MIDDGUARD

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

Your abstract goes here.

ACKNOWLEDGEMENTS

Your acknowledgements go here.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Be able to create bespoke visualizations quickly.

1.1 Visual Analytics

1. Why this is a useful tool in the context of visual analytics.

CHAPTER 2

BACKGROUND

2.1 State of the Art

1. Tools that currently exist and inspired MiddGuard.

   (a) Improvise

   (b) Eagle Eyes

2.2 Previous Work on MiddGuard

2.2.1 VAST 2014

Initial work on the MiddGuard framework began during summer 2014 as a research project with Christopher Andrews and Dana Silver. For a VAST 2014 Mini-Challenge 2 [5] submission, researchers 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 [9] and Flask RESTful [4] queried the database and transformed data for various front-end visualizations. The server also performed manipulations in addition those in the pre-processing stage on a request-by-request basis based on analyst input in the interactive visualizations.

An example of the flow between preprocessing scripts, back-end server, and front-end visualization is how analysts identified points of interest in the Mini-Challenge 2 geographical data. 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.

Available data for Mini-Challenge 2 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. A preprocessing script iterated through each trace's points, identified periods where a car was stopped for greater than 120 seconds, and saved that coordinate as a destination for the associated car. A visualization used TopoJSON [3] generated from the shapefile and the tracking data to draw a map of the city overlayed with cars' movements and destinations. By selecting a destination in the visualization, an investigator could create a point of interest based on the names of locations in the tourist map and persist the association of point of interest and destination to the database. During the persistence step other destinations within a certain radius would also be associate with that point of interest.

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

The first version of MiddGuard, which was developed during the same period of summer research at Middlebury, attempted to generalize parts of the web server and front-end that could be reused throughout multiple investigations, while keeping the framework unopinionated with respect to the data it could handle. 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 loading/unloading in the browser.

This version of MiddGuard achieved flexibility by automatically loading three types of customizable packages based on location in the file system. 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. Using the VAST 2014 example, they were meant to handle computations like finding other destinations near a point of interest.

Modules were front-end visualizations. JavaScript and CSS files required for the visualization were declared in a package's `manifest.json` file. The database was accessible on the front-end, with each table represented by a Backbone.js collection. Modules could access collections using the global `middguard.entities` object. Collections were updated in real-time over WebSockets, so by listening to changes in a collection or the models it contained, a visualization could update in real-time based on changing data on the server. By extending a base MiddGuard View [2] and registering with MiddGuard by calling a function `middguard.module(name, constructor)`, visualizations could be loaded and unloaded from the window with a button click from MiddGuard's sidebar.

Models were the final piece of customization, intended to give the database flexible schema to work with any data. Models were a combination of database migrations and a Bookshelf.js [7] model declaration. MiddGuard included a `migrate` command to migrate models on a table-by-table basis, applying the results a single database. The model declarations were patched to emit WebSocket events on create, read, update, and delete events so that analytics packages could be written to use the models, persist changes, and communicate those changes to connected clients without needing to explicitly alert connected clients. Relationships could be established between models using a special `relationship` table that stored pairs of table names and row ids.

2.2.2 View Reference Counting

1. MiddGuard Version 1 (Summer 2014)

    (a) Reusable components

    (b) Award/VAST 2015 challenge

CHAPTER 3

IMPLEMENTATION

3.1 Concepts

3.1.1 Data-flow Programming

At the core of MiddGuard is a data-flow model meant to formalize the analytic process. The highest unit of abstraction for the data-flow model is a graph. Analysts can create named graphs from the "Graphs" panel in MiddGuard's sidebar by focusing the "Graph name" input, entering a name, and clicking "Create". The new graph immediately appears in the list of graphs below the input for the client who created it and for all other connected clients. Clicking the graph opens a graph editor specific to the selected graph. Multiple editors can be open at once, with the editors stacking on the screen to the right. Figure 3.1 shows the sidebar with the "Graphs" panel in view and multiple graphs already created and listed below the new graph input.

Graphs are made up of nodes and edges that connect the nodes. In MiddGuard's vocabulary, these are called nodes and connections. Nodes are instantiated from modules, by clicking on one of the modules listed in the left panel of the graph editor. A sample list of modules in this configuration can be seen in figure 3.2. Modules are discussed further in the subsection on extensibility.

Nodes are at the heart of the data-flow model's implementation. In this section we will address the implementation of analytic nodes, seen with blue outlines in figure 3.2. Visualization nodes and their differences with respect to analytic nodes will be addressed in the subsequent section on visualization nodes.

Each node is representative of a function, called the handler, and the node's dynamically generated context. The handler is a function exposed by the node's backing

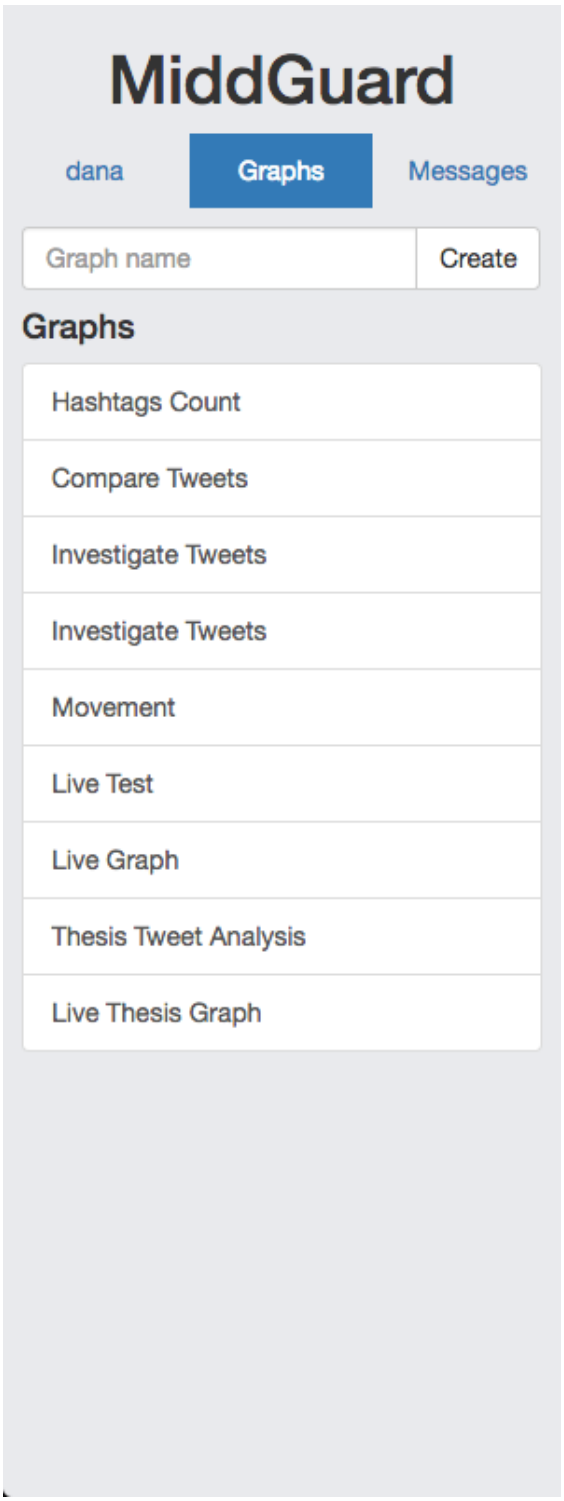

Figure 3.1: MiddGuard's built-in sidebar with the "Graphs" panel in view.

module that performs a data transformation. Each nodes' respective context has an input and an output component. In terms of inputs, the context is a mapping from the names a node gives its own inputs to names of the outputs of its incoming connections. For a node's output, the context is a mapping from a generically named variable, `output`, to a specifically named table in the database.

A node's inputs work at two levels, input groups and column-level connections. A set of column-level connections is encompassed by an input group. Input groups work at the table level, making a configuration based connection between two tables in the database. The column-level connections map columns in an output table to columns in an input table.

Each node has a corresponding table in the database. A mapping from an input group to an output node creates a mapping from the name the input node (the owner of the input group) has given that group to the table represented by the output node. Within each input group are several column mappings, also mapped from the names of input columns given by the input node to the names of columns in the output node's table.

The connections generated within the graph editor are stored in MiddGuard's table of nodes, as a JSON string in the same row as their corresponding input node. Listing 3.1 shows an example connection configuration for one of the "Time by Day/Hour" nodes in the figure 3.2. The connections for "Time by Day/Hour" only have one input group, called "tweets", which is connected to the output node with id `9`. Node `9` is the "@DanaRSilver" node. The `output_node` field serves as a foreign key referencing another row in the same table. Also stored within the input group are the column-level connections between the input group and output node `9`. These are stored in an array of objects called `connections`. Each object within the `connections` array has an 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"
      }
    ]
  }
}
```

Listing 3.1: A node's connection configuration. The node has a connection from its input group "tweets" to the node with id 9.

We considered multiple factors when deciding how to store nodes and their connections in the database before deciding on a JSON string stored with the connections' input node. 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 associate 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.

A node's connections can be edited in the graph editor until runtime, when a node's handler funtion is executed. At this point, MiddGuard makes a query for the node in the database and retrieves its stored connections. Parsing the connections JSON string 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 3.2 is a sample of the context passed into one of the "Time by Day/Hour" nodes visible in figure 3.2. 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 [8], 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` doesn't need a column mapping, since the column names are the same as the ones the node has assigned itself as outputs.

```
{
  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'
    }
  }
}
```

Listing 3.2: The context passed into a "Time by Day/Hour" node's handler function.

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 connections JSON string. We decided against duplicating the data and in favor of making additional 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 connections string in all other connected nodes.

Figure 3.2: 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.

3.1.2 Visualization Nodes

Visualization nodes, like analytic nodes, are added from modules in the graph editor. Figure 3.2 has a visualization node with the label "Hours Heatmap" with an orange outline. Like analytic nodes, visualization nodes 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 an analytic node is a single function that is called and run on MiddGuard's back-end server, while the handler for a visualization node is a newly instantiated Back-

bone.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, less the output `table`, since a visualization node's output is a visualization, rather than a table of data.

Additionally, the `knex` key for each input group is replaced with an instance of a Backbone Collection (with a key aptly named `connection`), which can be used like the `knex` key to access the data from 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.

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.

3.1.3 Visual Programming

Visual programming 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 visual programming 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 programming interface takes place in the three panels of the graph editor, seen in figure 3.2. 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 3.3 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 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’ action. In an analytic node clicking play calls its handler function. In a visualization node, the play button creates a new instance of a visualization. Pressing a visualization node’s play button again removes that visualization from the browser window. Like the graph editors, stack horizontally in the browser window. The user can scroll through them from left to right.

While web scrolling is typically vertical, we implemented view layout horizontally, since MiddGuard was designed to be used on the same system used for the preliminary

VAST 2014 and VAST 2015 investigations. These investigations used a system of three 27 inch displays arranged side by side [1].

An output node can be connected to another node's input group by clicking two of the red and green circles on different nodes that represent connections. Figure 3.3 shows a node with one green input group circle at the top of the node and a red output circle at the bottom of the node.

Since each node is backed by a table in MiddGuard's relational database, connections between two nodes, visualized by a line between one node's output and another's input, represent data flowing out of one or more nodes' backing table into another node's table. For example, in figure 3.2, data flows out of the tables for nodes "@jack" and "@dana" into the table for the node "Diff Two Means".

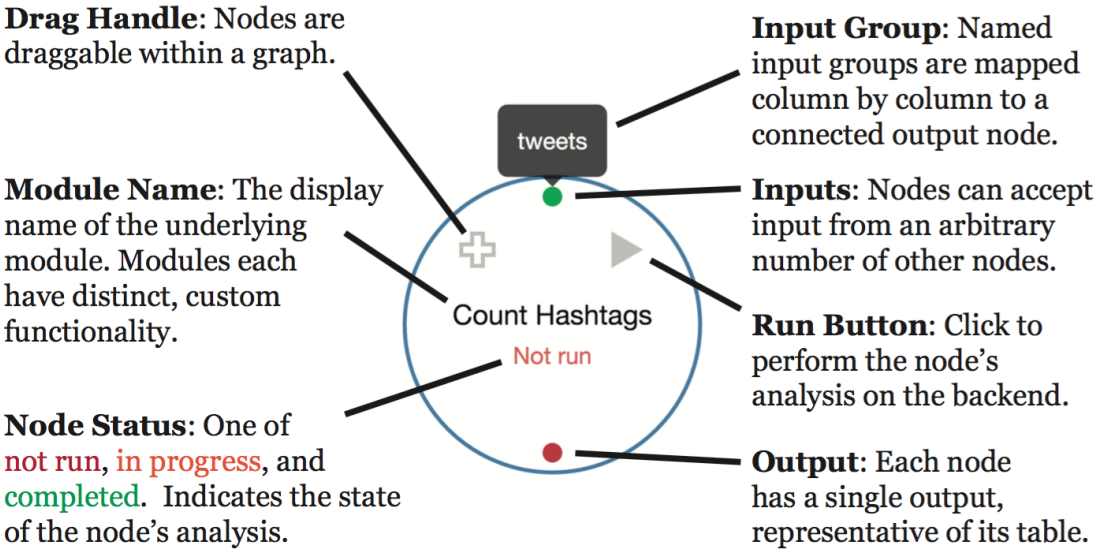

Figure 3.3: An analytic node in a graph. Important features are annotated and the node's only input group, "tweets", is moused over to show its accompanying tooltip.

3.2

3.2.1 Real-time collaboration

1. Dataflow programming
2. Visual programming
3. Collaborative/real-time
4. Extensibility
    (a) Front-end views
    (b) Back-end analytic nodes
5. Reusable nodes (agnostic of input/output data)
6. Technology choices

3.3 Technology

1. Node.js
2. Relational database (instead of NoSQL)
3. Socket.io
4. ORM
5. Backbone.js

CHAPTER 4

RESULTS

1. Test/example case (tweet analysis)
2. Ease of use for developer
3. APIs exposed on front-end and back-end

CHAPTER 5

DISCUSSION

1. Use in real investigation (VAST Challenge 2016)
2. Room for improvement
    (a) Better caching on front-end to help developers optimize DOM/memory usage
    (b) Have to see if there is any else I don't have time to implement

CHAPTER 6

CONCLUSION

1. Revisit points from previous sections
2. Why MiddGuard is an important visual analytics tool
3. Open source prospects

APPENDIX A

CHAPTER 1 OF APPENDIX

Appendix chapter 1 text goes here

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] Mike Bostock and Calvin Metcalf. The topojson format specification. `https://github.com/mbostock/topojson-specification/blob/master/README.md`, 2013.

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

[5] VA Community. Vast challenge 2014: Mini-challenge 2. `http://www.vacommunity.org/VAST+Challenge+2014%3A+Mini-Challenge+2`, 2014.

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

[7] Tim Griesser. Bookshelf.js. `http://bookshelfjs.org/`, 2014.

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

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