

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 System¹ 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>
```

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

²Note 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 System³ 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.

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.

```

[sean@Jotunheim] ~/../datasets >>> mongoimport \
--host=localhost --port=27017 \
--username=administrator --password="" --authenticationDatabase=admin \
--db=market --collection=stocks \
../datasets/stocks.json
Enter password:

2020-06-25T12:56:57.509-0400    connected to: mongodb://localhost:27017/
2020-06-25T12:56:58.738-0400    6756 document(s) imported successfully. 0 document(s) failed to
import.
[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&gssapiServiceName=mongodb
Implicit session: session { "id" : UUID("4cc3d402-f55e-48b4-874c-5b8d7f422054") }
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.

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 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 `config.json` file in the application's root directory to enable the application backend 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

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

```

> db.stocks.createIndex({'Ticker': 1})
{
  "createdCollectionAutomatically": false,
  "numIndexesBefore": 1,
  "numIndexesAfter": 2,
  "ok": 1
}
>

```

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

a given application will know best what indices will offer the greatest performance benefit for their 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:

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

This command, illustrated in Figure 2, will create the simple index on the **Ticker** field.

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.

```

> 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.

Command Line Tools and Library Functions

The Securities Tracking, Obs—okay, you know what, just call it STOCKS⁴—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.

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>
```

⁴I'll see myself out

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.

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 document⁵, 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—such as Mac OS or Windows Subsystem for Linux—but the author has only tested it on Ubuntu 20.04.

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

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


```

[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&gssapiServiceName=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.

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.

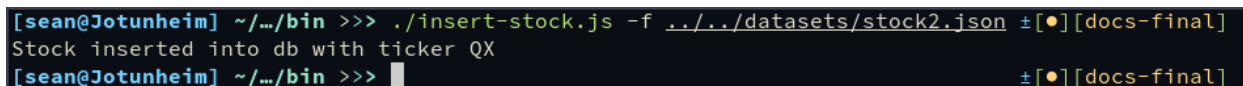
Creating New Stock Records with `insert-stock`

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 record⁶ being inserted into the database, while Figure 6 confirms that the record has been inserted successfully.



```
[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`.

⁶Which 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...

```

_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.

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 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.

```

[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’.

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 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

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 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

```

[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.

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.

```

[sean@Jotunheim] ~/../bin >>> ./sma-spread.js -l 0.02 -h 0.04 ±[●●][docs-final]
Found 886 matching records
[sean@Jotunheim] ~/../bin >>> ±[●●][docs-final]

```

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

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