma* et. al. define a very adaptable and general architecture for efficient IoT connectivity, data transfer, control, and data visualization.

Cawthon et. al. [12] explore the effect of aesthetics and style on the effectiveness of a data visualization. They assembled an online study to test the effectiveness of different visualizations relative to the rated aesthetic quality of a given visualization. Their goal was to identify correlations between the rated aesthetic beautify of a visualization and its ability to correctly and efficiently convey information. To do so, they first asked 285 valid users to *"reflect on the aesthetic quality of the image"* as they *"would with a painting or illustration"*. Following an aesthetic ranking, they asked the users 14 questions based on the data and rated their responses on correctness, response time, and task abandonment. Specifically, they allowed the user to abandon their task if it was found very difficult. Their results concluded that a visualization that is aesthetically appealing can effectively portray the data and may also encourage the user to spend more time analyzing the visualization. They mentioned that one of the techniques, *Sunburst*, that was ranked very aesthetically appealing had very low abandonment and a high rate of correct responses. They state that "These results prove that participants who did not immediately locate the correct answer felt encouraged to continue their task." However, they noticed that two of the visualizations ranked visually unappealing had the highest accuracy and speed ratings. Their results also point out that their choice of color palette as greens, blues, and browns were pleasing to the human eye. These observations and results are useful to keep in mind when developing adaptable visualizations of complex data that could easily become overwhelming or complex.

Gomez et. al. [13] present BioJS, an open source JavaScript framework for the creation and use of biological data visualizations. Their primary goal was to create a framework for others to build upon to create various reusable visualizations. As such, they opened the framework to the community for extension. They defined a framework that, once learned, would allow other developers to easily create new reusable visualizations as components. BioJS, as a framework, simply defined the component architecture, the protocol for event handling and communication between components, the component extension through Object-Oriented Inheritance, documentation format, and documentation on how to include examples to test the component functionality. Their framework seems to provide a sufficient base for a coherent and extendable visualization framework, meanwhile taking a hands-off approach and allowing other frameworks, packages, and plotting libraries to be used. As they say, they are "platform agnostic". Their project continues today and provides a number of JavaScript-based web visualizations for biological data.

Wessels et. al. [14] defined a framework for remotely rendering and streaming data visualizations. Their architecture defines four primary "layers" or, as they call them, components. These components are the Server, Visualization Engine, Daemon, and Client. The Server houses the rendering hardware and executes the Visualization Engine and the Daemon. The Visualization engine is tasked with rendering images to be sent to the client. The Daemon runs continuously and spawns visualization processes as requested. The Daemon would now be known as the web server as it controls all socket interactions and controls the provision of resources as requests are received. The Client is directly sent rendered images of their data visualization through a data stream to an HTML Canvas and can interact with these visualizations. If they interact with a visualization, for example a 3d image, their actions are sent back to the server, where a new frame is rendered and streamed back to the client. This architecture was created with the goal of overcoming client-side rendering limitations with large data sets. The client can slow down or even crash if given too many points to render. This architecture offers an interesting solution by offloading the rendering and data accessing tasks to the data storage server. The rendered visualizations are then streamed to the client. This architecture can be augmented to work with the other architectures discussed earlier, but they put more work on the server and less on the client!

The D^3: Data-Driven Documents charting platform introduced by Bostock et. al. [15] outlines 3 specific objectives: compatibility, debugging, and performance. The D3 team's work towards compatibility led to a focus on interoperability with outside tools and reusability through extension with a method to directly access the native representation of the visualization. The debugging-centric approach led to a focus on

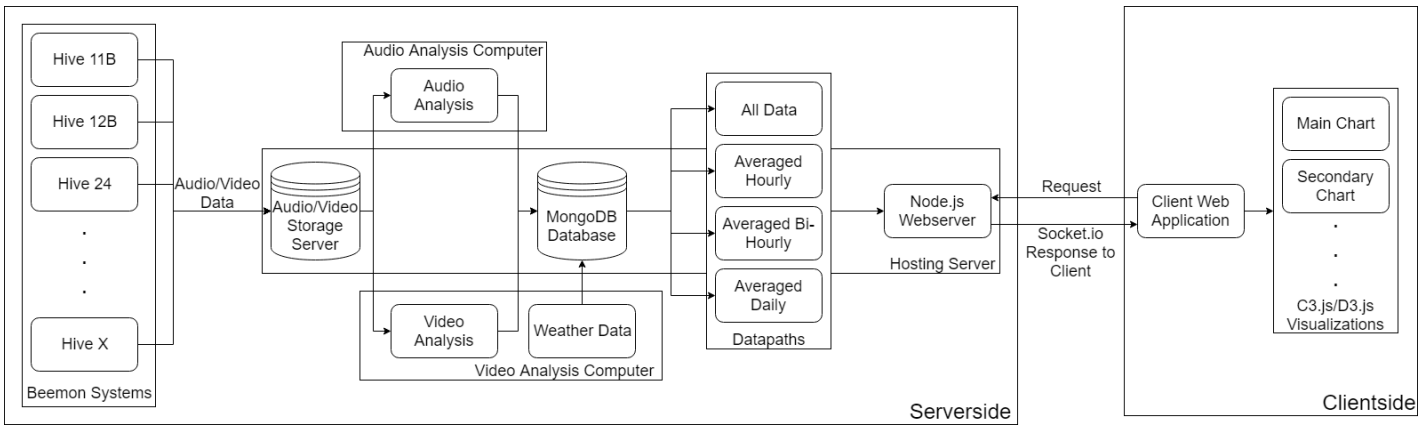

Figure 2: Beestream Architecture

the control flow, encouraging them to allow the developer to interact with every layer of the control flow to facilitate debugging. Their performance objective resulted in a focus on transformation. By focusing on transformation between states, you can avoid doing redundant work re-rendering data that hasn't changed. This also resulted in an affinity for interactivity. They provide an overview of their design choices related to the selection of DOM elements, their work towards an intuitive mapping of data to visual elements, their focus on transitions, interactions, and animation, and their focus on a modular design providing basic functionality while maintaining extendibility.

II. The Beemon System

The recent decline in the population of honey bees is attributed mainly to the Colony Collapse Disorder (CCD) which is strongly believed to be caused by a combination of several things. Although scientists have made a significant progress in determining the cause of CCD, there are still many unknowns. Our Beemon project was the result of several years of work developing a monitoring system for honey bee hives that would allow us to acquire audio and video recordings at honey bee hives as well as humidity and temperature data to be used for studying honey bee behavior. The Beemon system records one minute video and audio recordings and sends them to a server in our department for further analysis. This Raspberry Pi-based system, as shown in Figure (1), has the ability to operate continuously in an outdoor apiary environment and allows for constant, near real-time, data collection.

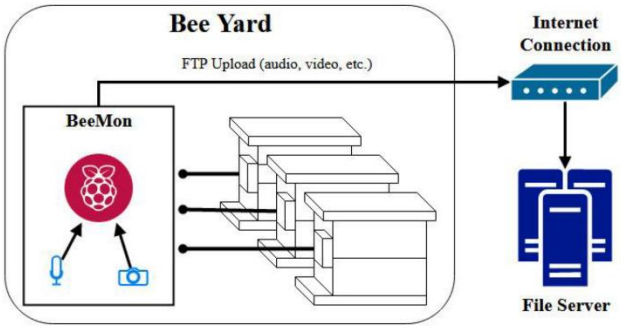

Figure 2: Beemon Honey Bee Monitoring System

The video and audio recording files that are stored on the server can be accessed via file transfer systems which would make the process slow and cumbersome. In order to make the data access and analysis more efficient, a well-designed web-based system was needed.

III. Web Architecture

Our web architecture aims to facilitate adaptable, extendable, and reusable data visualization through an interactive web application. This architecture is divided into a server-side and client-side architecture. The server-side architecture defines a web server whose task is to access data and deliver it to the client. The server will interact with available data and analytics and aggregate results as the developer deems appropriate. The client-side architecture allows the end user to filter, query, visualize, and interact with the data available from the server. The server-side architecture was implemented in JavaScript using Node.js and the client-side architecture was implemented using JavaScript and TypeScript using the Angular web framework. Our implementation of this architecture is visualized in Figure 2.

IV. Server-Side

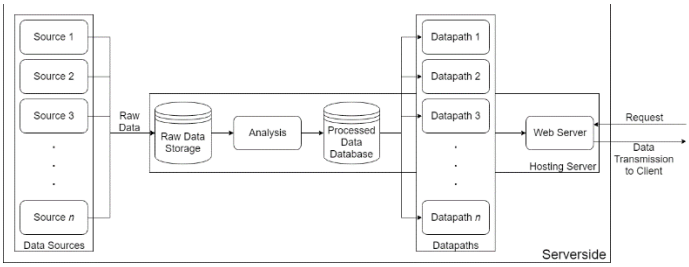

Figure 3: Server-Side Architecture

Our server-side architecture, as illustrated in Figure 3, defines a path between a raw data source and a database of processed data. This architecture can vary by implementation but it is important to have a queryable data source. This database can exist in as many instances, decentralized nodes, or collections/tables as desired. In our architecture, a web server relies on a set of developer-defined data paths to access the database. Each data path is required to publicly declare how many data elements it will return and have a function to carry

out a query. The query function should execute a query and aggregate/summarize the data if desired then return the queried data. This data path architecture allows the developer to aggregate data from multiple different sources if desired. Furthermore, it offers the dynamic behaviour necessitated by the limitations of web-based data visualization.

Web based data visualizations are limited by the performance of the client-side code in a browser, the end-user's internet speed for data transmission, and the main memory (RAM) available to the user's browser to store the working data set. The data set is typically larger than what can reasonably be transmitted, stored, and rendered by the browser. This problem is resolved using our data path architecture. The client application will determine the capability of a user's browser and dynamically request a certain number of data points. The web server will search the list of developer defined data paths until it finds the data path that provides the most data without exceeding the requested limit. It will use this data path to query the database(s) and return the queried data. This allows the web server to transmit only the smaller, summarized data set rather than the full data set. If aggregation/summaries were completed on the client-side, the server would have to transmit the entire data set to the client and the client code would have to process the data. By processing the data on the server, we ensure that the client only receives a reasonably sized data set ready for visualization.

Beyond data scalability, the data path architecture also offers a very simple interface for extension. This architecture mandates that the developer dynamically load all data paths from a directory and define server side logic to select the most optimal data path to respond to a given request. This design choice means that whenever the developer wishes to add a new data source or method of aggregating data, they need only create a new data path with the appropriate exports. Each data path is simply a modular piece that can easily be added, modified, or removed. In our implementation, the modularity of data paths proved useful when different aggregation methods and data sets were desired.

Once the developer has built modular data paths, they must build a driver web server to interact with the data paths and transmit data. This architecture is specifically optimized for use with websockets to transmit serializable data in JSON format. Once the web server has queried data using an optimal data path, it will transmit the data to the client using a transmission protocol of the developer's choice. As we discuss in our implementation section, we suggest the use of websockets using Socket.io. This modern protocol allows the developer to dynamically update the data providing a "live" visualization for the client, which is especially useful for IoT (Internet of Things) based applications.

Our architecture was implemented to visualize analytics data for the IoT Beemon project. Our implementation is illustrated in Figure 4. Each Beemon system uploads video, audio, and temperature/humidity data to a storage server. When a video or audio file is uploaded to the storage server, a document representing that file is inserted into a MongoDB database. The new file is then detected and analyzed by video or audio analytics software, called BeeVee and BeeSound respectively. BeeVee uses image processing techniques to count the number of bees that arrive and depart the hive within a given video. The video analysis data is then associated with that video's document in the MongoDB database. This provides our first quantitative metric for honey bee activity. BeeSound dehums raw audio data and creates spectrograms for each dehummed file. Spectrograms, along with other audio analytics data, are inserted into the MongoDB database and associated with a given audio file. This provides another quantitative metric for a hive. Finally, another application collects weather data from online sources for each Beemon system's location. The weather data is stored in the MongoDB database and is associated with the hive and datetime. These applications provide near real-time analytics to the database.

V. Server-Side Implementation

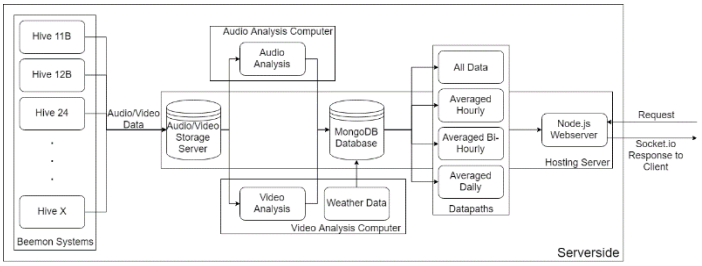

Figure 4: Server-Side Implementation

A web server was implemented using Node.js, Express, Mongoose, and Socket.io to handle requests from a client web application. Node.js allows us to efficiently build and execute a web server using JavaScript. Express is a widely-used solution to create a simple, adaptable, and extendable HTTP web server. Mongoose allows us to interact with MongoDB databases, collections, and views through developer-created schemas written in JavaScript. Socket.io allows us to quickly and efficiently send serializable data over websockets. We chose to implement our web server using Node.js and associated packages due to its modernity, adaptability, support, and features. By using Node.js, we were able to implement our entire server in JavaScript, leverage JavaScript's asynchronicity, and handle all data from MongoDB to the client in the serializable JSON format. Furthermore, Node.js features allowed us to implement our modular data path architecture easily. Each data path is implemented as a Node.js module that exports the number of items returned per unit (in our case per day), a name for the data path, and a query function that accepts a range to be queried, the data sets to be returned, and a callback function. In our implementation, 4 data paths are used. Each data path in our implementation accesses a different database collection or view. The data path files are placed in a specific data paths folder. The web server will include any data path file in the data paths folder provided it exports the necessary attributes. This provides future developers an easy method of extension.

Our web server acts as a "driver" for data paths. When the server receives a request for *n* data points, it will simply loop through the list of available data paths and find the path that provides the most data points less than *n*. It will then use that data path to query the MongoDB database. When the data path finishes its asynchronous query, it will call a callback function

to send a reply to the client. This reply will be sent using websockets. Our implementation uses Socket.io for these websocket communications due to its simplicity, reliability, and ubiquity. Socket.io allows us to easily and efficiently transmit the JSON response from a MongoDB query.

The driver components handle all user interactions concerning data requests, selection, and filtering. Driver components send requests to the server and receive responses including data payloads. They are responsible for obtaining the appropriate data to serve to all chart components. As such, each

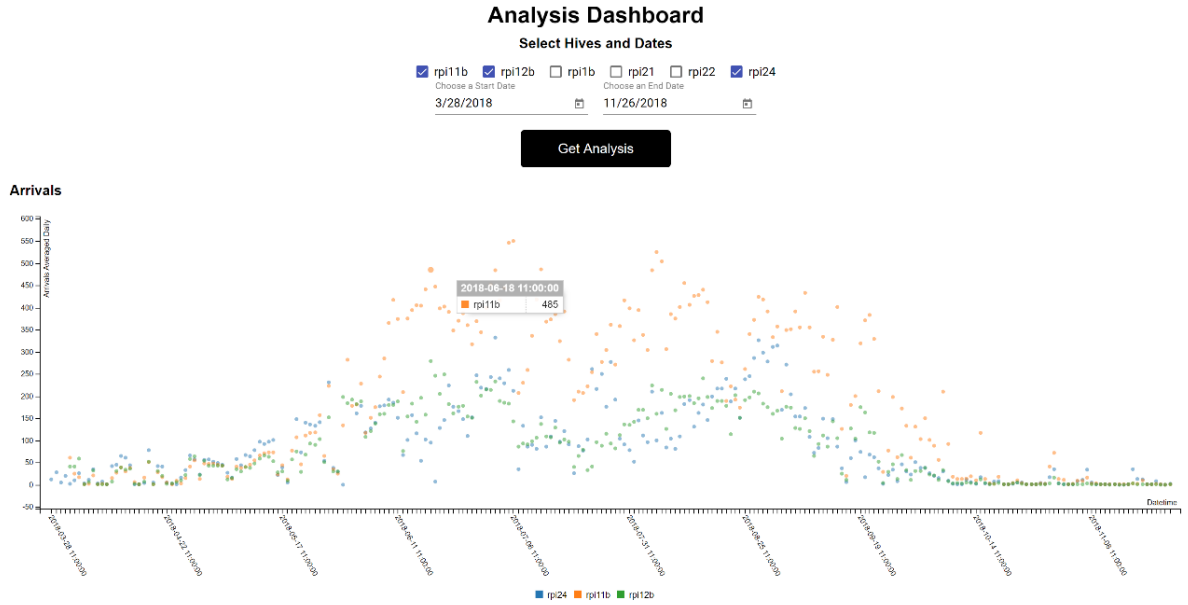

Figure 5: Beestream Dashboard

VI. Client-Side

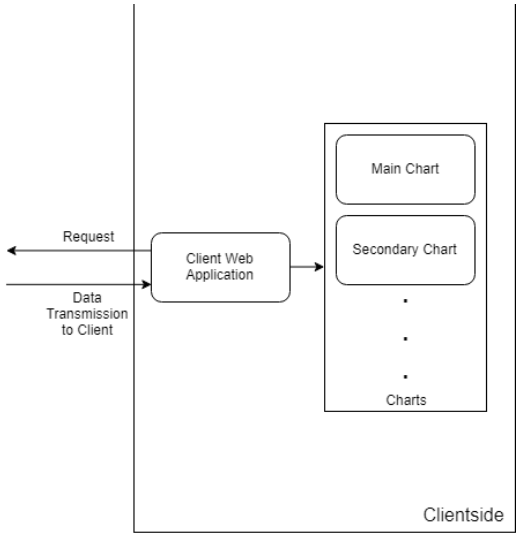

Figure 6: Client-Side Architecture

Our client-side architecture, shown in Figure 6, presents a simple architecture for the creation of a web-based data visualization client. This architecture seeks to separate data filtering and selection components from the charts and visualizations in an application. Holding to the Single Responsibility Principle (SRP), our architecture consists of two different types of components: driver components and chart components.

driver component will maintain a list of charts that are rendered as part of its page. Each registered chart component will have a data set or set of data sets it requires to render a visualization. When a driver component receives new data, it should pass the proper data sets to each chart. If a chart needs a data set that the server could not provide, that driver should not allow that chart to render.

The chart components are modular and define a single chart or set of charts. Each chart component should declare which data set(s) it needs to render and define a function to update the chart with new data. Each chart should have a common interface. Given a common interface, the driver component can maintain a list of chart components and dynamically update charts with new data whenever it receives a server response.

The primary concept of the client-side architecture is to create a driver that acquires data and manages chart components. The driver should rely on an abstraction for the chart components. This will allow the driver to keep a list of active charts and invoke a function on each chart to update data. In this case, charts are "subscribed" to receive updated data from specific data sets when available. The developer should be able to create new charts based on this abstraction, place them on the page, and view them immediately. This modularity makes extension, specifically adding new charts, trivial.

VII. Client-Side Implementation

Our client-side architecture was implemented as part of a web application to visualize data from the Beemon project. A web application was implemented using Angular, a TypeScript

based framework for building dynamic web applications. C3, a D3-based data visualization framework, was used for charts and visualizations. Our architecture suits the Angular framework very well. Angular defines each web page or set of web pages as a module. Each module typically includes a primary component and can include as multiple different components. A primary component was created for the driver component and numerous chart components were created. All components are part of the same module and all chart components have the same interface. The driver component includes each of the chart components in its HTML page template. It can access the chart components directly using Angular's component interaction functionality. As a result, the driver component can invoke functions to access each charts required data set and update their data when new data is available.

VIII. RESULTS

The web application created for visualization and analysis of honey bee data provides a flexible and modular platform for visualizing the Beemon data. The analysis provides features for selecting different hives and different data and set various date ranges. A screenshot of one of the pages in Figure 5 provides a visual of the previous year of data for our 3 primary honey bee hives. A powerful feature of this application is that the user can mouse-over any of the points in the plot to obtain detailed information for the selected point, in this case date and time. This tool can be used effectively for analysis of Beemon data and the system can be expanded to other data acquisition systems.

REFERENCES

[1] P. Meumann and N. L. Carreck, "Honey bee colony losses," Journal of Apicultural Research, vol. 49, no. 1, pp. 1-6, 2010.

[2] J. D. Ellis, J. D. Evans and J. Pettis, "Colony losses, managed colony population decline, and Colony Collapse Disorder in the United States," Journal of Apicultural Research, vol. 49, no. 1, pp. 134-136, 2010.

[3] J. M. Campbell, D. C. Dahn and D. A. Ryan, "Capacitance-based sensor for monitoring bees passing through a tunnel," Measurement Science and Technology, vol. 16, no. 12, p. 2503, 2005.

[4] J. Campbell, L. Mummert and R. Sukthankar, "Video monitoring of honey bee colonies at the hive entrance," Visual observation & analysis of animal & insect behavior, ICPR, vol. 8, pp. 1-4, 2008.

[5] V. Estivill-Castro, D. Lattin, F. Suraweera and V. Vithanage, "Tracking bees-a 3d, outdoor small object environment," in Image Processing, 2003. ICIP 2003. Proceedings. 2003 International Conference on, 2003.

[6] M. Crawford, "Automated collection of honey bee hive data using the Raspberry Pi," thesis, 2017.

[7] *A Honeybee Hive Monitoring System: From Surveillance Cameras to Raspberry Pis*, Rahman Tashakkori, Nathaniel P Hernandez, Ahmad Ghadiri, Alekxander P Ratzloff, and Michael B Crawford, IEEE SouthEastCon 2017, March 30-April 2, 2017, Charlotte, NC.

[8] B. J. &. H. L. Bo Huang, "An integration of GIS, virtual reality and the Internet for visualization, analysis and exploration of spatial data," *International Journal of Geographical Information Science*, vol. 15, no. 5, pp. 439-456, 2001.

[9] H. Thorvaldsdoittir, J. T. Robinson and J. P. Mesirov, "Integrative Genomics Viewer (IGV): high-performance genomics data visualization and exploration," *Briefings in Bioinformatics*, vol. 14, no. 2, pp. 178-192, 2018.

[10] D. Wood, M. King, D. Landis, W. Courtney, R. Wang, R. Kelly, J. A. Turner and V. D. Calhoun, "Harnessing modern web application technology to create intuitive and efficient data visualization and sharing tools," *Frontiers in Neuroinformatics*, vol. 8, p. 71, 2014.

[11] K. Ma and R. Sun, "Introducing WebSocket-Based Real-Time Monitoring System for Remote Intelligent Buildings," *International Journal of Distributed Sensor Networks*, vol. 9, no. 12, p. 867693, 2018

[12] N. Cawthon and A. V. Moere, "The Effect of Aesthetic on the Usability of Data Visualization," *2007 11th International Conference Information Visualization (IV '07)*, pp. 637-648, 2007.

[13] J. Gomez, L. J. Garcia, G. A. Salazar, J. Villaveces, S. Gore, A. Garcia, M. J. Martin, G. Launay, R. Alcantara, N. del-Toro, M. Dumousseau, S. Orchard, S. Velankar and e. al., "BioJS: an open source JavaScript framework for biological data visualization," *Bioinformatics*, vol. 29, no. 8, pp. 1103-1104, 2013.

[14] A. Wessels, M. Purvis, J. Jackson and S. Rahman, "Remote Data Visualization through WebSockets," *Eigth International Conference on Information Technology: New Generations*, pp. 1050-1051, 2011.

[15] M. Bostock, V. Ogeivetsky and J. Heer, "D3: Data-Driven Documents," *IEEE Transactions on Visualizaiton and Computer Graphics*, vol. 17, no. 12, pp. 2301-2309, 2011.