MIDDGUARD
COLLABORATIVE AND EXTENSIBLE VISUAL ANALYTICS

Dana Silver

Adviser: Professor Christopher Andrews

A Thesis

Presented to the Faculty of the Computer Science Department

of Middlebury College

in Partial Fulfillment of the Requirements for the Degree of

Bachelor of Arts

May 2016

ABSTRACT

MiddGuard is a web framework for collaborative and extensible visual analytics. It is built on the idea that a data-driven investigation can be represented as a graph of composable, chained data transformations and visualizations that are completely customizable by users and can take input from other arbitrary tools in the same graph. The pairing of customization and arbitrary chainability enables the creation of investigation-specific, yet reusable tools. Additonally, MiddGuard is built for teams to collaborate on an investigation both asynchronously and synchronously. Multiple investigators can connect to a single MiddGuard server to see a database-persisted, real-time reflection of their colleagues' work building and generating data for a visual analytics investigation.

ACKNOWLEDGEMENTS

Professor Andrews for his continued advisorship since we started collaborating during the summer of 2014. His work in visual analytics continues to inform and inspire my work on MiddGuard.

My family and friends for their support.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Visual Analytics

Visual analytics is the science of analytical reasoning facilitated by interactive visual interfaces [13]. A visual analytics based investigation combines tools to transform and visualize data with human judgment to evaluate information and gain insight. Effective visual analytics tools need to transform disparate types of data from different sources to support visualization and analysis. Investigations often involve responding to or preventing a threat and are time sensitive. In Illuminating the Path, Thomas and Cook write that "Research is needed to create software that supports the most complex and time-consuming portions of the analytical process, so that analysts can respond to increasingly more complex questions." [13]. For an investigation to be effective and conclusions to be convincing, results have to be understandable and reproducable.

MiddGuard aims to address the challenges posed by visual analytics. It partitions the analytic process into a series of data transformations and visualizations, combining them into a unified, transparent data-flow model with a visual representation. MiddGuard provides the backing framework and integrated analytic environment to communicate data between teams of investigators and load/unload visualizations. By building on MiddGuard instead of implementing this scaffolding themselves, analysts can devote their time to the investigative process.

MiddGuard's model for extensibility allows developers to focus solely on writing the tools they need to transform data and render visualizations. It exposes simple APIs to extend the framework while remaining agnostic as to the implementation details: both transformation and visualization tools can be written using any technologies. This allows developers to produce bespoke tools quickly.

1.2 Previous Work on MiddGuard

1.2.1 VAST 2014

The VAST Challenge is a visual analytics competition organized by Visual Analytics Community with results presented at IEEE VIS. The challenge gives competitors a description of a crime scenario and data surrounding the crime. It asks analysts to create and use tools to investigate the data to indentify abnormalities, people of interest, and clues for the police to pursue. The VAST 2014 Challenge [6] posited the following fictitous scenario:

> In January, 2014, the leaders of GAStech are celebrating their new-found fortune as a result of the initial public offering of their very successful company. In the midst of this celebration, several employees of GAStech go missing. An organization known as the Protectors of Kronos (POK) is suspected in the disappearance, but things may not be what they seem.

During summer 2014, Christopher Andrews and Dana Silver collaborated on a submission for VAST 2014 Mini-Challenge 2, one of four challenges (including an all encompassing “Grand Challenge”) dealing with the VAST 2014 Challenge scenario.

For our VAST 2014 submission, we created a web interface to visualize and analyze data from the challenge scenario. Data were preprocessed using several disjoint Python scripts and the resulting manipulations were persisted to a SQLite database. On the back-end of the web service, a simple RESTful Python web server implemented with Flask [11] and Flask RESTful [5] queried the database and transformed data for various front-end visualizations. The server also manipulated data on a request-by-request basis using analyst input from the interactive visualizations. Figure 1.1 shows the web interface for our tool.

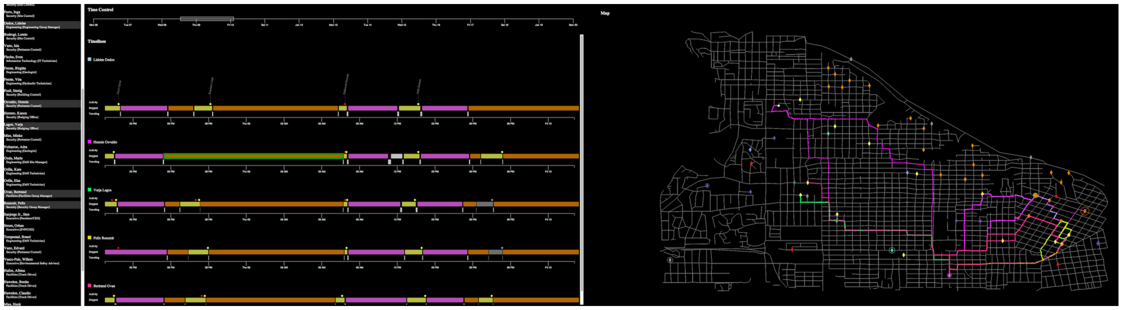

Figure 1.1: The web interface for Andrews and Silver's VAST 2014 entry. The visualizations, from left to right, are a list of people, a master brushable timeline and individual timelines for each person with selectable events, and a map of GPS traces from the individuals' cars.

For an example of the flow and feedback loop between preprocessing scripts, back-end server, and front-end visualizations we look at how we used the Mini-Challenge 2 geographical data to identify points of interest and associate them with car destinations. The VAST 2014 Mini-Challenge 2 dataset included vehicle tracking data from company cars, an ESRI shapefile of the island where GAStech is located, and an illustrated tourist map of the island. Tracking data contained lists of latitude, longitude, timestamp, and car ID.

We wrote a preprocessing script in Python to iterate through individual cars' GPS traces from the vehicle tracking data, identify periods where a car was stopped, and save the coordinate where the car stopped as a destination for the associated car. On the front-end, an interactive visualization rendered the shapefile and preprocessed tracking data to draw a map of the city overlayed with cars' movements and destinations. We created points of interest on the map using car destinations and names from the tourist map. Persisting the association of point of interest and a single destination to the database ran a procedure that identified other nearby destinations to automatically associate with the same point of interest.

Our VAST 2014 submission was unsuccessful. Working on the tool took most of the available time and we were not left with sufficient time to complete the investigation and write up the results.

1.2.2 MiddGuard: Summer 2014

The first version of MiddGuard, which was developed in response to our summer research at Middlebury working on the VAST 2014 Challenge, attempted to generalize parts of the web server and front-end that could be reused throughout multiple investigations into a web framework, while staying unopinionated with respect to the data it could handle.

From the VAST 2014 Challenge we drew conclusions that influenced this first version of MiddGuard. We found that the while the web could be an effective platform for visual analytics, the overhead of creating custom tools, getting those tools to work with the rest of the system, and implementation bugs in the server-client communication hindered our progress investigating. To address these issues, the framework's primary features were automatic persistence to a database, data transport between the server and connected web clients in real-time, centralized data storage in the web browser, and visualization module loading/unloading in the browser.

This version of MiddGuard achieved flexibility by automatically loading three types of customizable packages. These were referred to as analytics, modules, and models. Analytics were scripts that could be triggered by a remote procedure call from a front-end visualization. They could be passed data from the front-end. In the context of the VAST 2014 example, they were meant to handle computations like finding other destinations near a point of interest.

Modules were front-end visualizations that used JavaScript and CSS to render and style elements in the browser's DOM. Visualizations were interactive, could communicate with the backend to update and persist data, and could save state to a global state handler to link visualizations to each other. For example, a master brushable timeline saves the boundaries of the brushed region to its state, which other timelines read to update their detail view.

Models were table-level schema for the database, intended to allow MiddGuard to work with any data. A database table could then be created from each model. The entire database was accessible on the front-end, with each table represented by a Backbone.js Collection, which acts like an array of table rows. Collections were updated in real-time using a publish-subscribe like method. Updates to a collection on the front-end and to models on the back-end were communicated to one another in real-time. This allowed investigators to modify the data in visualization modules and analytics packages without implementing communication. By listening to changes in a Collection, a visualization could rerender as soon as data changed on the server or in another investigator's browser. The real-time, database-persisted communication protocol for models allowed investigators to collaborate synchronously and asynchronously.

1.2.3 VAST 2015

Christopher Andrews and Jullian Billings used MiddGuard for the VAST 2015 Challenge. They report that the framework allowed them to take a modular approach to developing tools for the investigation, deploying visualizations as needed without needing plan and coordinate the entire investigation before it began. They expanded the front-end state manager and used it to link their visualizations: "The shared state provided by Middguard meant that the modules could be easily snapped together into an integrated environment, facilitating the flow of information between the tools. This sped development because tools could be simple and focused, with data selection and filtering shared between tools." [1]. MiddGuard was well received by visual analytics professionals, winning a VAST 2015 Challenge award for integrated analysis environment.

The VAST 2015 Challenge investigation revealed some shortcomings of MiddGuard.

Storing all data in a web browser was not realistic. Datasets for investigations, including VAST, are often several gigabytes in size, more than can fit in the browser while maintaining the performance required for interactive visualizations. Even with modifications to load subsets of the data, the Backbone Collections quickly grew large, and filled with unnecessary data not reflected in any active visualizations. View Reference Counting was designed to address this issue.

Analytics packages, one of MiddGuard's built-in tools for extensibility, designed to run arbitrary code via remote procedure calls from the front-end, were not sufficient to obviate the need for preprocessing scripts. The investigators still wrote Python scripts to transform data and alter the database outside MiddGuard. The lack of record of how these scripts were used added a layer of opaqueness to the analytic process, making results hard to reproduce and collaboration difficult. The framework designed and implemented in this thesis addresses the issues of transparency and reproducibility in the analytic process, while introducing a method to include the preprocessing script contents in MiddGuard.

1.2.4 View Reference Counting

In the original implementation, one of MiddGuard's weaknesses was handling large amounts of data on the front-end. The framework was implemented to load the entire database into the browser with the idea that investigators would need access to all data during the investigation. MiddGuard's server would continue to push data updates to connected clients as they became available. However, with the large dataset from the VAST 2015 Challenge, the browser was not able to handle all the data at once. MiddGuard was modified with a stopgap solution during VAST 2015. Instead of loading all data from the outset, visualization modules made custom database queries as necessary.

This did not solve the problem of unused data in the browser. Once downloaded to the browser, data was never removed, even after the visualization had requested the data was no longer in use. MiddGuard stores all data in a central location to avoid the duplication that would occur by having each visualization store its own data. This makes it impossible for a visualization that has requested data to clean up after itself. Another visualization may have requested and currently be using the same data.

To keep the deduplication advantages of central storage and clean up after visualizations that were removed from the browser, we implemented automatic memory management in the browser called View Reference Counting. View Reference Counting (VRC) maintains an array of references to the views that use each piece of data as an attribute on the datum's Backbone.js Model. When a visualization (a Backbone.js View, hence the name) is removed, its reference is removed from the model. When a model has no view references it is removed from the browser.

Figures 1.2 and 1.3 demonstrate the efficacy of View Reference Counting through the three memory snapshots taken by the Google Chrome DevTools Memory Profiler. After a view with several megabytes of data was added and removed, MiddGuard cleaned up the data and the browser was able to reclaim the memory.

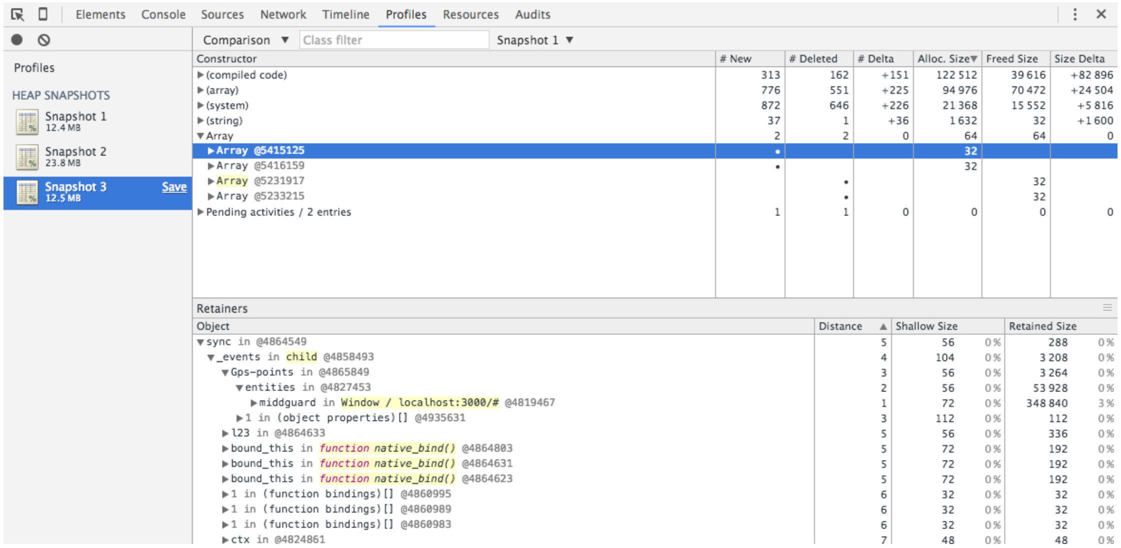

Figure 1.2: A screen capture of the Google Chrome DevTools Profiler demonstrating the efficacy of View Reference Counting. The panel on the left shows three snapshots. Snapshot 1 was taken before a view was added. Snapshot 2 was taken after a view was added and rendered with a significant amount of data loaded into the browser. Snapshot 3 was taken after that view was removed and the memory was reclaimed.

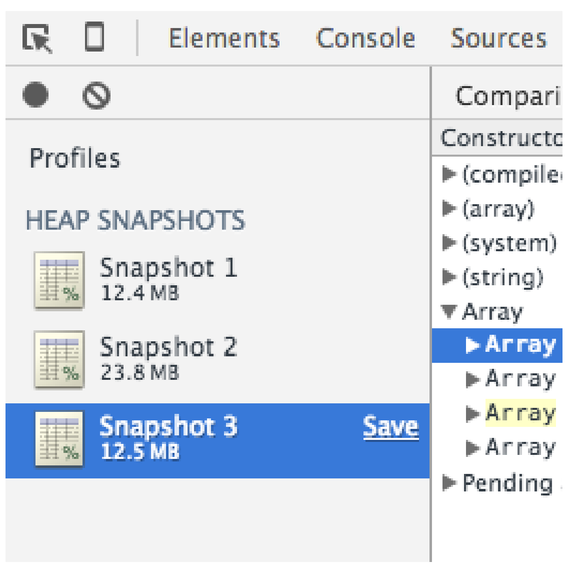

Figure 1.3: The snapshots portion of figure 1.2, cropped for readability.

CHAPTER 2

BACKGROUND

2.1 State of the Art

Jigsaw [12] is a visual analytics tool to explore and understand collections of text documents. Introduced in 2007, Jigsaw provides four visualizations, called views, to present different perspectives of the text and extracted entities. While MiddGuard is not a text based tool, the concepts that drive Jigsaw's views are of interest. Jigsaw views are coordinated and communicate with each other to update themselves. User interaction with one view updates the others. Multiple copies of each view can be created to reflect different perspectives of the data.

MiddGuard supports customizable views, rather a limited, baked in, set. Like Jigsaw, MiddGuard allows multiple copies of each type of view to be added to the investigation, each with a different view of the data. MiddGuard visualizations can be coordinated using a global state manager. Like the views themselves, coordinations need to be configured manually.

Improvise [14] enables users to build and browse highly-coordinated visualizations interactively. It allows users to load data, create views, specify visual abstractions, and establish coordinations. MiddGuard implements a similar build and browse system, where visualizations are assembled using a visual configuration and simultaneously rendered in the browser. Improvise places strong emphasis on complex, scriptable, and visually representable coordinations between views where MiddGuard relies on a global state manager, which developers can use as necessary to coordinate multiple views.

Eagleyes [3] is an interactive dataflow engine. Customizable modules can be connected to transform and query the data, and create visualizations. MiddGuard's dataflow model is similar, with extensions for a synchronous collaboration system that allows multiple analysts to work on the same data-flow graph at once.

CHAPTER 3

THE FRAMEWORK

3.1 Overview

MiddGuard is a web framework that enables software developers and analysts to create the tools to conduct complex, data-driven investigations. It provides a browser based front-end and web server back-end on top of which developers can build customizable tools specific to their data and investigation. Data is not uniform and investigating that data requires bespoke tools. MiddGuard, rather than implementing all the specific tools necessary to address all possible scenarios, provides the scaffolding on which developers can bring and build their own tools. The user interface and web server that MiddGuard implements create a simple environment to connect and use those tools transparently and efficiently.

MiddGuard breaks the operations of a visual analytics based investigation into two general steps: data transformation and data visualization. Data transformation involves any function on the data that results in a different, possible destructive, representation of the same dataset. These functions might involve reading, filtering, aggregating, annotating, or reformatting the dataset. As a general rule in MiddGuard, if the operation does not produce a visualization, it is a transformation. Data visualization takes place after the transformation steps, creating a visual, often interactive, representation of the dataset. By implementing these two steps, developers can extend MiddGuard to fit their data and investigation.

These extensions to MiddGuard are called modules. Modules are short pieces of code that often live in a single file. We divide modules into two types to represent data transformation and visualization respectively. The former is called an analytic module, while the latter is a visualization module. Modules implement a simple protocol that

MiddGuard is able to read and use to integrate module code into the framework. Analytic modules consist of code that runs solely on the web server. Visualization modules contain code that runs in the web browser to render DOM elements that make up its visualization.

Once the MiddGuard web server is running, investigators use these modules to build a data-flow graph. Modules are the graph's nodes. Edges between nodes describe data-flow from module to module. MiddGuard's web front-end comes with a graph editor that investigators use to add modules to the graph and connect modules to each other. Once added to the graph, a module has been instantiated in the context of that graph and is called a node. Analytic nodes can be chained from one to the next, making the graph a canvas to compose complex data transformations from multiple analytic modules, each with a singular task. Visualization modules can be connected to analytic nodes, which feed in data to create the visualization.

Although modules are customizable and can be written for a particular investigation, they are also reusable within the same graph, different graphs of the same investigation, or multiple different investigations. Modules' relationships to each other are managed by MiddGuard and defined by connections in the graph, rather than hardcoded into the modules themselves. For example, a developer could create a visualization module that renders a heatmap of two entities' activity moving around a city. The module would be written to accept input from Entity A, Entity B, and the data to draw the underlying city map. An investigator can connect the heatmap to any two cars, people, bikes, etc. from another dataset and render a heatmap with no additional development effort. As developers and investigators use MiddGuard, they build up a library of these reusable tools. In an investigation where time is a factor, being able to quickly plug in and test data transformations and visualizations promotes the investigator's efficiency.

3.2 Example: Using Tweets to Investigate Relationships

We can demonstrate the features of the data-flow model with a mock investigation. Our goal is to determine the relationships between multiple people: Alice, Bob, and Carlos. Our available data are lists of recent tweets each person has sent. We start by writing three similar modules that use a JavaScript library to access the Twitter API and download all of the tweets for each person, respectively. Between the three modules we only need to change the Twitter handle for which we are downloading tweets. We open MiddGuard and add a graph called "Tweet Relationships" then create nodes from these modules and add them to the graph. We can use the number of times one user mentions another (ex. Alice tweets the text "@Bob") as a metric for the relationship, so we write another module called "Mention Count" that extracts mentions from each tweet and creates a mapping from the Twitter user mentioned to the number of times mentioned throughout the incoming dataset. We add this module to the graph three times, and connect one "Mention Count" node to each of Alice, Bob, and Carlos's tweet download nodes. Already we are able to reuse "Mention Count" for each person's tweets. Finally, we visualize the relationships. We can use a force directed graph with a node for each person and strength of the edges proportional to the number of times one mentioned the other. Our visualization module, "Force Directed Graph" will take three inputs, one for each person. We create a node in the graph from the "Force Directed Graph" module and connect each of the outputs from our "Mention Count" nodes to the three inputs of "Force Directed Graph". Like the "Mention Count" module, "Force Directed Graph" is reusable and can be plugged into any three inputs.

At this point our graph is ready to produce data and a visualization. We work from the data entry points to the visualization, running the tweet download nodes, then the "Mention Count" nodes, then the "Force Directed Graph" visualization. The analytic nodes report when they are done so we know it is safe to run their dependents. Running the node “Force Directed Graph” renders the visualization next to our graph in the browser window.

3.3 Collaboration

MiddGuard not only enables single investigators to create and work with these tools, but also has built-in support for asynchronous and synchronous collaboration between teams of investigators. The framework includes user registration and authentication so multiple investigators can create accounts, log in, and work on the same investigations with the same graphs and access to the same data. All configuration and transformed data is persisted to a database, so investigators can log in and work with each other asynchronously, one picking up where the other left off. Investigators can also work together in real-time. As edits to the data-flow model are persisted to the database, they are pushed to all connected web clients and the user interface updates without a refresh to reflect those changes.

Since developers can collaborate to build the investigation, it follows that they should be able to collaborate to record conclusions from their analysis. MiddGuard comes with an observations tool for investigators to record and share observations about the analysis, creating a chronological record of what investigators saw in the data and when they saw it. An investigator of the tweet-based relationships from the previous example might record “Alice appears to have a close relationship with Bob. See the Force Directed Graph visualization in the Tweet Relationships graph.” Like graphs and data, these observations are persisted to the database and pushed to all connected clients in real-time.

CHAPTER 4

IMPLEMENTATION

4.1 Technology

MiddGuard builds on many open source software projects, some of which are instrumental to its implementation. Node.js, Knex.js, and Backbone.js make possible MiddGuard's structure and flexibility.

Node.js [8] is an asynchronous, event-driven JavaScript runtime built on Google Chrome's V8 JavaScript engine. The runtime is structured around an event loop where callbacks are registered and fired later in the program's life. Most I/O operations are performed indirectly through the event loop, so the process rarely blocks, allowing high concurrency and scalability. MiddGuard's server is implemented in JavaScript running on Node.js and its HTTP and WebSocket servers take advantage of the event loop. WebSocket is a bidirectional protocol for client-server communication. Since they are event-driven from the server, rather than driven by the HTTP request-response cycle, WebSockets are simpler to implement and deploy with Node.js than with many traditional servers for other languages and web frameworks.

Knex.js [9] is a SQL query builder with support for several relational databases including Postgres, MySQL, and SQLite. Knex exposes an API with function calls similar to SQL keywords that generate and execute SQL in the appropriate dialect for the connected database. It supports schema generation and returns the results of queries it runs on the database. MiddGuard uses Knex.js to simplify database connections for custom analytic modules and make the framework flexible to whichever database is best suited to the investigation. For the VAST 2015 Challenge Andrews and Billings used a Postgres database and connected over the network to collaborate from separate machines using the same database. SQLite is often preferable for single person investigations since it does not require running a database engine.

Backbone.js [2] is a front-end library designed to give structure to web applications. It consists of extendable Models, Views, and Collections, all of which we use to structure MiddGuard's front-end. Models manage data attributes and trigger events when that data changes. Collections are groups of related models. For example, there may be a Model called *Book* with attributes *title* and *author* and a Collection called *Library* that contains multiple books. Collections also emit events when updated. Both Models and Collections can persist their state to a web server that implements a REST API over HTTP. MiddGuard replaces the REST API persistence with a similar one implemented with WebSockets. Backbone.js Views are pieces of user interface. They render HTML in the browser and listen to events emitted from Models and Collections to update themselves. MiddGuard's front-end user interface is implemented using Backbone.js Views. Visualization modules extend a MiddGuard View, which is inherited from a Backbone.js View, to support View Reference Counting and automatic layout in MiddGuard's browser environment.

The browser, front-end, client, and other variants are all used to refer to the web browser, where MiddGuard's user interface lives. The browser is a non-traditional environment for visual analytics, which are often implemented as desktop applications to achieve higher performance through OS native code over JavaScript, which runs non-natively in the browser. However we were inspired by the expressiveness and ease with which we could create interactive visualizations with tools like D3.js [4], a JavaScript library for manipulating HTML, CSS, and SVG in the DOM based on data, and decided to implement MiddGuard as a web application.

4.2 Data-flow Model

MiddGuard's data-flow model allows arbitrary nodes, each with their own idea of input and output, to be chained together in a graph of data transformations and visualizations. Nodes are reusable units of code, so multiple instances of the same type of node, or module, can coexist in a single investigation. Connections between nodes allow data to pass between them.

4.2.1 Analytic Nodes

The first version of MiddGuard did not support preprocessing scripts like the ones we used in the VAST 2014 Challenge to transform data before visualizing it. These scripts did most of the work to setup and populate the database, so they were a major component missing from MiddGuard's idea of the analytic process. Nodes address this problem, creating a flexible representation within MiddGuard of the data processing phase of an investigation. In this section we will address the implementation of analytic nodes. Visualization nodes and their differences with respect to analytic nodes will be addressed in a subsequent section.

Analytic nodes are instances of modules, made unique from one another by the data they generate. MiddGuard is backed by a relational database where nodes are each assigned their own table. They use this table to persist their data. Nodes generate their data using their module's handler function, invoked from a button press in the user interface. Nodes can be created throughout an investigation and multiple nodes can be created from each module, so a node's table is created just before its handler is called.

Analytic modules specify a function that will be used to create all of its nodes' tables. That function is passed the table name to create and a Knex.js database connection. The function uses the connection and table name to generate the schema for its tables.

Nodes are not standalone scripts, they can work together to perform complex transformations, just as a developer might run one script after the other. Each node can output its data and receive input from other nodes. Inputs and outputs are passed into the node's handler function so it can use one to generate the other. The combination of input and output is a node's *context*. Creating a node's contextual output involves only the node itself. Every node has exactly one output, which is a database table. Other nodes that receive input from this one are simply querying that table. The output passed into a node's handler is a Knex.js database connection already assigned specifically to perform statements on the node's table. Creating a node's contextual inputs, however, requires analyzing its connections to other nodes.

4.2.2 Connections

Connections are a two-level protocol of node-to-node connections and intra-connection name mappings, used to determine the input passed into one node from another. Each node can have multiple named inputs, referred to as *input groups*. Each input group can have exactly one connection to another node, referred to as an *output node*. We refer to the parent node of an input group as the *input node*.

A mapping from an input group to an output node creates a mapping from that input group's name to the output node's table. This mapping is stored as a key-value pair where the key is the input group, and the value is the output table name.

When MiddGuard generates the contextual inputs for a node, the key value mapping allows developers to use the input group names they picked for the module to look up the values to access the input data. For the input context, the table name is translated to a combination of table name and a Knex.js accessor. The table name, while unnecessary for queries that only use that table, allows full flexibility for more advanced queries, such as table joins.

Input group to output node mappings tell us where a specific input's data lives, but not what the data looks like or how to refer to it. That is, we have the table to look in, but we don't know what its schema is and in particular, what its columns are named. Unless the only SQL we want to run is `SELECT * FROM 'output table'`, we need more information.

We address this at the second level of our connection protocol: intra-connection mappings within the input group to output node connection that identify the column names in the output table. This is another set of key-value pairs that map the names the input node has assigned to each attribute in an input group, to the corresponding column to access in the output table. When generating the contextual input for a node, this mapping is included for each input group. Like at the higher level of input group mappings, developers can look up the output table column name using a key they pick to represent that attribute.

Listing 4.1 shows an example connection configuration for a node called "Time by Day/Hour" that aggregates data by day of the week and hour of the day. The configuration for "Time by Day/Hour" has one input group, called "tweets", which is connected to the output node with id `9`. The `output_node` field serves as a foreign key referencing another row in the same table. The column-level connections between the input group and output node `9` are stored within the input group. Column mappings are stored in an array called `connections`. Each object within the `connections` array has a key `input` and a key `output`. The value of `input` is the name the input node has given to the column and the value of `output` is the name the output node has given to the column.

```
{
  "tweets": {
    "output_node": 9,
    "connections": [
      {
        "output": "handle",
        "input": "handle"
      },
      {
        "output": "tweet",
        "input": "tweet"
      },
      {
        "output": "timestamp",
        "input": "timestamp"
      }
    ]
  }
}
```

Figure 4.1: A node's connection configuration. The node has a connection from its input group "tweets" to the node with id 9.

4.2.3 Connection Storage

The connections generated by interaction with the graph editor are stored in MiddGuard's table of nodes as a JSON string in the same row as their corresponding input node. We considered multiple factors when deciding how to store connections in the database. We wanted a storage method that was portable, efficient, and convenient. Portability meant that we could easily export the configuration of nodes and connections to a text file so they could be read back in and the graph could be reassembled in a different system. Efficiency was determined by the number of database operations required to access the configuration. This was important since we have to read and write connections whenever a node is accessed or modified in the graph editor. Convenience meant that it was not overly complex to access and modify the connections from a programming perspective.

In addition to the JSON string storage method we implemented, we considered storing connections and nodes in separate tables, with either each column-level connection in its own row or each group of column-level connections in a row. The former performed no grouping amongst column mappings, while the latter grouped each input group's columns in a single row.

The first option (each column mapping has its own row) was appealing since it took advantange of the relational database, using foreign keys to associate column mappings with their nodes. However, this method is less portable since it requires multiple steps to export all the node information and their associated column mappings from the database to a structured text file. It is also less efficient since it requires reading a row from the database for every column mapping, in addition to a row for every node. Finally, it would be less convenient to develop with because it would require more queries to the database to obtain all the information to construct the graph than if we stored the connection information close to the nodes.

For similar reasons, we ruled out the second option of storing all column level connections in a row, grouped by their input group. This seemed like a poor compromise between storing all column mappings separately and storing all connection information with their nodes. We would lose the elegance of conforming to the facilities of a relational database, and still have to query the database multiple times to assemble a graph or export/import the data.

The implemented method of storing a node's connection in the same database table row as the node, in a JSON string, satisfied all our requirements. It is portable: JSON is common format to export human readable configuration. We can simply query all nodes and write out their metadata and JSON string as connections. It is efficient to access nodes and connections to construct a graph. All of a graph's nodes and connections can be accessed by reading n rows from the database, where n is the number of nodes in a graph. It is convenient to work with this format, since all the connection data for a node can be obtained by calling JavaScript's built-in `JSON.parse` method on a node's connections column.

4.2.4 Context Generation

A node's connections can be edited in the graph editor until runtime, when a node's handler function is executed. At this point, MiddGuard makes a query for the node in the database and retrieves its stored connections. Parsing the JSON string of connections lets MiddGuard access the mapping of input groups to output nodes and the mappings of column names between nodes. MiddGuard makes additional queries to determine the table names of connected output nodes. With just this information, MiddGuard can construct the dynamically generated context to pass into the handler function. Listing 4.2 is a sample of the context passed into one of the same "Time by Day/Hour" nodes whose connection was previously listed. At the top level it includes `inputs` and `table`. `inputs` is an object mapping each of the nodes input groups to data about the connected output node. Within `inputs` are: `knex`, an instance of the Knex.js SQL generator [9], used to access the table connected to an input group; `cols`, the column-level mapping between the node's input group and the connected output node's column names; and `tableName`, the name of the connected output node's table name. `cols` and `tableName` are meant to give access to the information available for more advanced queries, such as table joins.

The other top-level key in the context, `table`, gives access to the output table for this node. Like each input group in `inputs`, it has a `knex` accessor to generate SQL to query the database, and a `name`, which is the node's own table name. `table`, the output, doesn't need a column mapping, since the column names are the same as the ones the node has assigned itself as outputs.

Having to make additional queries to access output nodes' table names is a potential source of ineffiency not addressed by our connections storage format. A way around this would be to duplicate the table name each time it appears in a JSON string of connections. We decided against duplicating the data and in favor of making additional

```
{
  inputs: {
    tweets: {
      knex: [Object],            // database connection instance
      cols: {
        handle: 'handle',
        tweet: 'tweet',
        timestamp: 'timestamp'
      },
      tableName: 'download-tweets-danarsilver_1'
    },
    table: {
      knex: [Object],            // database connection instance
      name: 'aggregate-time_2'
    }
  }
}
```

Figure 4.2: The context passed into a "Time by Day/Hour" node's handler function.

database queries instead to avoid fragmenting the information, should the table name change. Should we need to update a node's table name, it can be done once for the row, rather than having to update the JSON string in all other connected nodes.

4.3 Visualization Nodes

Our model for visual analytics is incomplete without the visualizations themselves. We include visualizations in the data-flow model as their own nodes, which we refer to as *visualization nodes*. By integrating visualizations into the data-flow model, we can pass data transformed by the analytic nodes directly into our visualizations.

Visualization nodes, like analytic nodes, are added from modules in the graph editor. They have input groups that can be connected to output nodes, and column mappings between the two nodes on the ends of the connections. The primary difference between analytic nodes and visualization nodes is that the handler for a visualization node is a newly instantiated Backbone.js View [2] that is rendered in the web client.

The instantiated view for a visualization node has an instance method called `createContext`, which can be called to dynamically generate the context for a view, just as the MiddGuard generates the context for an analytic node on the back-end and passes it into the handler function. The context for a visualization node has the same structure as that of an analytic node, without the output, since a visualization node's output is a visualization, rather than a table of data.

Additionally, the Knex.js accessors for each input group are replaced with instances of Backbone.js Collections (with a new key aptly named `connection`), which can be used like the Knex.js accessor to access the data from the output node connected to that particular input. MiddGuard instantiates a Backbone Collection for each analytic node and a corresponding endpoint on the back-end to transmit the analytic node's data to the collection, as required by a visualization node.

Backbone.js and consequentially MiddGuard visualization nodes are not reliant on a particular library or framework to manipulate the DOM and render a visualization. This keeps MiddGuard flexible for any toolchain a developer wants to use to create visualizations.

Since only the representation of a visualization as a node and not the underlying structure of a visualization changed from the previous version of MiddGuard, View Reference Counting still works completely.

A potential improvement in the implementation of visualization nodes would be to only instantiate collections for analytic nodes that output to visualization nodes. Other nodes' data will never be accessed, so it is not necessary to maintain collections on the front-end or the endpoints on the back-end to transmit data to them. However, this is a low-priority improvement since there is little overhead in terms of memory usage to create an empty connection on the front-end or add the event listeners that handle data transmission to Node.js's event loop on the back-end.

4.4 Visual Configuration

A visual configuration abstracts away the details of the data-flow model within MiddGuard as descibed in the previous sections, and the independent implementation details of each node. A major motivation for MiddGuard is to facilite quick construction of complex visual analytic tools. MiddGuard's system for visually configuring nodes and connections allows investigators to quickly compose data transformations and visualizations. The visual component creates an expressive representation of the steps to reproduce a visualization.

The visual configuration interface is located in the three panels of the graph editor, seen in figure 4.3. The left panel, titled "Modules", lists all modules from which nodes can be instantiated. Clicking a module's button in the list adds a node of that type to the canvas in the middle panel.

The middle panel's canvas is a free-form space limited by the height of the window and a 500 pixel width constraint. Nodes, once added to the canvas, are outlined circles that can be rearranged and connected to one another. Analytic nodes and visualization nodes are outlined in blue and orange respectively, to make them easy to differentiate.

Figure 4.4 shows an analytic node with all its elements for user interaction in view. The cross in the upper left corner is used to drag the node around the canvas. Allowing nodes to be draggable is a simple solution to the problem of node layout. A downside is the additional effort and time required on the part of the user to position and reposition nodes in the canvas, but this is outweighed by both its simplicity to implement over a layout algorithm and the flexibility for the user to customize the graph view as best appeals to their idea of the investigation.

The "play" button, located in the top right of each node abstracts both analytic and visualization nodes' actions. In an analytic node clicking play calls its handler function. In a visualization node, the play button creates a new instance of a visualization. Press-

Figure 4.3: MiddGuard's graph editor user interface, open on a graph named "Compare Tweets". On the left, the modules panel lists all loaded modules, from which nodes can be created. In the center, the graph editor canvas has seven nodes initialized from their respective modules, and connections between the nodes. On the right, the detail panel shows the column mappings between the "Difference by Hour" node and its connections to two "Time by Day/Hour" nodes.

ing a visualization node's play button again removes that visualization from the browser window. Like the graph editors, visualizations stack horizontally in the browser window. The user can scroll through them from left to right.

While web scrolling is typically done vertically, we implemented view layout horizontally, since MiddGuard was designed to be used on the same system used for the VAST 2014 and VAST 2015 investigations. These investigations used a system of three 27 inch displays arranged side by side [1].

Each node contains two text indicators: in the center of the node in black is the node's module type. This is a visual indicator of the operation that will occur or visualization that will be rendered. Just below is the node's status indicator, one of "Not run", "In progress", or "Completed" in red, yellow, or green, respectively. The status indicates whether the handler function has already been invoked. Investigators ultimately use the node's status to determine when a visualization is able to be rendered in the browser. Only once all a visualization's dependent nodes have been run and have a status of "Completed", can a visualization be rendered.

The connections between nodes' inputs and outputs are key components in the visual configuration. They represent connecting code paths and passing data from one node to another. A connection can be created from one node to another by selecting one green input group indicator seen at the top of the node in figure 4.4 and one red output indicator like the one seen at the bottom of the same node. The selected input and output connectors are outlined with a black stroke. It is possible to connect a node's input to its own output, however this would result in no operation since the data required for the input would not exist at runtime. Since nodes can accept input from multiple outputs, hovering an input group indicator opens a tooltip with the name of the input group under the mouse to aid the investigator in creating the correct mapping.

Clicking a node widens its outline and opens the node's connections in the detail panel, seen on the right of figure 4.3. The detail panel lists each input group's column-level connections, grouped by that input group, and organized so output columns are on the left in red, and input columns are on the right in green. When a connection is made in the graph editor, MiddGuard attempts to automatically match columns based on the names. Any columns that don't match appear below the matched ones in gray. Columns can be connected manually in the same way as nodes: by clicking to select an output and an input to connect. The columns names in each group re-render to indicate the pairing after the connection has been made manually.

The similarity between interactions to edit connections at both the node and column level and the color coding of inputs and outputs in both the graph editor canvas and the detail panel is intentional, meant to make graph construction intuitive for an investigator. The goal of the visual configuration is to reduce the complications for an investigator to create a complex program. A familiar, easy to learn user interface promotes quick, simple development and reduces the cognitive load devoted to MiddGuard as a tool rather than the investigation itself.

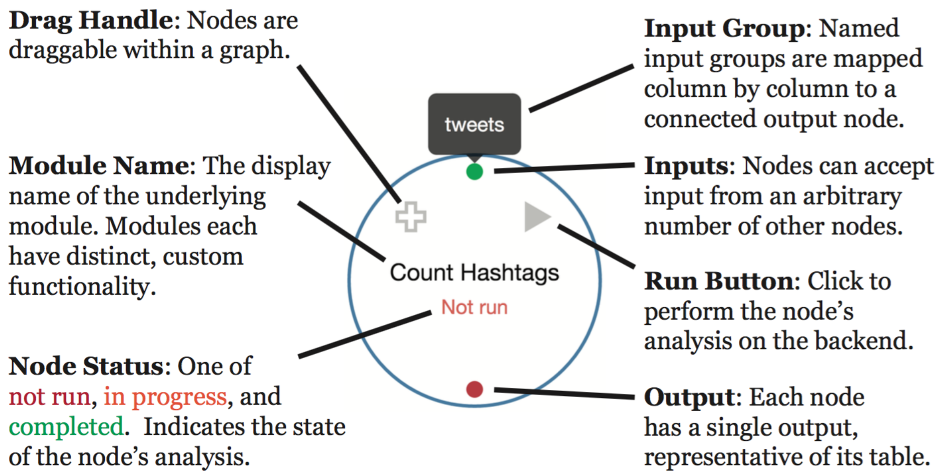

Figure 4.4: An annotated analytic node taken from the MiddGuard graph editor. Important features are annotated and the node's only input group, "tweets", is moused over to show its accompanying tooltip.

4.5 Extensibility

As mentioned before, the primary motivation for MiddGuard is to create a framework that allows investigators and developers to quickly and effectively create visual analytics tools. MiddGuard needs to be able to adapt to any investigation with any types of data and visualizations. To support any data or visualization, MiddGuard can register and load external code referred to as *modules*. The API for a module is the user interface for developers who work with MiddGuard, and need to quickly construct bespoke data transformations and visualizations.

Modules are short and designed to be written quickly. The code in figure 4.5 gives an example of an analytic module that aggregates tweets by the day of the week and hour of their timestamp. This module performs the final step of analysis in the "Compare Tweets" graph of figure 4.3 before data is fed into the visualization. This module is very small, but powerful when instantiated as a node and used in combination with other data transformations. Since MiddGuard modules are just Node.js modules, they can grow as needed, expanding to multiple files as necessary to organize code.

An analytic module can be as simple as one JavaScript file that exports the five objects exported in figure 4.5. Those exports declare inputs, outputs, how to create a table for the module (`createTable`), and what to do when the module is run (`handle`). The display name is a "pretty" version of the module's name in the file system used to label modules and nodes in the user interface.

The `createTable` and `handle` functions are passed in the node's dynamically generated context based on its input connections and the name MiddGuard gives its table as described in the previous sections on connections and context. The `handle` function in figure 4.5 demonstrates the use of its context. It uses `context.inputs.tweets.knex` (lines 28 and 38) to access the table where its input group *tweets* is stored, and `context.inputs.tweets.cols.timestamp` (line 29) to get the name of the timestamp column for that same input group, which it later (line 41) uses to access the timestamp attribute for each tweet. On line 48, the function uses `context.table.knex` to write data to its output.

Visualization modules are simpler than analytic modules in terms of exports, but often more complicated since they have to render a visualization in the browser. It is useful to separate the code for a visualization module into a main file, *index.js*, which exports its configuration and a directory, *static*, at the same level, which contains the front-end visualization code.

Figure 4.6 is the configuration code for a sample visualization module called "Hours Heatmap". Like an analytic module, it exports inputs, outputs, and a display name. To render on the front-end, it also exports (from top to bottom) that it is a visualization, the location of its front-end files (in this case, an adjacent directory called *static*), the JavaScript and CSS files required on the front-end, and the name of the main view to initialize and render on the front-end when MiddGuard loads the visualization. The main view must be a Backbone View included in one of the JavaScript files.

After a module is written, it can be added to a MiddGuard web server (Figure 4.7). Like the modules, a MiddGuard-based server is intended to be short and easy to work with. Only a few lines are required to create the server and start listening for web client connections. Adding a module is a single function call to the server's `module` function.

Using a simple function call to load modules, rather than discovering them automatically, has a few advantages: it makes the investigation explicit about its module dependencies, raising an error immediately if a required module is missing; it allows a specific server to use different names (the first parameter passed to `module`) to identify the module in case of a naming conflict between two or more modules; and it allows the user to install and require a module from Node.js's package manager, rather than relatively from the file system.

4.6 Real-time Collaboration

MiddGuard supports asynchronous and synchronous collaboration between multiple developers. Asynchronous collaboration is common in a web application. For example, User A makes changes, which are persisted to a data store. User B logs in some time later and the changes User A made are loaded from the database so User B can view them.

Synchronous collaboration is more difficult to implement. Web application communications are largely based on the HTTP protocol. Data is transferred from the web server to the client in an HTTP session, which is made up of a request from the client and a response from the server. The client must initiate an HTTP request before the server can send data. This is problematic for real-time communications. Like in the asynchronous example, User A might make a change, which should be immediately pushed to all other connected clients. User A can make an HTTP request to tell the server about the change, but there is no way for the server to tell other clients about the change immediately. With HTTP, User B must explicitly request the update, which requires either knowing when to check for an update (unreasonable) or continuously polling the server for changes (inefficient).

WebSockets help solve real-time communications, and are implemented in place of HTTP for all of MiddGuard's server-client communications after a user is authenticated and logged in. WebSockets is a bidirectional event-driven communication protocol designed for browers and servers to exchange data without relying on HTTP requests and responses [10]. WebSockets are layered on TCP. The connection from the browser to the server is initiated with the HTTP Upgrade header and a client-server handshake after the browser has received a traditional HTTP response from the server with the code to perform the Upgrade request [7].

The MiddGuard server registers WebSocket event handlers for its internal components and for nodes' data. Data on the front-end is structured using Backbone.js Models and Collections, which traditionally use HTTP to perform create, read, update, and delete (CRUD) operations. We use third-party libraries, Backbone.ioBind and Backbone.ioSync, to replace the HTTP requests with a similar protocol using WebSocket events. A HTTP request `POST /graphs` becomes `socket.emit( 'graphs:create', data)`. Emitted from the browser, these events offer no real advantage over their corresponding HTTP requests. The use case for WebSockets is emitting events and data from the server to the client, which is impossible over HTTP. With the connection open, we can send events from the server to the client to create, update, and delete (the server does not need to read data from the client) Backbone Models and Collections whenever the data change on the server, enabling real-time updates and collaboration for clients.

```
var _ = require('lodash');
var Promise = require('bluebird');
var moment = require('moment');

exports.inputs = [
  {name: 'tweets', inputs: ['handle', 'tweet', 'timestamp']}
];

exports.outputs = [
  'handle',
  'day',
  'hour',
  'count'
];

exports.displayName = 'Time by Day/Hour';

exports.createTable = function(tableName, knex) {
  return knex.schema.createTable(tableName, function(table) {
    table.string('handle');
    table.integer('day');
    table.integer('hour');
    table.integer('count');
  });
};

exports.handle = function(context) {
  var tweets = context.inputs.tweets,
      timestampCol = context.inputs.tweets.cols.timestamp,
      week = [];

  _.range(24).forEach(function(hour) {
    _.range(7).forEach(function(day) {
      week.push({day: day, hour: hour, count: 0});
    });
  });

  return tweets.knex.select('*')
  .then(function(tweets) {
    tweets.forEach(function(tweet) {
      var m = moment(tweet[timestampCol]),
          day = +m.format('d'),
          hour = +m.format('H');

      _.find(week, {day: day, hour: hour}).count++;
    });

    return context.table.knex.insert(week);
  });
};
```

Figure 4.5: Code for an example analytic module.

```
var path = require('path');

exports.inputs = [
  {name: 'hours', inputs: ['day', 'hour', 'count1', 'count2']}
];

exports.outputs = [];

exports.displayName = 'Hours Heatmap';

exports.visualization = true;

exports.static = path.join(__dirname, 'static');

exports.js = [
  'hours-heatmap-view.js'
];

exports.css = [
  'hours-heatmap.css'
];

exports.mainView = 'HoursHeatmapView';
```

Figure 4.6: Contents of the main configuration file, *index.js*, for an example visualization module, “Hours Heatmap”.

```javascript
var middguard = require('middguard');

var app = middguard({
  // database
  'knex config': require('./knexfile'),

  // sessions
  'secret key': process.env.SECRET_KEY || 'keep me secret'
});

// Hours Heatmap Visualization Module
app.module('hours-heatmap', require.resolve('./hours-heatmap'));

// Time by Day/Hour Analytic Module
app.module('aggregate-time', require.resolve('./aggregate-time'));

// Start the server
var port = process.env.PORT || 3000;
app.listen(port, function () {
  console.log('Listening on port %d...', port);
});
```

Figure 4.7: Code for an investigation's MiddGuard-based server. It creates an instance of the MiddGuard server, passing in the database configuration from a Knexfile [9] and a secret key to encrypt authenticated session data. This investigation uses the two modules from figures 4.5 and 4.6 and registers them with calls to `app.module`. Finally the server starts listening for connections on port 3000.

CHAPTER 5

DISCUSSION

5.1 Use Case

We constructed a small investigation into Twitter data to help implement and test MiddGuard as we implemented the framework. Using tweets from two users' timelines, we wanted to determine who tweets more each hour of each day of the week.

To find an answer we wrote four analytic modules and two visualization modules. Our first two analytic modules accessed the Twitter API to download tweets from the two subjects, "@DanaRSilver" and "@jack". These are also the names of the respective modules. Next, we wrote "Time by Day/Hour", which uses tweets' timestamps to aggregate the them by day of the week and hour of day. Our last analytic module, "Difference by Hour", computes the difference between counts for each combination of day and hour and groups the two counts into a single table. We created a new graph and connected the "@DanaRSilver" and "@jack" nodes each to a "Time by Day/Hour" and fed those into a "Difference by Hour" node. Figure 5.1 shows the complete graph.

Since our goal was to figure out who tweets more at each combination of hour of the day and day of the week, we wrote a visualization called "Hours Heatmap", a bubble chart with hours on the x axis and days on the y axis (Figure 5.2). Two circles, or bubbles, are drawn at entry in the chart, one for each person. The circles' radii are mapped to the number of times the corresponding person tweeted that hour and day. Mousing over a pair of circles adds a tooltip with the exact count.

From the "Hours Heatmap" visualization we are able to answer our question. We can look to any particular day and hour and see who tweets more. Wednesday at 12pm, for example, Dana tweets more than Jack. Dana tweeted nine times and Jack tweeted twice. We are also able to identify some patterns in the tweets. Both people rarely tweet

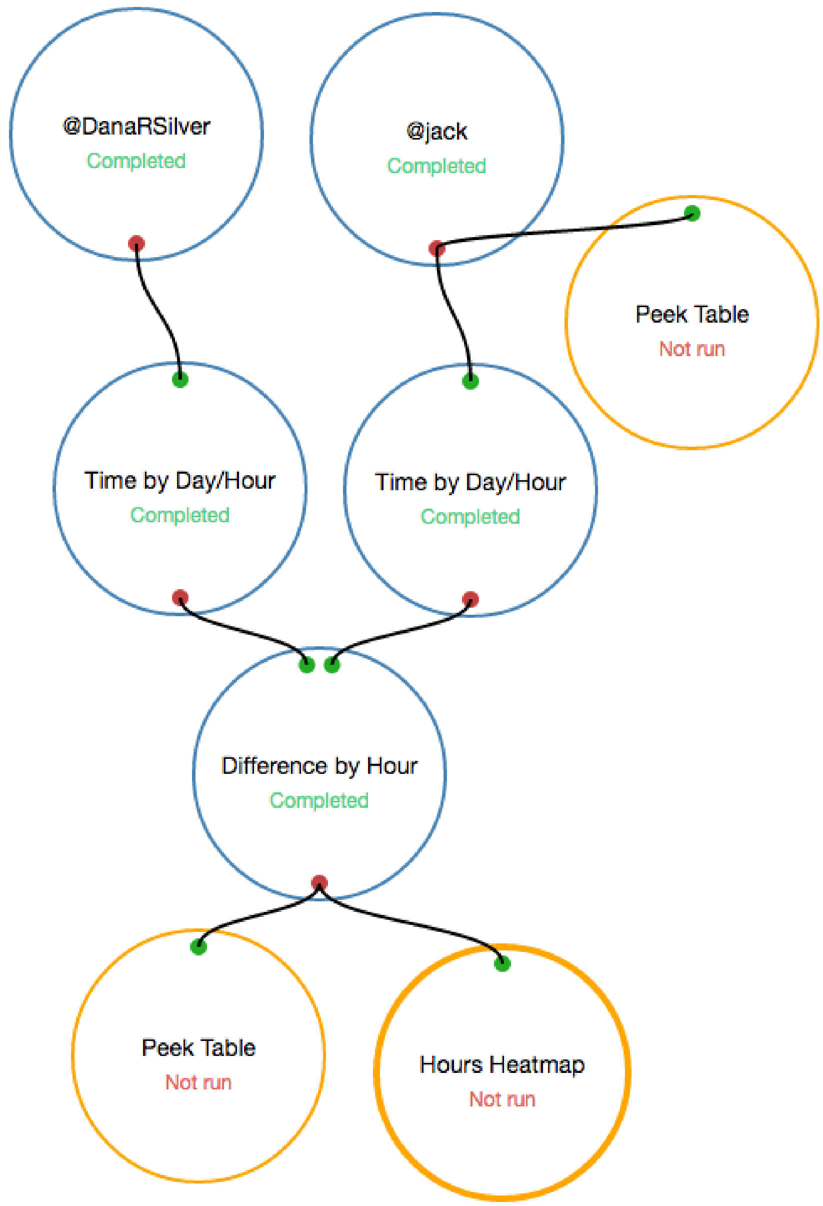

Figure 5.1: The complete graph from the mock investigation used to develop and test MiddGuard.

late at night, and never between 4am and 6am. Jack is more active on Saturday than Dana and both get a late start on the weekend.

While we were investigating our primary question, we wanted to look at the data we received from the Twitter API as well, to make our investigation more transparent, and to test that we had downloaded tweets correctly without having to work with the database

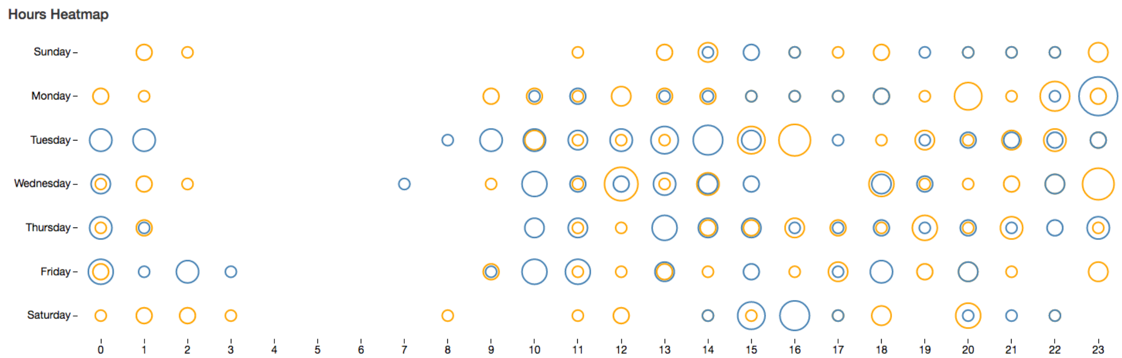

Figure 5.2: The "Hours Heatmap" visualization from the mock investigation used to develop and test MiddGuard. @DanaRSilver's tweets are orange, @jack's are blue. Circle radii are mapped to the number of times each person tweeted in that hour and day of the week.

outside MiddGuard. We wrote a visualization "Peek Table" that takes any input and renders it as a table. We hooked this up to both the "@DanaRSilver" and "@Jack" modules and could immediately tell that our download had worked as intended. Since we could see the text of the tweets in this alternate view of the data, we could also see that Jack retweets much more often than Dana.

5.2 Areas for Improvement

The mock investigation into @DanaRSilver and @jack's tweets revealed two areas for improvement in MiddGuard. The first is that modules only can change context between nodes with respect to the incoming and outgoing data. We have two almost identical modules to download @DanaRSilver's tweets and @jack's tweets. The only difference is the Twitter handle accessed. When one of our goals is reuse of the data transformation logic, it does not make sense to repeat logic just to change a variable. We could improve on this by allowing developers to define variables that can change from node to node and create an interface for investigators to define that variable for the node. This would have allowed us to write one module that downloads tweets, create two nodes from it, and pass “@DanaRSilver” and “@jack” in as variables.

Developing the modules was challenging, since it was hard to test if the transformation logic worked. We eventually created the “Peek Table” visualization module to check the table contents, however this required creating multiple nodes and running the user interface to test. There is no way to remove data from a node, so if the transformation was applied and saved data incorrectly, additional nodes would have to be created to test the module again. This issue could be solved with a procedure to pass data through a module without creating a node in the user interface, and without persisting that data to the node’s table. Besides simpler development, this solution would make it substantially easier to write tests for modules, which in MiddGuard’s current state would require creating a database, starting the web server, and manipulating the user interface in a web browser.

Outside the areas of improvement discovered during the use case, MiddGuard could be improved to better incorporate visualization nodes into the data flow. Visualization nodes should be able to modify data in the database and have their own data output to support brushing, linking, and detail in the browser. Visualization nodes need to be able to modify data, or at least report user interactions so the server can respond to them. This enables operations like those used in the VAST 2014 Challenge, where we selected car destinations to associate with points of interest on a map. Like analytic nodes take in data to transform, a new type of node, “Event Nodes” could take in events and associated data (like a click and the destination under the mouse) and perform a data transformation to respond to that event.

A second output, for visualization nodes to output a subset of the data they take in, could support brushing, linking, and detail interactions within the data-flow model, rather than in a separate and opaque global state with no visual representation. Other visualizations would take that output as their input, using it to render their own visualization. For example, the “Hours Heatmap” from the mock tweet investigation could output the data from a selected day of the week, which would be read by a bar chart visualization and used to render a bar chart of cumulative tweets per day of the week. As the selected day changes in the “Hours Heatmap”, the bar chart would receive updated data and rerender. Since analytic nodes output data following some transformation step, it is intuitive to the data-flow model that visualization nodes do the same. Building interactions into the data-flow model increases the transparency and reproducability of the investigation.

CHAPTER 6

CONCLUSION

MiddGuard is an effective web framework to create complex visual analytics tools quickly. Its data-flow model and its visual representation allows investigators to see exactly what steps were applied to a dataset to produce a visualization, increasing transparency throughout the investigation and reproducability of results afterwards. While the previous version of MiddGuard required developers to load and preprocess data with scripts external to the framework, our use case demonstrated the ease with which we can write those scripts into the current version of MiddGuard and represent them in the data-flow model.

Connections between nodes are a useful abstraction to simulate data flowing through the graph and generate the contextual input required to actually send data from node to node, while persisting it to a database. The model for extensiblity, analytic and visualization modules, is able to encompass operations that take place in the back-end and front-end and offer plug-and-play capabilities without sacrificing flexibility or developer choices for module implementation. The synchronous communication protocol implemented over WebSockets allows developers to work together to develop tools and share results.

By abstracting away the details required to structure components and communicate data to a simple graph builder and built-in tooling, MiddGuard lets analysts focus on writing and using the tools they need to analyze data in a timely manner. When taken together, these features make MiddGuard a novel tool for visual analytics.

We plan on making MiddGuard open source software. Open sourcing MiddGuard will encourage contributions to the core framework from outside collaborators. As other users investigate data with MiddGuard and write their own modules, they can contribute these back to the community to create a resource of analytic and visualization modules.

APPENDIX A

AN INVESTIGATION WITH MIDDGUARD

The code in Appendix A implements the mock investigation described in the use case. At the top level are *index.js* and *knexfile.js*, which setup the web server and configure the database connection respectively. The other ten files implement the analytic and visualization modules that we used to conduct the investigation. The analytic modules each consist of a single file, an *index.js* in a subdirectory of the project. The visualization modules include an *index.js* to configure the module, and a JavaScript and CSS file each to render their visualizations in the browser.

examples/simple/index.js

```
var middguard = require('../..');

var app = middguard({
  // database
  'knex config': require('./knexfile'),

  // sessions
  'secret key': process.env.SECRET_KEY || 'major     '
});

app.module('read-tweets', require.resolve('./read-tweets'));
app.module('count-hashtags', require.resolve('./count-hashtags'));
app.module('read-hashtags', require.resolve('./read-hashtags'));
app.module('hashtags-table', require.resolve('./hashtags-table'));
app.module('peek-table', require.resolve('./peek-table'));
app.module('hours-heatmap', require.resolve('./hours-heatmap'));
app.module('download-tweets-danarsilver',
  require.resolve('./download-tweets-danarsilver'));
app.module('download-tweets-jack', require.resolve('./download-tweets-jack'));
app.module('aggregate-time', require.resolve('./aggregate-time'));
app.module('difference', require.resolve('./difference'));
app.module('mean-difference', require.resolve('./mean-difference'));

var port = process.env.PORT || 3000;
app.listen(port, function () {
  console.log('Listening on port %d...', port);
});
```

examples/simple/knexfile.js

```
module.exports = {
  client: 'sqlite3',
  connection: {
    filename: 'simple.db'
  },
  pool: {
    min: 0,
    max: 1,
    afterCreate: function(conn, cb) {
      conn.run('PRAGMA foreign_keys = ON', cb);
    }
  }
}
```

examples/simple/download-tweets-danarsilver/index.js

```
var Promise = require('bluebird');
var fs = Promise.promisifyAll(require('fs'));
var path = require('path');
var _ = require('lodash');
var Twitter = require('twitter');

exports.inputs = [];

exports.outputs = [
  'handle',
  'tweet',
  'timestamp'
];

exports.displayName = "@DanaRSilver";

var client = new Twitter({
  consumer_key: 'fEYwq7R6fP7np546j799dMXJj',
  consumer_secret: '5
      pAk0lrSEZ1hrhbnRG6pJdcQIYkKMtIFNPvsyzV8jjyuhSnOCl',
  access_token_key: '354037431-
      sd7fd6inZSXWaw9LmC3gmFfaHWx6p8UJq8JUaPDM',
  access_token_secret: '
      B8clfzqPuJqUWKnSGTsEpV3eF1Y35RiIw7HI6YiMSOleS'
});

client = Promise.promisifyAll(client);

exports.handle = function(context) {
  var params = {screen_name: 'DanaRSilver', count: 200};
  return client.getAsync('statuses/user_timeline', params
      )
  .spread(function(tweets, response) {
    tweets = tweets.map(function(tweet) {
      return {
        handle: tweet.user.screen_name,
        tweet: tweet.text,
        timestamp: new Date(tweet.created_at)
```

examples/simple/download-tweets-jack/index.js

```javascript
var Promise = require('bluebird');
var fs = Promise.promisifyAll(require('fs'));
var path = require('path');
var _ = require('lodash');
var Twitter = require('twitter');

exports.inputs = [];

exports.outputs = [
  'handle',
  'tweet',
  'timestamp'
];

exports.displayName = "@jack";



var client = new Twitter({
  consumer_key: 'fEYwq7R6fP7np546j799dMXJj',
  consumer_secret: '5pAk0lrSEZ1hrhbnRG6pJdcQIYkKMtIFNPvsyzV8jjyuhSnOCl',
  access_token_key: '354037431-sd7fd6inZSXWaw9LmC3gmFfaHWx6p8UJq8JUaPDM',
  access_token_secret: 'B8clfzqPuJqUWKnSGTsEpV3eF1Y35RiIw7HI6YiMSOleS'
});

client = Promise.promisifyAll(client);

exports.handle = function(context) {
  var params = {screen_name: 'jack', count: 200};
  return client.getAsync('statuses/user_timeline', params)
  .spread(function(tweets, response) {
    tweets = tweets.map(function(tweet) {
      return {
        handle: tweet.user.screen_name,
      };
    });

    return context.table.knex.insert(tweets);
  });
};

exports.createTable = function(tableName, knex) {
  return knex.schema.createTable(tableName, function(table) {
    table.string('handle');
    table.string('tweet');
    table.dateTime('timestamp');
  });
};
```

```
        tweet: tweet.text,
        timestamp: new Date(tweet.created_at)
      };
    });

    return context.table.knex.insert(tweets);
  });
};

exports.createTable = function(tableName, knex) {
  return knex.schema.createTable(tableName, function(
      table) {
    table.string('handle');
    table.string('tweet');
    table.dateTime('timestamp');
  });
};
```

examples/simple/aggregate-time/index.js

```
var _ = require('lodash');
var Promise = require('bluebird');
var moment = require('moment');

exports.inputs = [
  {name: 'tweets', inputs: ['handle', 'tweet', 'timestamp
      ']}
];

exports.outputs = [
  'handle',
  'day',
  'hour',
  'count'
];

exports.displayName = 'Time by Day/Hour';

exports.createTable = function(tableName, knex) {
  return knex.schema.createTable(tableName, function(
      table) {
    table.string('handle');
    table.integer('day');
    table.integer('hour');
    table.integer('count');
  });
};

exports.handle = function(context) {
  var tweets = context.inputs.tweets,
      timestampCol = context.inputs.tweets.cols.timestamp
          ,
      week = [];

  _.range(24).forEach(function(hour) {
    _.range(7).forEach(function(day) {
      week.push({day: day, hour: hour, count: 0});
    });
```

```
  });

  return tweets.knex.select('*')
  .then(function(tweets) {
    tweets.forEach(function(tweet) {
      var m = moment(tweet[timestampCol]),
          day = +m.format('d'),
          hour = +m.format('H');

      _.find(week, {day: day, hour: hour}).count++;
    });

    return context.table.knex.insert(week);
  });
};
```

examples/simple/difference/index.js

```
var _ = require('lodash');
var Promise = require('bluebird');

exports.inputs = [
  {name: 'tweets1', inputs: ['day', 'hour', 'count']},
  {name: 'tweets2', inputs: ['day', 'hour', 'count']}
];

exports.outputs = [
  'day',
  'hour',
  'count1',
  'count2',
  'difference'
];

exports.displayName = 'Difference by Hour';

exports.createTable = function(tableName, knex) {
  return knex.schema.createTable(tableName, function(
      table) {
    table.integer('day');
    table.integer('hour');
    table.integer('count1');
    table.integer('count2');
    table.integer('difference');
  });
};

exports.handle = function(context) {
  var tweets1 = context.inputs.tweets1,
      tweets2 = context.inputs.tweets2,
      week = [];

  return Promise.join(tweets1.knex.select('*'), tweets2.
      knex.select('*'),
  function(tweets1, tweets2) {
    _.range(24).forEach(function(hour) {
```

```
    _.range(7).forEach(function(day) {
      var count1 = _.find(tweets1, {hour: hour, day:
          day}).count;
      var count2 = _.find(tweets2, {hour: hour, day:
          day}).count;
      week.push({
        day: day,
        hour: hour,
        count1: count1,
        count2: count2,
        difference: Math.abs(count1 - count2)
      });
    });
  });

  return context.table.knex.insert(week);
 });
};
```

examples/simple/hours-heatmap/index.js

```
var path = require('path');

exports.inputs = [
  {name: 'hours', inputs: ['day', 'hour', 'count1', '
      count2']}
];

exports.outputs = [];

exports.displayName = "Hours Heatmap";

exports.visualization = true;

exports.static = path.join(__dirname, 'static');

exports.js = [
  "hours-heatmap-view.js"
];

exports.css = [
  "hours-heatmap.css"
];

exports.mainView = 'HoursHeatmapView';
```

examples/simple/hours-heatmap/static/hours-heatmap-view.js

```
var middguard = middguard || {};

(function() {
  var HoursHeatmapView = middguard.View.extend({
    id: 'hours-heatmap',

    className: 'list-unstyled middguard-module',

    tagName: 'div',

    events: {
      'mouseover .dayhour': 'showInputTooltip',
      'mouseout .dayhour': 'hideInputTooltip'
    },

    template: _.template(
      '<h4>Hours Heatmap</h4>' +
      '<div class="heatmap-tooltip">' +
        '<span class="count1"></span>' +
        '<span class="count2"></span>' +
      '</div>'
    ),

    initialize: function() {
      this.context = this.createContext();
      console.log(this.context);

      var tableName = this.context.inputs.hours.tableName
          ;
      this.listenTo(this.context.inputs.hours.collection,
          'reset', this.render);

      this.fetch(tableName, {reset: true, data: {}});
    },

    render: function() {
      this.$el.html(this.template());
      this.$el.css('position', 'relative');

      var data = this.context.inputs.hours.collection.map
          (function(hours) {
        return _.clone(hours.attributes);
      });

      var margin = {top: 0, left: 90, right: 0, bottom:
          20};

      var rowHeight = 60,
          height = 7 * rowHeight - margin.top - margin.
              bottom,
          colWidth = 60,
          width = colWidth * 24 - margin.left - margin.
              right;

      var week = ["Sunday", "Monday", "Tuesday", "
          Wednesday", "Thursday", "Friday", "Saturday"];

      var x = this.x = d3.scale.linear()
          .domain([0, 23])
          .range([colWidth / 2, width - colWidth / 2]);

      var y = this.y = d3.scale.linear()
          .domain([0, 6])
          .range([rowHeight / 2, height - rowHeight / 2])
              ;

      var xAxis = d3.svg.axis()
          .scale(x)
          .ticks(24)
          .orient('bottom');

      var yAxis = d3.svg.axis()
          .scale(y)
          .orient('left')
          .ticks(6)
          .tickFormat(function(d) {
            return week[d];
```

```
        });

    var size = this.size = d3.scale.sqrt()
        .domain([0, d3.max(data, function(d) { return Math.max(d.count1, d.count2); })])
        .range([0, 25]);

    var svg = d3.select(this.el).select('svg')[0][0]
            ? d3.select(this.el).select('svg')
            : d3.select(this.el).append('svg');

    svg = svg
        .attr('width', width + margin.left + margin.right)
        .attr('height', height + margin.top + margin.bottom)
      .append('g')
        .attr('transform', 'translate(' + margin.left + ',' + margin.top + ')')

    var circles = svg
      .selectAll('g.dayhour')
        .data(data)
      .enter().append('g')
        .attr('class', 'dayhour')
        .attr('transform', function(d, i) {
          return 'translate(' + x(d.hour) + ',' + y(d.day) + ')';
        });

    circles.append('circle')
        .attr('r', function(d) { return size(d.count1); })
        .style('fill', 'transparent')
        .style('stroke-width', 2)
        .style('stroke', 'orange');

    circles.append('circle')
        .attr('r', function(d) { return size(d.count2); })
        .style('fill', 'transparent')
        .style('stroke-width', 2)
        .style('stroke', 'steelblue');

    svg.append('g')
        .attr('class', 'y axis')
        .call(yAxis);

    svg.append('g')
        .attr('class', 'x axis')
        .attr('transform', 'translate(0,' + height + ')')
        .call(xAxis);

    return this;
  },

  showInputTooltip: function(event) {
    var tooltip = d3.select('.heatmap-tooltip');

    var d = d3.select(event.target).datum();
    tooltip.select('.count1').text(d.count1);
    tooltip.select('.count2').text(d.count2);

    var bounds = event.currentTarget.getBoundingClientRect(),
        inputRadius = 5,
        tooltipWidth = parseFloat(tooltip.style('width')) / 2,
        tooltipHeight = parseFloat(tooltip.style('height')) + 5;

    tooltip
      .style('left', (this.x(d.hour) + 65) + 'px')
      .style('top', (this.y(d.day) - this.size(Math.max(d.count1, d.count2)) - 10) + 'px')
      .style('visibility', 'visible');
  },
```

```
    hideInputTooltip: function() {
      d3.select('.heatmap-tooltip')
          .style('visibility', 'hidden');
    }
  });

  middguard.addModule('HoursHeatmapView',
      HoursHeatmapView);
})();
```

examples/simple/hours-heatmap/static/hours-heatmap.css

```
.axis line {
  fill: none;
  stroke: #000;
  shape-rendering: crispEdges;
}

.axis path {
  fill: none;
  stroke: none;
}

span.count1, span.count2 {
  font-size: 16px;
}

span.count1 {
  color: orange;
  padding-right: 10px;
}

span.count2 {
  color: steelblue;
}

.heatmap-tooltip {
  position: absolute;
  padding: 10px;
  border-radius: 5px;
  font-size: 12px;
  line-height: 1.4;
  text-align: center;
  color: #fff;
  background-color: rgba(0, 0, 0, 0.7);
  visibility: hidden;
  z-index: 1;
}

.heatmap-tooltip:after {
```

```
  position: absolute;
  top: 100%;
  left: 50%;
  margin-left: -5px;
  width: 0;
  border-top: 5px solid rgba(0, 0, 0, 0.7);
  border-right: 5px solid transparent;
  border-left: 5px solid transparent;
  content: " ";
}
```

examples/simple/peek-table/index.js

```
var path = require('path');

exports.inputs = [
  {name: 'table', inputs: ['col1', 'col2', 'col3', 'col4'
      ]}
];

exports.outputs = [];

exports.displayName = "Peek Table";

exports.visualization = true;

exports.static = path.join(__dirname, 'static');

exports.js = [
  "peek-table-view.js"
];

exports.css = [
  "peek-table.css"
];

exports.mainView = 'PeekTableView';
```

examples/simple/peek-table/static/peek-table-view.js

```
var middguard = middguard || {};

(function() {
  var PeekTableView = middguard.View.extend({
    id: 'hashtags-table',

    className: 'list-unstyled middguard-module',

    tagName: 'table',

    template: _.template(
      '<th><tr><td><%- col1 %></td><td><%- col2 %></td><
          td><%- col3 %></td><td><%- col4 %></td></tr></
          th>'
    ),

    rowTemplate: _.template(
      '<tr><td><%- col1 %></td><td><%- col2 %></td><td
          ><%- col3 %></td><td><%- col4 %></td></tr>'
    ),

    initialize: function() {
      this.context = this.createContext();

      var collection = this.context.inputs.table.
          collection;

      var tableName = this.context.inputs.table.tableName
          ;
      this.listenTo(collection, 'reset', this.addAllRows)
          ;

      this.fetch(tableName, {reset: true, data: {}});
    },

    render: function() {
      var cols = this.context.inputs.table.cols;

      this.$el.html(this.template({
        col1: cols.col1,
        col2: cols.col2,
        col3: cols.col3,
        col4: cols.col4
      }));

      return this;
    },

    addAllRows: function() {
      var collection = this.context.inputs.table.
          collection;
      collection.each(this.addOneRow, this);
    },

    addOneRow: function(row) {
      var cols = this.context.inputs.table.cols;

      console.log(cols, row)

      this.$el.append(this.rowTemplate({
        col1: row.get(cols.col1),
        col2: row.get(cols.col2),
        col3: row.get(cols.col3),
        col4: row.get(cols.col4)
      }));
    }
  });

  middguard.PeekTableView = PeekTableView;
  middguard.addModule('PeekTableView', PeekTableView);
})();
```

examples/simple/peek-table/static/peek-table.css

```
#hashtags-table {
  padding: 10px;
}

#hashtags-table tr:nth-child(even) {
  background-color: #e5e5e5;
}

#hashtags-table tr {
  padding: 4px;
}
```

APPENDIX B

CORE CODE FROM THE MIDDGUARD FRAMEWORK

The code in Appendix B is the core of MiddGuard's data-flow model and front-end visualization loading/unloading. Additional code, style sheets, and HTML templates that make up complete the MiddGuard framework have been omitted from this listing. *index.js*, *middguard/application.js*, and *middguard/middguard.js* implement the web server from which all MiddGuard investigations (including that in Appendix A) are instantiated. Code in *middguard/socket* implements the server side of MiddGuard's WebSocket protocol. *middguard/models* and *middguard/migrations* contain data models and schema to persist graphs and data to the database. *static* contains all front-end code for MiddGuard's client-side environment.

index.js

```
'use strict';

module.exports = require('./middguard/middguard');
```

middguard/application.js

```
'use strict';

/**
 * Module dependencies.
 * @private
 */

var path = require('path');
var http = require('http');

var bodyParser = require('body-parser');
var cookieParser = require('cookie-parser');
var express = require('express');
var socketio = require('socket.io');
var ios = require('socket.io-express-session');
var session = require('express-session');
var KnexSessionStore = require('connect-session-knex')(
    session);
var _ = require('lodash');

/**
 * Application prototype methods to extend
 * the express application prototype.
 */

var app = exports = module.exports = {};


app.middguardInit = function () {
  this.middguardExpressMiddleware();
```

```javascript

  var server = http.createServer(this);
  this.set('http server', server);

  var io = socketio(server);
  this.set('io', io);
  this.middguardSocketMiddleware();

  io.on('connection', require('./socket'));

  require('./routes')(this);
};

/**
 * Setup the express middleware for MiddGuard.
 *
 * @private
 */

app.middguardExpressMiddleware = function middguardExpressMiddleware() {
  this.use('/static', express.static(path.join(__dirname, '..', 'static')));

  var knex = require('knex')(this.get('knex config'));
  var sessionStore = new KnexSessionStore({knex: knex});
  this.set('sessionStore', sessionStore);

  // Set up ORM middlware
  require('./config/bookshelf')(this);

  this.use(cookieParser(this.get('secret key')));
  this.use(bodyParser.urlencoded({extended: true}));
  this.use(bodyParser.json());

  this.set('session', session({
    store: sessionStore,
    secret: this.get('secret key'),
    resave: true,
    saveUninitialized: true,
    cookie: {maxAge: 7 * 24 * 60 * 60 * 1000}  // 1 week
  }));
  this.use(this.get('session'));

  this.set('views', path.join(__dirname, 'views'));
  this.set('view engine', 'jade');
};

app.middguardSocketMiddleware = function middguardSocketMiddleware() {
  var io = this.get('io');
  var session = this.get('session');

  io.use(ios(session));

  io.use((socket, next) => {
    socket.bookshelf = this.get('bookshelf');
    next();
  });
};

/**
 * Register an analytics module with the `middguard` app.
 *
 * @return `middguard.Analytics`
 * @public
 */
app.module = function module(name, requirePath) {
  var Bookshelf = this.get('bookshelf');
  var AnalyticsModule = Bookshelf.model('AnalyticsModule');
  var register = Bookshelf.collection('analytics');

  var attributes = require(requirePath);

  if (_.has(attributes, 'visualization')) {
    this.use(`/modules/${name}`, express.static(attributes.static));
```

```
  }

  register.add(new AnalyticsModule({
    name: name,
    requirePath: requirePath,
    displayName: attributes.displayName,
    inputs: attributes.inputs,
    outputs: attributes.outputs,
    visualization: attributes.visualization,
    main: attributes.visualization ? attributes.mainView
        : null
  }));
};

/**
 * Listen for connections.
 *
 * A node `http.Server` is returned, with this
 * application (which is a `Function`) as its
 * callback.
 *
 * This is the same as `express.listen`, but uses
 * the already created server, rather than creating
 * a new one in `listen`. The `http.Server` must
 * already be created to setup socket.io.
 *
 * @return {http.Server}
 * @public
 */

app.listen = function listen() {
  var server = this.get('http server');
  return server.listen.apply(server, arguments);
};
```

middguard/middguard.js

```
'use strict';

/**
 * Module dependencies.
 */

var _ = require('lodash');
var express = require('express');
var proto = require('./application');

/**
 * Expose `createApplication()`.
 */

exports = module.exports = createApplication;

/**
 * Create a middguard application.
 *
 * @return {Function}
 * @public
 */

function createApplication(settings) {
  var app = express();

  _.each(settings, (value, key) => {
    app.set(key, value);
  });

  _.extend(app, proto);

  // expressConfig(app);
  //
  // var server = http.createServer(app);
  // var io = socketio(server);
  // app.set('io', io);
  //
```

```
  // var sessionSockets = new SessionSockets(io,
  //   app.get('sessionStore'),
  //   app.get('cookieParser'));
  //
  // bookshelfConfig(app);
  //
  // // require('./middguard/loaders/models_loader')(app)
      ;
  // // require('./middguard/loaders/analytics_loader')(
      app);
  //
  // sessionSockets.on('connection', require('./middguard
      /socket'));
  //
  // require('./middguard/routes')(app);

  app.middguardInit();

  return app;
};
```

middguard/socket/index.js

```
var _ = require('lodash'),
    pluralize = require('pluralize'),
    analyst = require('./analyst'),
    message = require('./message'),
    modules = require('./modules'),
    node = require('./node'),
    io = require('socket.io')();

module.exports = function (socket) {
  var Bookshelf = socket.bookshelf;

  // Only set up sockets if we have a logged in user
  if (!socket.handshake.session.user) return;

  // Set up sockets middguard internal sockets
  socket.on('messages:create', (data, cb) => message.
      create(socket, data, cb));
  socket.on('messages:read', (data, cb) => message.
      readAll(socket, data, cb));

  socket.on('modules:read', (data, cb) => modules.readAll
      (socket, data, cb));

  socket.on('analyst:read', (data, cb) => analyst.read(
      socket, data, cb));
  socket.on('analysts:read', (data, cb) => analyst.
      readAll(socket, data, cb));

  socket.on('node:connect', (data, cb) => node.connect(
      socket, data, cb));
  socket.on('node:run', (data, cb) => node.run(socket,
      data, cb));
  socket.on('nodes:create', (data, cb) => node.create(
      socket, data, cb));
  socket.on('nodes:read', (data, cb) => node.readAll(
      socket, data, cb));
  socket.on('nodes:update', (data, cb) => node.update(
      socket, data, cb));
```

```
  var Graph = Bookshelf.model('Graph');
  patchModelToEmit(socket, 'graph', Graph);
  setupSocketEvents(socket, 'graph', Graph);

  Bookshelf.model('Node').fetchAll()
  .then(nodes => nodes.each(node => node.createReadSocket(socket)));

  // Set up sockets to call analytics from client
  // Patched models will automatically emit create, update, and delete events
  // Bookshelf.collection('analytics').each(function (analyticsAttrs) {
  //   var name = analyticsAttrs.get('name');
  //   var requirePath = analyticsAttrs.get('requirePath');
  //
  //   socket.on('analytics:' + name, function (data, callback) {
  //     require(requirePath)(Bookshelf, data);
  //   });
  // });
};

function patchModelToEmit(socket, modelName, model) {
  if (!model.prototype._emitting) {
    var _initialize = model.prototype.initialize;

    model.prototype.initialize = function () {
      var args = Array.prototype.slice.call(arguments);
      _initialize.apply(args);

      this.on('created', function (model, attrs, options) {
        // If the model was created on the client, we don't want to emit a
        // create event, since we need to assign an id on the creator via
        // a callback and do a broadcast.emit for everyone else.
        // The create listener will take care of this.
        if (!options.clientCreate) {
          io.emit(pluralize(modelName) + ':create', model.toJSON());
        } else {
          socket.broadcast.emit(pluralize(modelName) + ':create', model.toJSON());
        }
      });

      this.on('updated', function (model) {
        socket.broadcast.emit(pluralize(modelName) + ':update', model.toJSON());
      });

      this.on('destroying', function (model) {
        socket.broadcast.emit(pluralize(modelName) + ':delete', model.toJSON());
      });
    };

    model.prototype._emitting = true;
  }
}

function setupSocketEvents(socket, modelName, model) {
  // Set up create, read, update, delete sockets for each model
  socket.on(pluralize(modelName) + ':create', function (data, callback) {
    // Pass clientCreate to save so the model won't emit anything on the
    // created event and confuse the client.
    // Create is a special case since the model on the creating client doesn't
    // have an id yet.
    new model().save(data, {clientCreate: true})
```

```
    .then(function (newModel) {
      callback(null, newModel.toJSON());
    })
    .catch(function (error) {
      throw new Error(error);
    })
});

socket.on(modelName + ':update', function (data,
    callback) {
  new model({id: _.result(data, 'id')})
    .save(_.omit(data, 'id'), {patch: true});
});

socket.on(modelName + ':delete', function (data,
    callback) {
  var x = new model({id: _.result(data, 'id')});
  //console.log(String(x.destroy().
      _resolveFromSyncValue));
  new model({id: _.result(data, 'id')}).destroy();
});

socket.on(pluralize(modelName) + ':read', function (
    data, callback) {
  if (data){
    var fetchData = new model().where(data).fetchAll();
  } else {
    //if fetching all models at once
    var fetchData = new model().fetchAll();
  }
  fetchData
    .then(function (collection) {
      callback(null, collection.toJSON());
    })
    .catch(function (error) {
      callback(error);
    });
});

socket.on(modelName + ':read', function (data, callback
    ) {
  new model({id: _.result(data, 'id')}).fetch({require:
      true})
    .then(function (fetchedModel) {
      callback(null, fetchedModel.toJSON());
    })
    .catch(model.NotFoundError, function () {
      callback({'error': `Model ${modelName} not found
          .`});
    })
    .catch(function (error) {
      callback(error);
    });
});
}
```

middguard/socket/modules.js

```
/**
 * Respond to the modules:read event from a connected
    client.
 * Emits all registered analytics modules.
 *
 * @return undefined
 * @private
 */

exports.readAll = function (socket, data, callback) {
  var register = socket.bookshelf.collection('analytics')
      ;

  callback(null, register.toJSON());
};
```

middguard/socket/node.js

```
var Promise = require('bluebird');
var _ = require('lodash');

exports.create = function(socket, data, callback) {
  var Node = socket.bookshelf.model('Node');

  new Node()
  .save(data, {clientCreate: true})
  .then(node => {
    node.createReadSocket(socket);
    callback(null, node.toJSON());
    socket.broadcast.emit('nodes:create', node.toJSON());
  });
};

exports.readAll = function(socket, data, callback) {
  var Node = socket.bookshelf.model('Node');
  var nodes = new Node();

  if (data) nodes = nodes.where(data);

  nodes.fetchAll()
  .then(collection => callback(null, collection.toJSON())
      )
  .catch(callback);
};

exports.update = function(socket, data, callback) {
  var Node = socket.bookshelf.model('Node');

  new Node({id: data.id})
  .save(_.omit(data, 'id'), {patch: true})
  .then(function(node) {
    callback(null, node.toJSON());
    socket.broadcast.emit('nodes:update', node.toJSON());
  })
  .catch(callback);
};
```

```

/* Connect data.inputNode at data.inputGroup to data.
    outputNode.
 */
exports.connect = function(socket, data, callback) {
  var Node = socket.bookshelf.model('Node');
  var modules = socket.bookshelf.collection('analytics');

  var outputNode = new Node({id: data.outputNode});
  var inputNode = new Node({id: data.inputNode});

  Promise.all([outputNode.fetch(), inputNode.fetch()])
  .spread(function(outputNode, inputNode) {
    var outputModule = modules.findWhere({name:
        outputNode.get('module')});
    var inputModule = modules.findWhere({name: inputNode.
        get('module')});

    // Get outputs list from the corresponding output
        module
    var outputs = require(outputModule.get('requirePath')
        ).outputs;

    // Get inputs list from the corresponding input
        module
    var inputGroups = require(inputModule.get('
        requirePath')).inputs;
    var inputs = inputGroups.filter(function(group) {
      return group.name === data.inputGroup;
    })[0].inputs;

    // The array of connections we'll set on the input
        node
    var connections = {
      output_node: data.outputNode,
      connections: []
    };

    if (data.connections &&
        validateConnections(data.connections, inputs,
            outputs)) {
      // Use `data.connections` if the connections are
          valid
      connections.connections = data.connections;
    } else {
      // Match input and output names
      connections.connections = connectionsByName(inputs,
          outputs);
    }

    inputNode.setInputGroup(data.inputGroup, connections)
        ;
    return inputNode.save();
  })
  .then(node => {
    socket.emit('nodes:update', node.toJSON());
    socket.broadcast.emit('nodes:update', node.toJSON());
  })
  .catch(callback);
};

/**
 * Validate that all potential inputs and outputs have
    inputs and outputs on
 * with the same name on the respective nodes.
 *
 * @private
 * @param {Object[]} connections The passed in data of
    connections to set.
 * @param {Object[]} inputs Named inputs on the existing
    input node.
 * @param {Object[]} outputs Named outputs on the
    existing output node.
 */
function validateConnections(connections, inputs, outputs
    ) {
  var potentialInputs = connections.map(connection =>
      connection.input);
```

```
  var potentialOutputs = connections.map(connection => connection.output);

  return potentialInputs.length === potentialOutputs.length &&
         inputs.every(input => _.has(potentialInputs, input)) &&
         outputs.every(output => _.has(potentialOutputs, output));
}

/**
 * Generate the connections array by matching names between inputs and outputs.
 * Returns an array with size equivalent to the cardinality of inputs     outputs.
 *
 * @private
 * @param {Object[]} inputs Inputs to match
 * @param {Object[]} outputs Outputs to match
 */
function connectionsByName(inputs, outputs) {
  return outputs.filter(output => _.indexOf(inputs, output) > -1)
                .map(output => ({output: output, input: output}));
}

exports.run = function(socket, data, callback) {
  var Node = socket.bookshelf.model('Node');
  var modules = socket.bookshelf.collection('analytics');

  new Node({id: data.id})
  .fetch()
  .tap(node => node.ensureTable())
  .then(node => node.save({status: 1}))
  .then(function(node) {
    socket.emit('nodes:update', node.toJSON());
    socket.broadcast.emit('nodes:update', node.toJSON());
    return node;
  })
  .then(node => Promise.join(node, node.outputNodes()))
  .spread(function(node, outputs) {
    var module = modules.findWhere({name: node.get('module')}),
        connections = JSON.parse(node.get('connections'))
          ,
        context = {};

    context.inputs = _.reduce(_.keys(connections), function(inputs, inputGroup) {
      var groupConnections = connections[inputGroup].connections;

      // Reduce the array of input output pairs to a single associative array
      // mapping input to output.
      var columns = _.reduce(groupConnections, function(connections, pair) {
        connections[pair.input] = pair.output;
        return connections;
      }, {});

      inputs[inputGroup] = {};
      inputs[inputGroup].knex = socket.bookshelf.knex(outputs[inputGroup].get('table'));
      inputs[inputGroup].cols = columns;
      inputs[inputGroup].tableName = outputs[inputGroup].get('table');

      return inputs;
    }, {});

    context.table = {};
    context.table.knex = socket.bookshelf.knex(node.get('table'));
    context.table.name = node.get('table');

```

```
    var handle = require(module.get('requirePath')).
        handle;
    return Promise.join(node, handle(context));
  })
  .spread(function(node, result) {
    return node.save({status: 2});
  })
  .then(function(node) {
    socket.emit('nodes:update', node.toJSON());
    socket.broadcast.emit('nodes:update', node.toJSON());
  })
  .catch(callback);
};
```

middguard/models/connection.js

```
/**
 * Register the `Connection` model in the Bookshelf
    registry.
 *
 * Access this model using `Bookshelf.model('Connection')
    `.
 *
 * @return {Bookshelf.Model}
 * @private
 */

module.exports = function(app) {
  var Bookshelf = app.get('bookshelf');

  var Connection = Bookshelf.Model.extend({
    tableName: 'connection',

    from: function() {
      return this.belongsTo('Node');
    },

    to: function() {
      return this.belongsTo('Node');
    }
  });

  return Bookshelf.model('Connection', Connection);
};
```

middguard/models/graph.js

```
/**
 *
 */

module.exports = function(app) {
  var Bookshelf = app.get('bookshelf');

  var Graph = Bookshelf.Model.extend({
    tableName: 'graph',

    nodes: function() {
      return this.hasMany('Node')
    }
  });

  return Bookshelf.model('Graph', Graph);
};
```

middguard/models/node.js

```
'use strict';

var _ = require('lodash');
var Promise = require('bluebird');

/**
 * Register the `Node` model in the Bookshelf registry.
 *
 * @return {Bookshelf.Model}
 * @private
 */

module.exports = function(app) {
  var Bookshelf = app.get('bookshelf');

  var Node = Bookshelf.Model.extend({
    tableName: 'node',

    initialize: function() {
      this.on('creating', this.createTableName);
    },

    graph: function() {
      return this.belongsTo('Graph');
    },

    status: function() {
      var statuses = {
        0: 'Not run',
        1: 'In progress',
        2: 'Done'
      };

      return statuses[this.get('status')];
    },

    createTableName: function() {
      return Node
```

```
    .where('module', this.get('module'))
    .count()
    .then(count => {
      return this.set('table', `${this.get('module')}_$
          {count + 1}`);
    });
  },

  /**
   * Get a mapping from input group names to output
       nodes.
   *
   * @return a promise for an object mapping input
       group name
   *         to a fetched output node
   */
  outputNodes: function() {
    var connections = JSON.parse(this.get('connections'
        ));

    return Promise.reduce(_.keys(connections), function
        (outputs, inputGroup) {
      var outputId = connections[inputGroup].
          output_node;

      return new Node({id: outputId}).fetch()
      .then(node => {
        outputs[inputGroup] = node;
        return outputs;
      });
    }, {});
  },

  /**
   * Create this node's table if it doesn't already.
   */
  ensureTable: function() {
    return Bookshelf.knex.schema.hasTable(this.get('
        table'))
    .then(exists => {
      if (!exists) {
        return this.module().createTable(this.get('
            table'), Bookshelf.knex);
      }
    });
  },

  module: function() {
    var modules = Bookshelf.collection('analytics'),
        moduleName = this.get('module'),
        module = modules.findWhere({name: moduleName});

    return require(module.get('requirePath'));
  },

  /**
   * Set an input group on the node's connections.
   * The text column "connections" remains in its
       stringified JSON state.
   *
   * @param {String} inputGroup Input group to set.
   * @param {Object} connections Connections to set for
         `inputGroup`.
   * @return this
   */
  setInputGroup: function(inputGroup, connections) {
    let groups = JSON.parse(this.get('connections')) ||
         {};
    groups[inputGroup] = connections;

    return this.set('connections', JSON.stringify(
        groups));
  },

  createReadSocket: function(socket) {
    let table = Bookshelf.knex(this.get('table'));

    socket.on(`${this.get('table')}:read`, (data,
```

```
            callback) => {
        if (!_.isEmpty(data)) {
          var query = Bookshelf.knex(this.get('table')).
              where(data).select('*');
        } else {
          var query = Bookshelf.knex(this.get('table')).
              select('*');
        }

        query.then(results => callback(null, results));
      });
    }
  });

  return Bookshelf.model('Node', Node);
};
```

middguard/migrations/20140728124252˙initial.js

```
'use strict';

exports.up = function(knex, Promise) {
  return knex.schema.createTable('analyst', function(
      table) {
    table.increments('id').primary();
    table.text('username').unique();
    table.text('password');
  })
  .createTable('message', function(table) {
    table.increments('id').primary();
    table.integer('analyst_id').references('analyst.id');
    table.text('state');
    table.text('content');
    table.dateTime('timestamp');
  })
  .createTable('graph', function(table) {
    table.increments('id').primary();
    table.string('name');
  })
  .createTable('node', function(table) {
    table.increments('id').primary();
    table.integer('graph_id').references('graph.id');
    table.string('module');
    table.string('table');
    table.integer('status').defaultTo(0);
    table.string('connections').defaultTo('{}');
  });
};

exports.down = function(knex, Promise) {
  return knex.schema.dropTable('analyst')
                    .dropTable('message')
                    .dropTable('graph')
                    .dropTable('node')
                    .dropTable('connection');
};
```

middguard/migrations/20160405022013˙node˙coordinates.js

```
'use strict';

exports.up = function(knex, Promise) {
  return knex.schema.table('node', function(table) {
    table.integer('radius').defaultTo(75);

    // These are the top left coordinates of the node,
    // not the center coordinates.
    table.integer('position_x').defaultTo(0);
    table.integer('position_y').defaultTo(0);
  });
};

exports.down = function(knex, Promise) {
  return knex.schema.table('node', function(table) {
    table.dropColumns('radius', 'position_x', 'position_y
        ');
  });
};
```

static/js/entities.js

```
var middguard = middguard || {};

(function() {
  middguard.entities = {};

  middguard.EntityCollection = Backbone.Collection.extend
      ({
    initialize: function (models, options) {
      this.url = _.result(options, 'url');

      _.bindAll(this, 'serverCreate', 'serverUpdate', '
          serverDelete');
      this.ioBind('create', this.serverCreate, this);
      this.ioBind('update', this.serverUpdate, this);
      this.ioBind('delete', this.serverDelete, this);

      this.listenTo(this, 'sync', this.addViewReferences)
          ;
    },
    serverCreate: function (data) {
      var exists = this.get(data.id);
      if (!exists) {
        this.add(data);
      } else {
        exists.set(data);
      }
    },
    serverUpdate: function (data) {
      var exists = this.get(data.id);
      if (exists) exists.set(data);
    },
    serverDelete: function (data) {
      // Already deleted from database, so don't need to
          model.destroy
      var exists = this.get(data.id);
      if (exists) this.remove(exists);
    },
    addViewReferences: function (collection, response,
```

```
      options) {
    var middguard_view_name = options.
        middguard_view_name;

    // if a view name wasn't passed in we can't do
        anything about it
    if (!middguard_view_name)
      return;

    console.log('Adding view references for view "' +
        middguard_view_name +
                '" to ' + response.length + ' fetched
                    models.');

    // get models added to the collection that match
        the criteria
    // we fetched for
    (options.data
      ? collection.where(options.data)
      : collection).forEach(function (model) {
      var currentViews = model.get('middguard_views');

      // if 'middguard_views' doesn't exist on the
          model, set it to an empty
      // array
      if (!currentViews)
        model.set('middguard_views', []);

      currentViews = model.get('middguard_views');

      // if the view has already been added
      if (currentViews.indexOf(options.
          middguard_view_name) > -1)
        return;

      // add the view to the model's 'middguard_views'
      currentViews.push(middguard_view_name);
      model.set('middguard_views', currentViews);
    });
      }
    });
})();
```

static/js/visualization-manager.js

```
var middguard = middguard || {};

(function() {

  middguard.View = Backbone.View.extend({
    className: 'middguard-module',
    fetch: function (collection, options) {
      // set the view name to add to the middguard_views when we create/update
      // the models
      options.middguard_view_name = this.cid;

      // add the entity to this view
      // so we can check the entities and remove the view from middguard_views
      // when the view is destroyed
      if (this.middguard_entities.indexOf(collection) < 0) {
        this.middguard_entities.push(collection);
      }

      middguard.entities[collection].fetch(options);
    },

    /* middguard.View.prototype.remove
     * Extend the view remove function to remove referenced models
     *
     * Important: If you need to extend remove functionality, you must call
     * `middguard.View.prototype.remove.call(this)` as the super call instead
     * of the usual `Backbone.View.prototype.remove.call(this)`.
     */
    remove: function () {
      var viewName = this.cid;

      console.log('About to remove view "' + viewName + '".');

      // For each model this view references
      this.middguard_entities.forEach(function (entityName) {
        var collection = middguard.entities[entityName];

        // First iteration to remove reference to this model
        collection.each(function (model, i) {
          if (model.get('middguard_views').indexOf(viewName) > -1) {
            removeFromArray(model.get('middguard_views'), viewName);
          }
        });

        // Get an array of models from this entity collection to remove
        var toRemove = collection.filter(function (model) {
          if (model.get('middguard_views').length === 0) {
            delete model.attributes.middguard_views;
            return true;
          }
        });

        console.log('Removing ' + toRemove.length +
                    ' models that are no longer in use from collection "' +
                    entityName + '".');
        // remove them without sending anything to the server
        collection.remove(toRemove);
      });

      console.log('Done removing view "' + viewName + '".
```

```
          ');

    // call super
    Backbone.View.prototype.remove.call(this);
  },

  createContext: function() {
    var moduleName = this.model.get('module')
        module = middguard.PackagedModules.findWhere({
            name: moduleName}),
        connections = JSON.parse(this.model.get('
            connections')),
        context = {};

    context.inputs = _.reduce(_.keys(connections),
        function(inputs, inputGroup) {
      var groupConnections = connections[inputGroup].
          connections,
          outputNode = middguard.Nodes.get(connections[
              inputGroup].output_node);

      var columns = _.reduce(groupConnections, function
          (connections, pair) {
        connections[pair.input] = pair.output;
        return connections;
      }, {});

      inputs[inputGroup] = {};
      inputs[inputGroup].collection = middguard.
          entities[outputNode.get('table')];
      inputs[inputGroup].cols = columns;
      inputs[inputGroup].tableName = outputNode.get('
          table');

      return inputs;
    }, {});

    return context;
  }
});

middguard.activateView = function(node) {
  var main = middguard.Nodes.get(node).module().get('
      main');
  var ctor = middguard.__modules[main].ctor;
  var live = new ctor({model: middguard.Nodes.get(node)
      });

  middguard.__modules[node] = {};
  middguard.__modules[node].live = live;

  $('.middguard-views').append(live.render().el);
};

middguard.deactivateView = function(node) {
  middguard.__modules[node].live.remove();
  middguard.__modules[node].live = null;
};

middguard.toggleView = function(node) {
  if (middguard.__modules[node] && middguard.__modules[
      node].live) {
    middguard.deactivateView(node);
  } else {
    middguard.activateView(node);
  }
};

// Internal hash of module views
middguard.__modules = {};

// Internal hash of submodule views
middguard.__submodules = {};

/* middguard.addModule
 * Makes MiddGuard aware of a top level view.
 * Top level views are listed under "Modules" in the
     sidebar.
```

```
 */
middguard.addModule = function (name, view) {
  _addView(name, view, true /* top level */);
};

/* middguard.addSubview
 * Makes MiddGuard aware of a subview (a view
     instantiated from another view)
 * Subviews are not listed in the sidebar, but have
     models they fetch tracked
 * and removed when the view is removed.
 */
middguard.addSubview = function (name, view) {
  _addView(name, view, false /* not top level */);
};

var _addView = function (name, view, topLevel) {
  if (!Object.prototype.hasOwnProperty.call(middguard.
      __modules, name)) {
    view.prototype.middguard_view_name = name;
    view.prototype.middguard_entities = [];

    if (topLevel) {
      middguard.__modules[name] = {ctor: view, live:
          null};
    } else {
      middguard.__submodules[name] = {ctor: view, live:
           null};
    }
  } else {
    throw new Error('Module ' + name + ' already loaded
        ');
  }
};

/* Remove elements from an array.
 * arr is the array to remove from (param 0).
 * Elements to remove are arguments 1 .. n.
 * Source: http://stackoverflow.com/questions/3954438
 */
function removeFromArray(arr) {
  var what, a = arguments, L = a.length, ax;
  while (L > 1 && arr.length) {
    what = a[--L];
    while ((ax= arr.indexOf(what)) !== -1) {
      arr.splice(ax, 1);
    }
  }
  return arr;
}
})();
```

static/js/collections/graphs.js

```
var middguard = middguard || {};

(function() {
  'use strict';

  var Graphs = middguard.BaseCollection.extend({
    model: middguard.Graph,
    url: 'graphs'
  });

  middguard.Graphs = new Graphs();
})();
```

static/js/collections/nodes.js

```
var middguard = middguard || {};

(function() {
  'use strict';

  var Nodes = middguard.BaseCollection.extend({
    model: middguard.Node,

    url: 'nodes',

    initialize: function() {
      _.bindAll(this, 'serverCreate', 'serverUpdate');

      this.ioBind('create', this.serverCreate, this);
      this.ioBind('update', this.serverUpdate, this);
    },

    serverCreate: function(data) {
      var exists = this.get(data.id);
      if (!exists) {
        this.add(data);
      } else {
        exists.set(data);
      }
    },

    serverUpdate: function(data) {
      var exists = this.get(data.id);
      if (exists) {
        exists.set(data);
      }
    },
  });

  middguard.Nodes = new Nodes();
})();
```

static/js/collections/packaged-modules.js

```
var middguard = middguard || {};

(function () {
  var PackagedModules = Backbone.Collection.extend({
    url: 'modules',
    model: middguard.PackagedModule
  });

  middguard.PackagedModules = new PackagedModules();
})();
```

static/js/models/graph.js

```
var middguard = middguard || {};

(function() {
  middguard.Graph = Backbone.Model.extend();
})();
```

static/js/models/node.js

```javascript
var middguard = middguard || {};

(function() {
  middguard.Node = Backbone.Model.extend({
    blacklistAttributes: [
      'selectedInput',
      'selectedOutput'
    ],

    defaults: {
      status: 0,
      radius: 75,
      position_x: 0,
      position_y: 0,
      selectedInput: null,
      selectedOutput: null,
      connections: '{}'
    },

    statusMap: {
      0: 'Not run',
      1: 'In progress',
      2: 'Completed'
    },

    connectToOutput: function(other, inputGroup) {
      middguard.socket.emit('node:connect', {
        outputNode: other.get('id'),
        inputNode: this.get('id'),
        inputGroup: inputGroup
      });
    },

    run: function() {
      middguard.socket.emit('node:run', {
        id: this.get('id')
      });
    },

    position: function(x, y) {
      if (!arguments.length) {
        return {x: this.get('position_x'), y: this.get('
            position_y')};
      } else {
        this.set('position_x', x);
        this.set('position_y', y);
      }
    },

    toJSON: function(options) {
      return _.omit(this.attributes, this.
          blacklistAttributes);
    },

    statusText: function() {
      return this.statusMap[this.get('status')];
    },

    module: function() {
      return middguard.PackagedModules.findWhere({
        name: this.get('module')
      });
    },

    unconnectedInputs: function(inputGroup) {
      var connections = JSON.parse(this.get('connections'
          ))[inputGroup],
          allInputs = _.find(this.module().get('inputs'),
                            {name: inputGroup}).inputs;

      if (!connections) {
        return allInputs;
      }

      var connectedInputs = connections.connections.map(c
          => c.input);
      return _.difference(allInputs, connectedInputs);
```

```
  },

  unconnectedOutputs: function(inputGroup) {
    var connections = JSON.parse(this.get('connections'
        ))[inputGroup];

    if (!connections.output_node) {
      return [];
    }

    var connectedOutputs = connections.connections.map(
        c => c.output);
    var outputNode = middguard.Nodes.get(connections.
        output_node);
    var allOutputs = middguard.PackagedModules
        .find({name: outputNode.get('module')})
        .get('outputs');

    return _.difference(allOutputs, connectedOutputs);
  },

  isVisualization: function() {
    return this.module().get('visualization');
  }
});
})();
```

static/js/models/packaged-module.js

```
var middguard = middguard || {};

(function () {
  middguard.PackagedModule = Backbone.Model.extend({
    defaults: {
      'name': '',
      'main': '',
      visualization: false
    }
  });
})();
```

static/js/views/graphs-view.js

```
var middguard = middguard || {};

(function() {
  'use strict';

  middguard.GraphsView = Backbone.View.extend({
    className: 'middguard-graphs',

    template: _.template($('#graphs-panel-template').html()),

    events: {
      'click #create-new-graph': 'createGraph'
    },

    initialize: function() {
      this.listenTo(middguard.Graphs, 'add', this.addOneGraph);
      this.listenTo(middguard.Graphs, 'reset', this.addAllGraphs);

      middguard.Graphs.fetch({reset: true, data: {}});
    },

    render: function() {
      this.$el.html(this.template());

      this.$graphs = this.$('.graphs-list');

      return this;
    },

    addOneGraph: function(graph) {
      var graphView = new GraphView({model: graph});

      this.$graphs.append(graphView.render().el);
    },

    addAllGraphs: function() {
      middguard.Graphs.each(this.addOneGraph, this);
    },

    createGraph: function(e) {
      e.preventDefault();
      var name = this.$('#new-graph-name').val().trim();

      middguard.Graphs.create({name: name}, {wait: true});
      this.$('#new-graph-name').val('');
    }
  });

  var GraphView = Backbone.View.extend({
    className: 'middguard-graph list-group-item',

    tagName: 'a',

    template: _.template('<%= name %>'),

    events: {
      'click': 'toggleEditor'
    },

    initialize: function() {
      this.editing = false;

      this.listenTo(this.model, 'update', this.render);
    },

    render: function() {
      this.$el.html(this.template(this.model.toJSON()));

      this.$el.attr('href', '#');
      return this;
    },

    toggleEditor: function() {
```

```
      if (this.editor) {
        this.editor.remove();
        this.editor = null;
      } else {
        this.editor = new middguard.GraphEditorView({graph: this.model});
        $('.middguard-views').append(this.editor.render().el);
      }

      this.$el.toggleClass('active', Boolean(this.editor));
    }
  });


})();
```

static/js/views/graph-editor-view.js

```
var middguard = middguard || {};

(function() {
  'use strict';

  middguard.GraphEditorView = Backbone.View.extend({
    className: 'middguard-graph-editor middguard-module',

    tagName: 'div',

    template: _.template($('#graph-editor-template').html()),

    initialize: function(options) {
      this.graph = options.graph;
      this.detailView = null;

      this.listenTo(middguard.PackagedModules, 'reset', this.addModules);
      this.listenTo(middguard.Nodes, 'reset', this.addAllNodes);
      this.listenTo(middguard.Nodes, 'reset', this.addAllConnectorGroups);
      this.listenTo(middguard.Nodes, 'reset', this.ensureEntityCollections);
      this.listenTo(middguard.Nodes, 'add', this.addNode);
      this.listenTo(middguard.Nodes, 'add', this.addConnectorGroup);

      middguard.PackagedModules.fetch({reset: true, data: {}});
      middguard.Nodes.fetch({reset: true, data: {}});
    },

    render: function() {
      this.$el.html(this.template(this.graph.toJSON()));
      d3.select(this.el).select('.editor').append('svg')
```

```
        .attr('class', 'graph')
        .attr('width', 500);

    this.resizeEditor();

    return this;
  },

  ensureEntityCollections: function() {
    middguard.Nodes.each(this.ensureEntityCollection,
        this);
  },

  ensureEntityCollection: function(node) {
    var tableName = node.get('table');

    if (!tableName || middguard.entities[tableName])
        return;

    var collection = new middguard.EntityCollection([],
         {
      url: tableName
    });

    middguard.entities[tableName] = collection;
  },

  resizeEditor: function() {
    d3.select(this.el).select('.editor svg')
        .attr('height', $(window).height() - this.$('.
            header').outerHeight());
  },

  addModules: function() {
    this.$('.modules-list').html('');

    middguard.PackagedModules.each(function(model) {
      var view = new ModuleListItemView({model: model,
          graph: this.graph});
      this.$('.modules-list').append(view.render().el);
    }.bind(this));

    this.resizeEditor();
  },

  addNode: function(node) {
    if (node.get('graph_id') !== this.graph.get('id'))
        {
      return;
    }

    var view = new NodeView({model: node, editor: this
        });
    this.$('.graph').append(view.render().el);
  },

  addAllNodes: function(node) {
    middguard.Nodes.each(this.addNode, this);
  },

  addConnectorGroup: function(node) {
    if (node.get('graph_id') !== this.graph.get('id'))
        {
      return;
    }

    var view = new ConnectorGroupView({model: node});
    this.$('.graph').append(view.render().el);
  },

  addAllConnectorGroups: function() {
    middguard.Nodes.each(this.addConnectorGroup, this);
  },

  setDetailView: function(view) {
    if (this.detailView) {
      this.detailView.remove();
    }
```

```
      this.$('.detail').html(view.render().el);
      this.detailView = view;
    }
  });

  var ModuleListItemView = Backbone.View.extend({
    tagName: 'li',

    className: 'btn btn-default module',

    template: _.template('<%= displayName %>'),

    events: {
      'click': 'createNode',
    },

    initialize: function(options) {
      this.model = options.model;
      this.graph = options.graph;
    },

    render: function() {
      this.$el.html(this.template(this.model.toJSON()));
      return this;
    },

    createNode: function() {
      middguard.Nodes.create({
        module: this.model.get('name'),
        graph_id: this.graph.get('id')
      });
    }
  });

  /* Nodes' connections are stored on the input node.
   * All the connecting lines from an a node's
       connections
   * to the corresponding output node.
   */
  var ConnectorGroupView = Backbone.NSView.extend({
    tagName: 'svg:g',

    initialize: function() {
      this.connections = [];

      if (this.model.get('connections'))
        this.addAllConnectingLines();

      // `this.model` is the "input" node
      this.listenTo(this.model, 'change', this.render);
    },

    render: function() {
      this.connections.forEach(connection => connection.
          render());
      this.unrenderedConnections().forEach(this.
          addConnectingLine, this);

      return this;
    },

    addAllConnectingLines: function() {
      _.chain(JSON.parse(this.model.get('connections')))
          .keys()
          .each(this.addConnectingLine, this);
    },

    addConnectingLine: function(inputGroup) {
      var view = new ConnectorView({
        model: this.model,
        inputGroup: inputGroup
      });
      this.$el.append(view.render().el);
      this.connections.push(view);
    },

    renderedConnections: function() {
```

```javascript
    return this.connections.map(connection => connection.inputGroup);
  },

  unrenderedConnections: function() {
    return _.chain(JSON.parse(this.model.get('connections')))
        .keys()
        .difference(this.renderedConnections());
  }
});

var ConnectorView = Backbone.NSView.extend({
  tagName: 'svg:path',

  className: 'connecting-line',

  initialize: function(options) {
    this.model = options.model;
    this.inputGroup = options.inputGroup;
    this.outputNode = middguard.Nodes.findWhere({
      id: JSON.parse(this.model.get('connections'))[this.inputGroup].output_node
    });
    this.module = middguard.PackagedModules.findWhere({
      name: this.model.get('module')
    });

    this.diagonal = d3.svg.diagonal();

    // Only rerender this particular line when the output node moves.
    // We rerender all the lines (parent view) when the input node moves.
    this.listenTo(this.outputNode, 'change', this.render);

    // Check if the connection has changed. In this context, "changed"
    // means that the connection's input group either no longer has a
    // connection, or the input group is connected to a different output.
    this.listenTo(this.model, 'change', this.connectionChanged);
  },

  render: function() {
    this.diagonal
        .source(this.outputPosition())
        .target(this.inputPosition());

    this.$el.attr('d', this.diagonal());

    return this;
  },

  inputPosition: function() {
    var i = _.findIndex(this.module.get('inputs'), input => {
          return input.name === this.inputGroup;
        }),
        r = this.model.get('radius'),
        n = this.module.get('inputs').length,
        offset = NodeView.prototype.inputPosition(i, r, n);

    return {
      x: this.model.position().x + offset.x,
      y: this.model.position().y + offset.y
    };
  },

  outputPosition: function() {
    var r = this.outputNode.get('radius');

    return {
      x: this.outputNode.position().x + r,
```

```
        y: this.outputNode.position().y + 2 * r - 10
      };
    },

    connectionChanged: function() {
      var connections = this.model.get('connections'),
          connection = JSON.parse(connections)[this.inputGroup];

      // No longer a connection for this input group
      if (!connection) {
        this.remove();
      }

      // A connection exists for this input group, but connected to a
      // different output node
      if (connection.output_node !== this.outputNode.get('id')) {
        // Stop listening to changes in the old output node
        this.stopListening(this.outputNode);

        // Find and bind to the new output node
        this.outputNode = middguard.Nodes.get(connection.output_node);
        this.listenTo(this.outputNode, 'change', this.render);
        this.render();
      }
    }
  });

  var NodeView = Backbone.NSView.extend({
    tagName: 'svg:g',

    className: 'node',

    events: {
      'mouseover .input': 'showInputTooltip',
      'mouseout .input': 'hideInputTooltip',
      'click .input': 'toggleInputSelected',
      'click .output': 'toggleOutputSelected',
      'click .run': 'runNode',
      'click': 'toggleDetail'
    },

    initialize: function(options) {
      this.editor = options.editor;
      this.model = options.model;
      this.module = middguard.PackagedModules.findWhere({
        name: this.model.get('module')
      });

      this.d3el = d3.select(this.el)
          .datum(this.model.position());

      this.drag = d3.behavior.drag()
          .origin(function(d) { return d; })
          .on('dragstart', this.dragstarted.bind(this))
          .on('drag', this.dragged.bind(this))
          .on('dragend', this.dragended.bind(this));

      this.listenTo(this.model, 'change', this.render);
    },

    template: _.template($('#graph-node-template').html()),

    render: function() {
      var x = this.model.position().x;
      var y = this.model.position().y;

      this.d3el
          .datum(this.model.position())
          .attr('transform', 'translate(' + x + ',' + y + ')')
          .call(this.drag);
```

```

  this.$el.html(this.template({
    r: this.model.get('radius'),
    handlePosition: this.dragHandlePosition(),
    dragHandlePath: d3.svg.symbol().type('cross').
        size(150)(),
    runPosition: this.runPosition(),
    runPath: d3.svg.symbol().type('triangle-up').size
        (150)(),
    status: this.model.get('status'),
    statusText: this.model.statusText(),
    displayName: this.module.get('displayName'),
    inputs: this.module.get('inputs'),
    output: this.module.get('outputs').length,
    inputPosition: this.inputPosition
  }));

  var selectedInput = this.model.get('selectedInput')
      ,
      selectedOutput = this.model.get('selectedOutput
          ');
  if (selectedInput)
    this.d3el.select('[data-name="' + selectedInput.
        name + '"]')
        .classed('selected', true);

  if (selectedOutput)
    this.d3el.select('.output')
        .classed('selected', true);

  if (this.model.isVisualization())
    this.d3el.classed('visualization', true);


  return this;
},

dragstarted: function(d) {
  this.dragStartPosition = _.clone(d);
},

dragged: function(d) {
  if (!d3.select(d3.event.sourceEvent.target).classed
      ('drag-handle'))
    return;

  var x = d3.event.x;
  var y = d3.event.y;
  var r = this.model.get('radius');

  var svg = d3.select(this.editor.el).select('svg');
  var bounds = {x: svg.attr('width'), y: svg.attr('
      height')};

  // Prevent element from being dragged out bounds
  if (x < 0) x = 0;
  if (y < 0) y = 0;
  if (y + r * 2 > bounds.y) y = bounds.y - r * 2;
  if (x + r * 2 > bounds.x) x = bounds.x - r * 2;

  this.model.position(x, y);
  d3.select(this.el)
      .attr('transform', 'translate(' + (d.x = x) + '
          ,' + (d.y = y) + ')');
},

dragended: function() {
  if (this.dragMoved())
    this.model.save();
},

dragMoved: function() {
  var origin = this.dragStartPosition,
      current = this.model.position();

  return origin.x !== current.x ||
         origin.y !== current.y;
},
```

```
  showInputTooltip: function(event) {
    var tooltip = d3.select('.input-tooltip');

    if (!tooltip[0][0])
      tooltip = d3.select('body').append('div')
          .attr('class', 'input-tooltip');

    var input = _.find(this.module.get('inputs'),
        function(input) {
      return input.name === $(event.currentTarget).data
          ('name');
    });
    tooltip.html(input.name);

    var bounds = event.currentTarget.
        getBoundingClientRect(),
        inputRadius = 5,
        tooltipWidth = parseFloat(tooltip.style('width'
            )) / 2,
        tooltipHeight = parseFloat(tooltip.style('
            height')) + 5;

    tooltip
      .style('left', bounds.left - tooltipWidth +
          inputRadius + 'px')
      .style('top', bounds.top - tooltipHeight + 'px')
      .style('visibility', 'visible');
  },

  hideInputTooltip: function() {
    d3.select('.input-tooltip')
        .style('visibility', 'hidden');
  },

  toggleInputSelected: function(event) {
    var previouslySelected = middguard.Nodes.find(
        function(node) {
      return node.get('selectedInput');
    });

    // Deselect the previously selected input.
    previouslySelected && previouslySelected.set('
        selectedInput', null);

    var selectedGroup = _.find(this.module.get('inputs'
        ), function(input) {
      return input.name === $(event.target).data('name'
          );
    });

    // If the clicked node was already selected, return
         after toggling it off.
    if (previouslySelected &&
        this.model.get('id') === previouslySelected.get
            ('id') &&
        selectedGroup.name === previouslySelected.get('
            name')) {
      return;
    }

    this.model.set('selectedInput', selectedGroup);
    this.connectNodes();
  },

  toggleOutputSelected: function(event) {
    var previouslySelected = middguard.Nodes.find(
        function(node) {
      return node.get('selectedOutput');
    })

    previouslySelected && previouslySelected.set('
        selectedOutput', null);

    this.model.set('selectedOutput', true);
    this.connectNodes();
  },
```

```
  connectNodes: function() {
    var input = middguard.Nodes.find(function(node) {
      return node.get('selectedInput');
    });

    var output = middguard.Nodes.find(function(node) {
      return node.get('selectedOutput');
    });

    if (!input || !output)
      return;

    var group = input.get('selectedInput').name;

    input.connectToOutput(output, group);
    input.set('selectedInput', null);
    output.set('selectedOutput', null);
  },

  runNode: function() {
    if (this.model.isVisualization()) {
      middguard.toggleView(this.model.get('id'));
    } else {
      this.model.run();
    }
  },

  toggleDetail: function() {
    var view = new NodeDetailView({model: this.model});

    this.editor.$('.node').removeClass('selected');
    this.$el.addClass('selected');

    this.editor.setDetailView(view);
  },

  dragHandlePosition: function() {
    var r = this.model.get('radius');
    return {
      x: r + -r * Math.sqrt(2) / 2 + 15,
      y: r - r * Math.sqrt(2) / 2 + 15
    };
  },

  runPosition: function() {
    var r = this.model.get('radius');
    return {
      x: r + r * Math.sqrt(2) / 2 - 15,
      y: r - r * Math.sqrt(2) / 2 + 15
    };
  },

  /* Calculate each input circle's position.
   * Circles are arranged in rows of three from the top
       down.
   * Assume 5 pixel circle radius and 15 pixels spacing
       between
   * circle centerpoints. Circles are centered around
      the node's centerline.
   *
   * Example: 5 inputs (x is an input circle)
   *     x <--15px-->  x  <--15px--> x
   *          (15px between rows)
   *           x <-- 15px --> x
   *
   * @param i: input index
   * @param r: the input parent node's radius
   * @param n: total number of inputs for the node
   *
   * @return the center position for the input circle
   */
  inputPosition: function(i, r, n) {
    var rowIndexX = i % 3,
        rowIndexY = Math.floor(i / 3),
        rowLength = i >= n - n % 3 ? n % 3 : 3,
        baseX = r - (rowLength - 1) * 7.5,
        baseY = 10;

```

```
      return {
        x: baseX + 15 * rowIndexX,
        y: baseY + 15 * rowIndexY
      };
    }
  });

  var NodeDetailView = Backbone.View.extend({
    initialize: function() {
      this.connections = JSON.parse(this.model.get('
          connections'));
      this.module = this.model.module();

      this.selectedInputGroup = null;
      this.selectedOutput = null;
      this.selectedInput = null;

      this.listenTo(this.model, 'change', this.render);
    },

    template: _.template(
      `<h4><%- name %></h4>
      <div class="connection-groups"><div>`),

    connectionGroupTemplate: _.template($('#connection-
        group-template').html()),

    events: {
      'click .connection': 'selectConnector',
    },

    render: function() {
      this.$el.html(this.template({
        name: this.module.get('displayName')
      }));

      this.addAllConnectionGroups();

      return this;
    },

    addAllConnectionGroups: function() {
      _.each(this.connections, (value, key) => {
        var inputs = value.connections.map(connection =>
            connection.input),
            outputs = value.connections.map(connection =>
                 connection.output),
            outputNode = middguard.Nodes.get(value.
                output_node),
            outputModule = middguard.PackagedModules.
                findWhere({
              name: outputNode.get('module')
            });

        this.$('.connection-groups').prepend(this.
            connectionGroupTemplate({
          inputGroupName: key,
          inputs: inputs,
          unconnectedInputs: this.model.unconnectedInputs
              (key),
          outputModuleName: outputModule.get('displayName
              '),
          outputs: outputs,
          unconnectedOutputs: this.model.
              unconnectedOutputs(key)
        }));
      });
    },

    deselectOutput: function() {
      this.selectedOutput = null;
      this.$('.connection.output').removeClass('selected'
          );
    },

    deselectInput: function() {
      this.selectedInput = null;
      this.$('.connection.input').removeClass('selected')
```

```
      ;
},

selectConnector: function(event) {
  var $clicked = $(event.target),
      group = $clicked.closest('.connection-list-
          group').data('inputgroup'),
      name = $clicked.text(),
      isInput = $clicked.hasClass('input'),
      isOutput = $clicked.hasClass('output'),
```

```
        sameGroup = this.selectedInputGroup === group;

    if (isInput) {
      if (sameGroup) this.deselectInput();
      else this.deselectOutput();

      this.selectedInput = name;
    }

    if (isOutput) {
      if (sameGroup) this.deselectOutput();
      else this.deselectInput();

      this.selectedOutput = name;
    }

    this.selectedInputGroup = group;
    $clicked.addClass('selected');
    this.connectSelection();
  },

  connectSelection: function() {
    if (!this.selectedInputGroup ||
        !this.selectedInput ||
        !this.selectedOutput) {
      return;
    }

    var connections = this.connections[this.
        selectedInputGroup].connections;

    var exists = _.find(connections, {input: this.
        selectedInput}) ||
                 _.find(connections, {output: this.
                     selectedOutput});

    if (exists) {
      exists.input = this.selectedInput;
      exists.output = this.selectedOutput;
    } else {
      connections.push({
        input: this.selectedInput,
        output: this.selectedOutput
      });
    }

    this.connections[this.selectedInputGroup].
        connections = connections;

    this.deselectInput();
    this.deselectOutput();
    this.selectedInputGroup = null;

    this.model.set('connections', JSON.stringify(this.
        connections));
    this.model.save();
  }
});
})();
```

BIBLIOGRAPHY

[1] C. Andrews and J. Billings. Middguard at dinofun world. In *Visual Analytics Science and Technology (VAST), 2015 IEEE Conference on*, pages 129–130, Oct 2015.

[2] Jeremy Ashkenas and DocumentCloud. Backbone.js. `http://backbonejs.org/`, 2016.

[3] Liu Bin and Chen Gang. Eagleyes: Performing data analysis using an interactive dataflow. In *Visual Analytics Science and Technology (VAST), 2015 IEEE Conference on*, pages 165–166, Oct 2015.

[4] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. D3: Data-driven documents. *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)*, 2011.

[5] Kevin Burke, Kyle Conroy, Ryan Horn, Frank Stratton, and Guillaume Binet. Flask restful. `http://flask-restful-cn.readthedocs.io/en/0.3.5/`, 2015.

[6] VA Community. Vast chellenge 2014. `http://www.vacommunity.org/VAST+Challenge+2014`, 2014.

[7] I. Fette and A. Melnikov. The websocket protocol. RFC 6455, RFC Editor, December 2011. `http://www.rfc-editor.org/rfc/rfc6455.txt`.

[8] Node.js Foundation. About — node.js. `https://nodejs.org/en/about/`, 2016.

[9] Tim Griesser. Knex.js - a sql query builder for javascript. `http://knexjs.org/`, 2016.

[10] Mozilla Developer Network and individual contributors. Websockets - web apis — mdn. `https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API`, 2016.

[11] Armin Ronacher. Flask. `http://flask.pocoo.org/`, 2016.

[12] John Stasko, Carsten Görg, and Zhicheng Liu. Jigsaw: Supporting investigative analysis through interactive visualization. *Information Visualization*, 7(2):118–132, April 2008.

[13] James J. Thomas and Kristin A. Cook. *Illuminating the Path: The Research and Development Agenda for Visual Analytics*. National Visualization and Analytics Ctr, 2005.

[14] Chris Weaver. Building highly-coordinated visualizations in Improvise. In *Proceedings of the IEEE Symposium on Information Visualization (InfoVis)*, pages 159–166, Austin, TX, October 2004. IEEE Computer Society.