Securities Tracking, Observation, and Computational Knowledge System: Documentation and Development Report

Sean Gallagher
CS-340: Client-Server Development
Southern New Hampshire University

Professionals in the Financial Services sector must track and account for a milieu of statistics and data in the course of the daily conduct of business. Fortunately, computer systems present an opportunity to facilitate the acquisition and analysis of this knowledge in new and compelling ways. The Securities Tracking, Observation, and Computational Knowledge System[1] is one such system—one which aims to revolutionize the financial services sector. This report documents the system, its features and functions, and the interface it exposes to enable those ends. The system is built on Node.js in a modern Linux operating environment, familiar to both administrators and engineering teams. This allows developers to more flexibly move from back end to front end development and focus on creating powerful, responsive interfaces for their users.

This documentation consists of two sections. The first covers administration of the system back end through the life cycle of included data. Following a discussion of database initialization and indexing, the report will cover the basic command line tools included with the product and their applications in manipulating and validating the data stored within the installed system. The second section details the Web Service API. This API—which is intended to be the primary interface for both daily use and administration of the system—includes endpoints for basic database manipulation and the two specified reporting interfaces. Each section includes detailed descriptions of operations involved in system administration, as well as screenshots illustrating operations and their results. Names of variables, options, executable scripts or programs, and short commands meant to be entered at a Linux or MongoDB shell prompt are set inline in `monospace`[2]. Long commands, especially those that extend across more than one line, will be set according to the following convention:

```
> command [optional value] <mandatory value>
```

[1] Yes, that's quite a mouthful. We know; we're working on it.

[2] Note that this document assumes that the `cs340-project/bin` directory is in the user's `$PATH`, and omits '`./`' from any such commands.

Installing, Initializing, and Administering the System

The Securities Tracking, Observation, and Computational Knowledge System[3] is built on MongoDB, a document-oriented NoSQL database system, and Node.js, a server-side runtime for modern JavaScript based on Google's V8 JavaScript engine. It is designed and tested to run on modern Linux platforms—development and testing was done on Ubuntu 20.04 LTS, though any recent Linux distribution should suffice. As a full discussion of the pertinent concepts relating to each of these technologies could fill several textbooks, this document assumes the prospective administrator is familiar with installing and maintaining a Linux system; installing, configuring, and securing a MongoDB server instance; and basic usage of Node's `npm` package manager. Once these dependencies are met, issue the following command to download the system files:

```
> git clone --depth 1 https://github.com/seangllghr/cs340-project
```

As with any `git clone` operation, you can specify a custom installation directory at the end of the command. Once the files have been downloaded, enter the installation directory and run `npm install` to install the necessary Node dependencies

System Initialization and Configuration

Before the installed system can be used to generate insights into financial securities markets, it must be configured and populated with data. This process consists of three principal steps: (a) importing the data and configuring the application to access the database, and (b) indexing the database for performance. Each of these steps plays a vital role in ensuring that the configured application suits the needs of the front-end development team and the system's target users.

Database creation and import

Once the application is operational, data will be managed and manipulated using the Web Service API; however, the database should be populated during initial configuration using data stored in

[3] We really need a new name for this...

```
[sean@Jotunheim] ~/.../cs340/datasets >>> ?mongoimport
zsh: no matches found: ?mongoimport
stocks.json
Enter password:

2020-06-19T12:45:35.342-0400    connected to: mongodb://localhost:27017/
2020-06-19T12:45:36.603-0400    6756 document(s) imported successfully. 0 document(s) failed t
o import.
[sean@Jotunheim] ~/.../cs340/datasets >>> mongoadm
MongoDB shell version v4.2.7
Enter password:
connecting to: mongodb://127.0.0.1:27017/?authSource=admin&compressors=disabled&gssapiServiceN
ame=mongodb
Implicit session: session { "id" : UUID("e2013246-94c0-4410-a797-95f833fe3005") }
MongoDB server version: 4.2.7
Mongo-Hacker 0.1.1
> use market
switched to db market
> db.stocks.findOne({})
{
  "_id": ObjectId("52853800bb1177ca391c1802"),
  "Ticker": "AAIT",
  "Sector": "Financial",
  "Change from Open": -0.0232,
  "Performance (YTD)": 0.1048,
  "Performance (Week)": -0.0097,
  "Performance (Quarter)": 0.1129,
  "200-Day Simple Moving Average": 0.0492,
  "52-Week High": -0.0641,
  "Change": -0.02,
  "Volatility (Week)": 0.014,
  "Country": "USA",
  "50-Day Low": 0.0609,
  "Price": 29.95,
  "50-Day High": -0.0641,
  "Dividend Yield": 0.036,
  "Industry": "Exchange Traded Fund",
  "52-Week Low": 0.1791,
  "Average True Range": 0.45,
  "Company": "iShares MSCI AC Asia Information Tech",
  "Gap": 0.0033,
  "Relative Volume": 0.17,
  "Volatility (Month)": 0.0111,
  "Volume": 250,
  "Short Ratio": 0.04,
  "Performance (Half Year)": 0.0307,
  "Relative Strength Index (14)": 38.79,
  "20-Day Simple Moving Average": -0.0295,
  "Performance (Month)": 0.003,
  "Performance (Year)": 0.1947,
  "Average Volume": 1.64,
  "50-Day Simple Moving Average": -0.0065
}
>
```

Figure 1: Importing data from a JSON file into the database and verifying correct import from the MongoDB shell.

Figure 2: Creation of a simple ascending index on the `ticker` field.

JSON file. While the specifics of the application will dictate the details of the schema, the included utilities and API expect each stock record to have the following fields:

- `Ticker` (unique)
- `Sector`
- `Industry`
- `Volume`
- `50-Day Simple Moving Average`
- `Analyst Recom`

Figure 1 depicts the process of importing seed data into the database instance using the `mongoimport` command. While the specific construction of the command will depend on the environment and configuration of the MongoDB server, the general form is:

```
mongoimport \
[--host=<host>] [--port=<port>] \
[authentication opts] \
[--db=<db>] [--collection=<collection>] \
<file>
```

Once the `mongoimport` tool has finished importing data from the JSON file, log into the MongoDB shell and issue a simple `findOne()` query to verify that the data has imported correctly, as illustrated in the figure. After verifying that the database has imported correctly, set the `dbName` and `colName` values in the the `config.json` file in the application's root directory to enable the application back end to access the database.

Index Design and Implementation

Index design and implementation is, like the broader topic of general MongoDB administration, too vast to cover in this report. While the development team creating the specific queries requested for a given application will know best what indices will offer the greatest performance benefit for their

```
> db.stocks.createIndex({'Industry': 1, 'Analyst Recom': -1})
{
  "createdCollectionAutomatically": false,
  "numIndexesBefore": 2,
  "numIndexesAfter": 3,
  "ok": 1
}
>
```

Figure 3: Creation of a compound index on the `Industry` and `Analyst Recom` fields will increase performance for the Industry Report query.

application, some generalizations can be offered. As searching for a specific stock or security will likely be a staple of most analysts' workflows, a simple index on the `Ticker` field will benefit most applications. Creation of such an index is straightforward: after logging into the MongoDB shell and selecting the application database, issue the following command (illustrated in Figure 2:

```
db.<collection>.createIndex({"Ticker": 1})
```

In addition to simple indices, one or more compound indices may be necessary for optimal performance of a given application. A variety of considerations influence the order in which indexed fields appear in a compound index, and these factors are highly dependent on the queries required to generate reports used in a given implementation. To illustrate this, take the example Industry Report provided with this software. While a full description will be included in the API section of this document, the report, in brief, returns a list of the top 5 stocks in a user-supplied industry. A compound index across the `Industry` (ascending) and `Analyst Recom` (descending) fields (shown in Figure 3) will improve the performance of this report considerably.

Command Line Tools and Library Functions

The Securities Tracking, Obs—okay, you know what, just call it STOCKS[4]—ships with several tools for manipulating the database back end from the command line. These tools are not meant to be exhaustive replacements for the MongoDB shell; rather they aim to facilitate some common interactions and provide examples of how the utilities and scripts can harness the internal application architecture to enhance and automate aspects of the typical administrative workflow.

[4] I'll see myself out

Library Functions

Two libraries of utility functions are provided with STOCKS. The first, `jsonUtils`, provides functions for loading and JSON stock and security data. While Node provides native functions for manipulating JSON data through the `JSON` object, the `jsonUtils` library provides abstractions for creating a string from a stream, parsing that stream into a native JavaScript object, and scanning for and replacing query terms, such as `$oid` and `$date` with their respective native JavaScript objects.

The other library, `cli-utils`, provides two generalized prompt functions. Because Node.js relies heavily on asynchronous programming techniques, synchronous I/O akin to what is offered by other common server-side languages is quite a bit more complex in Node. STOCKS relies on an outside library to abstract away that complexity, and offers two straightforward prompt functions—`promptForString()` and `promptForNumber()`—that mimic traditional synchronous I/O for command line tools.

Aside: `stock-client.sh`, a Rudimentary API Front End

```
Usage: stock-client delete [ticker]
       stock-client industry-report [industry]
       stock-client insert <JSON/YAML stock document>
       stock-client read [ticker]
       stock-client stock-report <JSON/YAML ticker list>
       stock-client update <ticker> <JSON/YAML update document>
```

Take a look in the `bin` folder, and one file stands out as immediately different than the rest: `stock-client.sh`. This script, written in Bash and reliant on several Linux command line tools, started as a platform that the STOCKS development team used to test the STOCKS Web Service API. Over the course of this testing, `stock-client.sh` gradually expanded in scope, to the point where it is effectively a full-featured command line client for the STOCKS Web Service API. It provides this interface using an executable-command-operand syntax that should be familiar to most Linux users, summarized above.

```
[sean@Jotunheim] ~/…/datasets >>> mongo --authenticationDatabase "admin" -u "administrator" -p

MongoDB shell version v4.2.7
Enter password:
connecting to: mongodb://127.0.0.1:27017/?authSource=admin&compressors=disabled&gssapiServiceN
ame=mongodb
Implicit session: session { "id" : UUID("5cd47119-3220-4453-9711-e3112714893b") }
MongoDB server version: 4.2.7
Mongo-Hacker 0.1.1
> use market
switched to db market
> db.stocks.findOne({"Ticker": "GOOG"})
{
  "_id": ObjectId("52853804bb1177ca391c2221"),
  "Ticker": "GOOG",
  "Profit Margin": 0.217,
  "Institutional Ownership": 0.862,
  "EPS growth past 5 years": 0.196,
  "Total Debt/Equity": 0.06,
  "Current Ratio": 4.8,
  "Return on Assets": 0.125,
  "Sector": "Technology",
  "P/S": 6,
  "Change from Open": 0.0035,
  "Performance (YTD)": 0.4596,
  "Performance (Week)": 0.0095,
  "Quick Ratio": 4.7,
```

```
_id: 52853804bb1177ca391c2221
Ticker: GOOG
Profit Margin: 0.217
Institutional Ownership: 0.862
EPS growth past 5 years: 0.196
Total Debt/Equity: 0.06
Current Ratio: 4.8
Return on Assets: 0.125
Sector: Technology
P/S: 6
Change from Open: 0.0035
Performance (YTD): 0.4596
Performance (Week): 0.0095
Quick Ratio: 4.7
Insider Transactions: -0.8138
P/B: 4.15
EPS growth quarter over quarter: 0.346
Payout Ratio: 0
Performance (Quarter): 0.201
Forward P/E: 19.8
P/E: 29.66
200-Day Simple Moving Average: 0.1928
Shares Outstanding: 333.62
Earnings Date: "2013-10-17T20:30:00.000Z"
52-Week High: -0.0039
P/Cash: 6.09
:
```

Figure 4: `stock-client read` offers a user-friendly, paged alternative to manually verifying the following command-line utilities from the MongoDB shell.

Aside from Bash, the client depends on a number of *nix command line tools to process input and server responses. At the client's core, `curl` is used to access the STOCKS Web Service API. The `read`, `delete`, and `industry-report` commands take input as a simple string, but the `insert`, `update`, and `stock-report` commands take input in the form of a stock record document, update document[5], and a list of `Ticker` symbols. Using a tool called `yq`, a command line YAML processor, the client is able to accept both JSON and YAML for these input documents. The client also uses `yq` to pretty-print (as human-friendly YAML) the output of commands that return JSON data. Additionally, the client filters the stock and industry reports using another tool called `jq` to present only the most relevant information to the user, and `less` is used for paging. Assuming these dependencies can be satisfied, the client *should* run on any system with a Bash shell. The client has been tested successfully on bare-metal Ubuntu 20.04 and Windows Subsystem for Linux 2 (also running Ubuntu 20.04). The client is written to interface with a STOCKS server running on `localhost` by default. There is currently no run-time option or configuration file to specify a server; however, the single variable `hostname` can be set within the script itself to allow `stock-client` to access a remote STOCKS server.

The second section of this report goes into detail about the specific API endpoints and operations supported by the client; additional discussion in this section would be redundant. However, because the interface is straightforward, the output concise, and the client offers paged output by default for full-record read operations, the author recommends that administrators use `stock-client read` to verify correct functioning of the command line tools in this section. Figure 4 illustrates the difference between a Mongo Shell `find` operation and the `stock-client read` command—namely, that there is essentially no difference, except for the significant keystroke savings and quality of life improvements for users of the latter. Screenshots from `stock-client read` are used to verify the command line tools and API calls throughout this document, primarily for space-saving reasons.

[5] Technically, `update` also takes a `Ticker` string, as well.

```
[sean@Jotunheim] ~/…/bin >>> ./insert-stock.js -f ../../datasets/stock2.json ±[●][docs-final]
Stock inserted into db with ticker QX
[sean@Jotunheim] ~/…/bin >>>                                                 ±[●][docs-final]
```

Figure 5: Inserting a stock record from a JSON file with `insert-stock`.

Creating New Stock Records with `insert-stock`

```
Usage: insert-stock.js [-f <json file>]
```

The `insert-stock` utility provides an interface for quickly inserting new stock and security records into the database from the command line. The tool allows the user to manually enter stock data line-by-line, terminating input with the End-of-Document (Ctrl-d) character, or accepts input piped from another command in the usual *nix way. Additionally, with the `-f` flag, the tool will read input from a file on disk. Whether input is passed from a file, pipe, or direct input, `insert-stock` expects to read data in standard JSON format. Like MongoDB, it will process `$oid` and `$date` fields into native Mongo/Javascript `ObjectID` and `Date` objects. Other fields may not parse correctly; however, it should be trivial to add rules for additional field types in the `convertMongoQueryFields` function in `jsonUtils.js`, discussed later.

Internally, the utility wraps around `db.dataCreate`, which provides a simplified interface to the MongoDB driver `insertOne` and `insertMany` methods, automatically selecting the appropriate method for the document object passed as an argument.

Figure 5 illustrates the utility being used to insert a new stock record[6] being inserted into the database, while Figure 6 confirms that the record has been inserted successfully.

Updating Trade Volumes with `update-volume`

```
Usage: update-volume [-t <ticker>] [-v <volume>]
```

Where `insert-stock` has a broad purview, `update-volume` is a tightly-focused single-purpose tool. It wraps the `db.dataUpdate` function in a simple interface, prompting the user for a ticker symbol and updated value and updating the database accordingly. Because `db.dataUpdate` expects

[6] Which is definitely not just a copy of the record for China Zenix Auto International (ZX) with the first letter of its ticker symbol and first character of its ObjectID changed...

its input as a key-value object, the wrapper includes a bit of logic to transform the input—which can optionally be passed at command invocation with UNIX-style flags—into appropriate query and update documents.

Figure 7 depicts both the tool in operation and its verification by piping output from `stock-client read` through `grep` to focus on only the lines containing the stock's ticker symbol and volume data.

Deleting Existing Stock Records with `delete-ticker`

```
Usage: delete-ticker [-t <ticker>]
```

At this point, astute readers might be wondering, "but now that I've added this worthless dummy stock record and changed one value, how do I remove it from the database so it doesn't pollute my analysts' reports?" Fortunately, the `delete-ticker` utility gives administrators a simple, effective tool to do just that. The utility wraps around the `db.dataDelete` function, which it supplies with a query on the ticker symbol provided by the user. Like `update-volume`, `delete-ticker` can take

```
_id: 42853810bb1177ca391c3262
Ticker: QX
Profit Margin: 0.064
Institutional Ownership: 0.208
EPS growth past 5 years: 0.433
Total Debt/Equity: 0.26
Current Ratio: 1.3
Return on Assets: 0.053
Sector: Consumer Goods
P/S: 0.35
Change from Open: 0.0052
Performance (YTD): 0.2776
Performance (Week): -0.0545
Quick Ratio: 1.1
P/B: 0.5
EPS growth quarter over quarter: -0.385
Payout Ratio: 0
Performance (Quarter): 0.2949
Forward P/E: 3.7
P/E: 5.46
200-Day Simple Moving Average: 0.2125
Shares Outstanding: 51.63
Earnings Date: "2013-11-20T13:30:00.000Z"
52-Week High: -0.103
P/Cash: 1.36
Change: 0.0026
:
```

Figure 6: Verifying that the stock QX inserted correctly.

```
[sean@Jotunheim] ~/…/bin >>> stock-client read QX | grep '\(Ticker\)\|\(Volume\)'
Ticker: QX
Relative Volume: 0.05
Volume: 1103
Average Volume: 22.69
[sean@Jotunheim] ~/…/bin >>> ./update-volume.js -t QX -v 1234              ±[●][docs-final]
Updated Ticker: QX Volume: 1234
[sean@Jotunheim] ~/…/bin >>> stock-client read QX | grep '\(Ticker\)\|\(Volume\)'
Ticker: QX
Relative Volume: 0.05
Volume: 1234
Average Volume: 22.69
[sean@Jotunheim] ~/…/bin >>>                                                ±[●][docs-final]
```

Figure 7: Using `update-volume` to update the `Volume` field on our dummy stock, 'QX'.

input in the form of a command line flag, or it will prompt the user to enter a ticker symbol if one is not provided. Figure 8 illustrates the use of `delete-ticker` without providing a ticker at the command line.

`sma-spread`, a Demonstration of Find Queries

```
Usage: sma-spread.js [-l <low value>] [-h <high value>]
```

Read operations are the bread and butter of an analytical database system such as STOCKS. While administrators might spend more time with document insertion and manipulation, the bulk of the system's user base will inevitably be analysts, tasked with querying the compiled data and drawing conclusions about the appropriate direction to take the company and its clients in the days and weeks to come. In developing STOCKS, it became clear early on that no set of queries

```
[sean@Jotunheim] ~/…/bin >>> stock-client read QX | head                   ±[●●][docs-final]
_id: 42853810bb1177ca391c3262
Ticker: QX
Profit Margin: 0.064
Institutional Ownership: 0.208
EPS growth past 5 years: 0.433
Total Debt/Equity: 0.26
Current Ratio: 1.3
Return on Assets: 0.053
Sector: Consumer Goods
P/S: 0.35
[sean@Jotunheim] ~/…/bin >>> ./delete-ticker.js                            ±[●●][docs-final]
Ticker:
> QX
Deleted stock record for Ticker QX
[sean@Jotunheim] ~/…/bin >>> stock-client read QX | head                   ±[●●][docs-final]
Ticker symbol not found.
[sean@Jotunheim] ~/…/bin >>>                                               ±[●●][docs-final]
```

Figure 8: Deleting the 'QX' stock record with `delete-ticker`. Again, read output is piped for brevity. Note that this usage omits the command line flag, opting to use the prompt for ticker entry, instead.

developed by the back-end team could—or even should—fulfill all foreseeable needs encountered by the analysis team; instead, we provide examples, in the hopes of empowering the analysts to find their need and fill it.

While most analysts will choose to leverage the tools made available by the front-end team or work directly with the STOCKS Web Service API to construct these queries and gather this data, there remains a need for command line tools for analysis. To this end, `sma-spread` should be viewed not so much as a tool in itself, but rather as a blueprint of what a command line tool built on Node and the STOCKS libraries might look like. The `sma-spread` follows the pattern established in the tools presented earlier. It wraps a thin layer of logic around the `db.countMatching` function to transform user input into a query object that can be passed to the data access layer directly. In this case, that query object is a simple range query, using MongoDB's `$gt` and `$lt` conditionals to search for Simple Moving Average values within user-specified bounds. For brevity, this function only returns a count of matching values; however, it would be trivial to amend this to return a list of values using `db.dataRead`[7].

Generating a List of Ticker Symbols for a Given Industry

```
Usage: tickers-by-industry [-i <industry>]
```

As with any tool built on a powerful, feature-rich back end like MongoDB, it would be impossible to expose all of the low-level functionality of the Node MongoDB driver in the STOCKS command line tools or Web Service API. Writing tools in such a framework would hardly be better than developing with the MongoDB driver API directly. However, in abstracting away the complexities

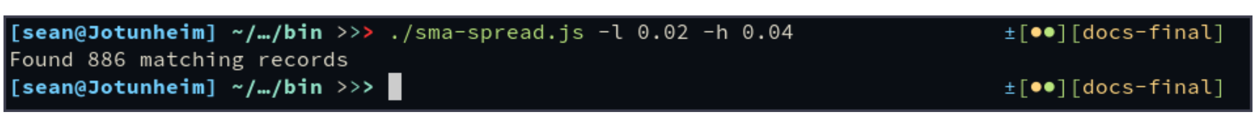

Figure 9: The `sma-spread` tool in action.

[7] At which point, the author hopes, they would be processed, and not just dumped to `stdout`.

of connecting to and manipulating the MongoDB database, some functionality inevitably falls by the wayside.

The `tickers-by-industry` utility is a perfect example of this sort of compromise. In the interest of simplifying the method signature for `db.dataRead`, we chose to omit the option to limit what fields the query returns. As a result, logic to limit the returned results from `tickers-by-industry` relies on a simple JavaScript loop to print only the stock ticker symbols to the console. This sort of client-side workaround isn't ideal, as there is a performance cost incurred as the script iterates over the returned array, but that cost is often negligible. Figure 10 illustrates the `tickers-by-industry` utility retrieving a ticker list for the industry matching the most results, showing a total execution time of just under a second on the author's decade-old workstation. While there are certainly cases where client-side processing would place undue burden on the client, or where server-side processing can leverage significantly more performant technologies, the cost often does not justify the added complexity in back-end code.

Generating a List of Outstanding Shares by Industry

```
Usage: agg-shares.js [-s <sector>]
```

The `agg-shares` utility follows a similar pattern to the other string-fed[8] tools presented above. It takes input from a user prompt or, optionally, a command line argument. In this case, `agg-shares`

```
[sean@Jotunheim] ~/…/bin >>> time ./tickers-by-industry.js -i 'Exchange Traded Fund' | head
AAIT
AADR
AAXJ
ACCU
ACIM
ACWI
ACWX
ADRA
ADRE
ADRU
./tickers-by-industry.js -i 'Exchange Traded Fund'  0.95s user 0.10s system 111% cpu 0.945 total
head  0.00s user 0.00s system 0% cpu 0.745 total
[sean@Jotunheim] ~/…/bin >>>
```

Figure 10: The `tickers-by-industry` utility in action.

[8] Not to be confused with grain-fed. We raise our command line utilities using traditional techniques, feeding them a healthy diet of caffeine and suffering like civilized programmers.

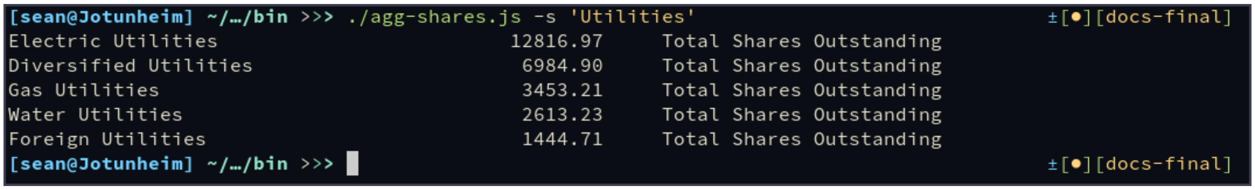

Figure 11: `agg-shares` operating on the "Utilities" industry (chosen for brevity).

asks for a sector, and returns a sorted list of industries in that sector, along with the total number of shares outstanding in the industry. This utility illustrates, to a small extent, what can be done with MongoDB's aggregation framework, one of its most powerful features. Because the STOCKS application's data access layer function `db.dataAggregate` accepts MongoDB aggregation pipeline arrays directly, any operation that can be conducted as part of an aggregation pipeline can be executed through the STOCKS aggregation functionality.

While a full discussion of MongoDB's aggregation framework is well outside the scope of this document, a brief overview of the functions utilized in `agg-shares` is both reasonable and warranted. Three aggregation stages are used in this pipeline: (a) a `$match` stage matching records with the `Sector` field value specified; (b) a `$group` stage, which collects records into groups by industry and sums their outstanding shares; and (c) a `$sort` stage, which is responsible for ordering the results in descending order by total shares outstanding. This pipeline is passed directly to the `db.dataAggregate` function, which returns the results. Additional processing occurs on the client side; the utility uses a JavaScript implementation of C's `printf` function to format each result before printing it to screen in nice, neat, orderly columns[9]. Figure 11 illustrates the tool in operation on the "Utilities" sector.

The STOCKS Web Service API

While the command line utilities in the first section offer insight into how STOCKS works under the hood, the meat of the application is dedicated to providing the STOCKS Web Service API.

[9] It's worth noting that `agg-shares` breaks when included in a UNIX pipe. I'm not really sure why this is—it throws a Node error and complains about writing to a pipe, but I'm not entirely sure what's going wrong.

The Web Service API is built on a layered architecture model centered on a separation of concerns. The core of this architecture is the Data Access layer, described in part in the first section, which wraps around the MongoDB driver methods. Above the Data Access layer, a Services layer contains business logic that processes input data before passing it to Data Access functions and output data to be returned to the route's Controller. This Controller contains logic associated with the particular web service framework—in this case, the venerable Express framework. At the most abstract, the Routing layer takes incoming requests and passes them to the appropriate controller for processing.

This architecture is ideal both for a rapidly-growing startup and the fluid ecosystem of Node.js. By writing business logic in simple, standard JavaScript, developers can focus on the logic itself, rather than the vagaries of Express. Conversely, when the tides shift and Express inevitably falls out of favor, developers need only focus on transitioning the routing and request handling logic to whatever new, shiny tech is in vogue. This approach does add some complexity—not counting the command line utilities, the resulting codebase is spread across a dozen files—but the scalability and sustainability of this approach are worth the cost.

Contrary to the breakdown above, this section breaks the API down into its six endpoints, four for the basic CRUD operations fundamental to any database system, along with two examples of more complex read operations. These endpoints are far from comprehensive; like the command line tools, the API presented aims to provide a minimal level of pre-built functionality which can serve as a framework and inspiration for more complex workflows.

A Brief Return to `stock-client.sh`

Because the `stock-client` script is arguably the most distilled form of a client designed to access the STOCKS Web Service API—and because it is written in Bash, not JavaScript—it seems appropriate to revisit it in this section as an example of the versatility of the API. While the client is by no means a complete illustration of what could be done with a dedicated front-end solution, it effectively demonstrates the power of the REST model. As API requests are passed in JSON, rather than as proprietary binary data or native language constructs, the client accessing the REST resources is not constrained to any one language, architecture, or methodology. While it is a simple tool, the `stock-client` script is eminently usable: it offers a concise command syntax; offers readable, paged output; and allows the user to manipulate data once it has been retrieved. That usability and power is, of course, predicated on a familiarity with UNIX tools and concepts. However, a software application *should* tailor itself to its target user. A client for non-technical users could just as easily offer all of the graphical bells and whistles that those users expect while still harnessing the same API.

Basic CRUD Operations

At the core of the STOCKS Web Service API are the four CRUD operations. These four functions form the building blocks that front-end developers can use to build their interfaces. While they do not expose all of the features of the underlying layers, their simplicity makes them powerful in their own right.

`createStock`

```
URI:     http://<hostname>:<port>/api/v1.0/createStock/<ticker>
Method:  POST
```

Before data can be analyzed or modified, it must first be created. The create endpoint empowers users to create these stock records, and is intended to be used primarily in an administrative capacity. A `createStock` request takes the form of a POST request directed at the `createStock/<ticker>` URI. This variable URI field, called a *parameter*, indicates to the system the stock ticker symbol of the record to be created, while the body of the request is a JSON stock record document. Provision of the ticker symbol in the URI provides a modicum of validation; if the ticker in the request body does not match the URI parameter, the request is rejected.

Once this validation is performed, the controller calls `services.stocks.createStockService`, which converts any MongoDB query fields, such as `$oid` or `$date`, into native JavaScript objects of

```
[sean@Jotunheim] ~/…/cs340 >>> stock-client insert datasets/stock2.json
Record created successfully%
[sean@Jotunheim] ~/…/cs340 >>> stock-client read QX |
egrep '(Ticker)|(Company)|(Country)|(Industry)|(Sector)|(Price)|(Volume)'
Average Volume: 22.69
Company: China Zenix Auto International Ltd.
Country: China
Industry: Auto Parts
Price: 3.83
Relative Volume: 0.05
Sector: Consumer Goods
Ticker: QX
Volume: 1103
[sean@Jotunheim] ~/…/cs340 >>>
```

Figure 12: Successful insertion of 'QX', a bald-faced copy of the stock record for China Zenix Auto International Ltd., with filtered `readStock` output for verification.

the appropriate type. It then validates that the stock record does not already exist in the database. Assuming the record is unique, the service calls the data access function `db.dataCreate` to insert the stock record document into the database. The ultimate success of the insert—or its failure at any stage—is returned to the controller, which responds to the original request with code 201 for a successful insert, 400 if the ticker already exists in the database, or 500 if the server encounters another error.

The `stock-client` front end simplifies usage of the `createStock` endpoint by assembling an appropriate `curl` command from the JSON data passed by the user. Figure 12 shows the utility making a `createStock` request with the command:

```
curl -H "Content-Type: application/json" \
     -X POST -d "$inputjson" -s \
     "http://$hostname:3000/api/v1.0/createStock/$ticker"
```

readStock

```
URI:    http://<hostname>:<port>/api/v1.0/readStock/<ticker>
Method: GET
```

The `readStock` API endpoint provides a significant amount of functionality to the end user and front-end developer. Because it returns highly detailed data about a single stock record, it can be used in a targeted query against a single record, or to collect information about several different records using client-side language features, akin to the strategy used by the `tickers-by-industry` command line tool demonstrated in the first section. While more complex queries or searches looking for detailed data on long lists of individual stocks might be better served with a single-request API endpoint utilizing aggregation or complex `find` functionality, the increase in query performance may not justify the increased development effort in many cases. That being said, the sections on the advanced query endpoints `tickerList` and `industryReport` illustrate the ease with which new API endpoints can be added, if necessary.

Requests to `readStock` take the form of an HTTP GET request; the query variable, `Ticker`, is passed through a URI parameter in the final position. This request is routed to `stocks.readStockController`, which extracts the parameter object—which shares the same structure as the desired MongoDB query document—and passes it to the basic read service. This service calls the data access layer directly, and returns the resulting array to the controller. If the array is empty, the controller sends an HTTP 404 response, indicating that the item does not appear in the records[10]. Otherwise, the controller responds with the data as a JSON string.

The `readStock` API endpoint has been demonstrated throughout this document, as the `stock-client` script implements calls to the API in its `read` interface. The `curl` command used by the client is:

```
curl "http://$hostname:3000/api/v1.0/readStock/$ticker" -s
```

updateStock

```
URI: http://<hostname>:<port>/api/v1.0/updateStock/<ticker>
Method: PUT
```

Updating stock records with the Web Service API works similarly to other basic CRUD operations. The URI follows the same structure, with calls to `updateStock` followed by a single positional parameter indicating the target stock ticker. Like requests to `createStock`, `updateStock` takes additional JSON data in the request body; however, unlike `createStock` requests, this data need not contain a matching ticker symbol. In fact, the JSON request body should contain only the fields

[10] And, therefore, does not exist.

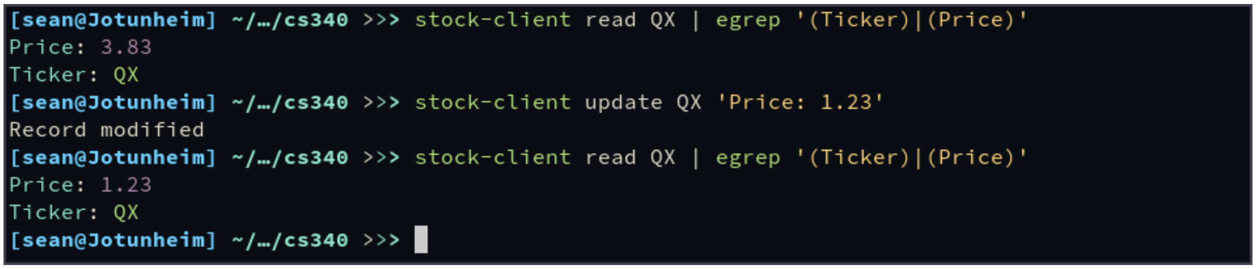

Figure 13: `stock-client` demonstrating successful execution of a call to the `updateStock` endpoint (and YAML document input, but that's not the point).

the user wants to update, with their new values. Processing of this document into a MongoDB `$set` query is handled by the STOCKS application.

After the incoming request is routed to the `stocks.readController`, the controller verifies that the record exists, then extracts URI parameter object and the request body. These objects are passed to the update service as the `query` and `update` parameters, respectively. The query object is already in the correct form to send to the data access layer, but the service sets the `update` parameter as the value in a MongoDB `$set` key-value pair before it is passed to `db.dataUpdate`. Assuming the update completes successfully, the service returns the result to the controller. If the `result` object indicates that no records were updated—such as would be the case if the fields to be updated already contained the values requested—the server responds with a message indicating that the record was not modified; otherwise, the message indicates that the update was successful. In both cases, the response code sent to the client is 200, indicating that the value(s) of the specified fields match those requested.

The `updateStock` endpoint is accessed by `stock-client` using the `curl` command below, and is illustrated in operation in Figure 13.

```
curl -H "Content-Type: application/json" \
     -X PUT -d "$inputjson" -s \
     "http://$hostname:3000/api/v1.0/updateStock/$ticker"
```

deleteStock

```
URI:    http://<hostname>:<port>/api/v1.0/deleteStock/<ticker>
Method: DELETE
```

Like the other CRUD operations, the API endpoint is accessed through an HTTP request with a `Ticker` parameter in the final position. In this case, the request is made as a DELETE request, which takes no additional data. This request is routed to `stocks.deleteStockController`, which extracts the parameter object and passes it to the delete service. This service is a thin wrapper around the `db.dataDelete` function. Once the data access layer returns a MongoDB result object, this result is passed to the controller; on a successful deletion, the server responds to the client with an HTTP 200 response indicating the record was successfully deleted. If the record was not found, it returns code 404.

The `curl` command for this request is essentially identical to that passed to `readStock`, with the addition of a `-X DELETE` flag indicating the request type. Its execution is illustrated in Figure 14.

Advanced Reporting

MongoDB is an incredibly powerful database system, and it offers features which can greatly simplify and enhance complex reporting use cases. It would be impossible to cover all possible reports that a particular customer might desire for even an application as simple as STOCKS; however, the decision to build the STOCKS back end on the Node.js platform makes it straightforward for front-end developers with JavaScript skills to enhance the back end with additional reports, as necessary. The two reports included in this section are designed to provide examples of the possibilities the system offers and inspiration for future front- and back-end developers alike.

`stockReport`

```
[sean@Jotunheim] ~/…/cs340 >>> stock-client read QX | egrep '(Ticker)|(Price)'
Price: 1.23
Ticker: QX
[sean@Jotunheim] ~/…/cs340 >>> stock-client delete QX
Record successfully deleted.
[sean@Jotunheim] ~/…/cs340 >>> stock-client read QX | egrep '(Ticker)|(Price)'
Ticker symbol not found.
[sean@Jotunheim] ~/…/cs340 >>>
```

Figure 14: A call to the `deleteStock` endpoint successfully deleting a stock record from the database.

```
http://<hostname>:<port>/api/v1.0/stockReport
```

The `stockReport` API endpoint presents a common use case for more advanced reporting functionality; rather than seeking detailed information on a single stock, a user is likely to want to compare data about several stocks. This API facilitates these queries by allowing the user to enter a list of ticker symbols[11], which is transformed into query that leverages MongoDB's `$in` operator.

`stockReport` is the only API endpoint in the application that doesn't take a positional parameter; instead, it takes input, in the form of a JSON array, as the body of an HTTP POST request to the base URI. This array, extracted from the request body by the controller, is passed to the service layer, where it is inserted into a MongoDB `$in` query object, which is passed to `db.dataRead`. The array of results is returned to the controller, which sends it in the body of an HTTP 200 response to the client. This approach leaves further filtering of displayed fields to the client, focusing instead on streamlining data retrieval for multiple stock tickers.

The sample client included with the STOCKS application provides an example of this client-side data processing, passing the returned data through `jq` to limit the number of fields displayed on the terminal. However, a client could execute commands of arbitrary complexity on the returned data, depending on its needs. Figure 15 shows an example of the `stockReport` API endpoint and this client-side processing.

`industryReport`

```
http://<hostname>:3000/api/v1.0/industryReport/<industry>
```

The `industryReport` endpoint offers an example of injecting user input into a pre-constructed aggregation pipeline designed to generate a specific type of report. It takes a single positional parameter, an `Industry` string, and returns a list of the top five stock choices in that industry. The actual logic to do this is surprisingly simple, making this an excellent example of the ways that a team could extend the application to suit their needs.

[11] Technically, a JSON array.

```
[sean@Jotunheim] ~/…/cs340 >>> stock-client stock-report '
- AMZN
- GOOG
- MSFT
'
Running stock report on ["AMZN","GOOG","MSFT"]
- Ticker: AMZN
  Company: Amazon.com Inc.
  Sector: Services
  Industry: Catalog & Mail Order Houses
  Price: 367.23
- Ticker: GOOG
  Company: Google Inc.
  Sector: Technology
  Industry: Internet Information Providers
  Price: 1037.5
- Ticker: MSFT
  Company: Microsoft Corporation
  Sector: Technology
  Industry: Business Software & Services
  Price: 38.04
[sean@Jotunheim] ~/…/cs340 >>>
```

Figure 15: `stock-client` running a stock report request to the `stockReport` API endpoint. Note that data processing is happening on the client side with `jq`.

After receiving the routed request, the controller extracts the industry string from the URI parameter and passes it to the service layer. The service layer receives this string and inserts it into a three-stage MongoDB aggregation pipeline array. This pipeline consists of a `$match` stage targeting the user-supplied industry string, a `$sort` stage that sorts by analyst recommendation rating, and a `$limit` stage that returns only the top five results. This pipeline is passed to `db.dataAggregate`, which performs the database operation and returns the resulting array to the service layer, which sends it to the controller. Assuming the operation executes successfully, the controller sends the array to the client as a JSON string.

This example is an excellent demonstration of the extensibility of the STOCKS platform—not only because it shows *how* the system can be extended to cover additional use cases by the front-end team or eventual end users, but because it shows *why* those users may want to develop such functionality themselves. The field of financial services is broad, complex, and technical, and this developer has essentially no knowledge of what makes a stock a “top stock”. Relying on a single number, even a “recommendation,” is a poor substitute for the detailed understanding that

```
[sean@Jotunheim] ~/…/cs340 >>> stock-client industry-report 'Business Software & Services'
- Ticker: VELT
  Company: Velti Plc
  Sector: Technology
  Industry: Business Software & Services
  Price: 0.15
  Analyst Recom: 3.3
- Ticker: JKHY
  Company: Jack Henry & Associates Inc.
  Sector: Technology
  Industry: Business Software & Services
  Price: 56.39
  Analyst Recom: 3.1
- Ticker: CSPI
  Company: CSP Inc.
  Sector: Technology
  Industry: Business Software & Services
  Price: 7.81
  Analyst Recom: 3
- Ticker: GVP
  Company: GSE Systems Inc.
  Sector: Technology
  Industry: Business Software & Services
  Price: 1.64
  Analyst Recom: 3
- Ticker: CA
  Company: CA Technologies
  Sector: Technology
  Industry: Business Software & Services
  Price: 32.63
  Analyst Recom: 2.9
[sean@Jotunheim] ~/…/cs340 >>>
```

Figure 16: An `industryReport` run by `stock-client`

professionals in the field possess; it is far more practical, in this case and many like it, to provide tools that those who have this knowledge can employ to gain the insights they seek than it is to try to foresee every need without that background.

Figure 16 illustrates a call to the `industryReport` API using `stock-client`. As with `stockReport`, additional data processing occurs on the client side, using `jq`.