IOTA Visualizer
— Final Report —

Ao Shen, Yuxiang Wu, Ruikun Cao, Jiangbo Yu, Xiao Luo, Dongxiao Huang
{as5017, yw1117, rc2917, jy3016, xl1716, dh4317}@ic.ac.uk

Supervisor: Prof. William J Knottenbelt, Dominik L Harz
Course: CO530, Imperial College London

1st August, 2018

Contents

1 Introduction

1.1 IOTA

Blockchain promised a decentralized, and open peer-to-peer network that fostered micro transactions without intermediaries or fees. Nonetheless, early adopters of blockchain applications like Bitcoin have been saddled with inhibitive settlement time and transaction costs. IOTA's mission is to build a fee-less settlement layer for the Machine Economy that actually speeds up as it scales. Instead of miners, IOTA asks each participant to verify two other transactions with minimal compute power. This ensures that IOTA transactions will always be free. In fact, IOTA takes this one step further by allowing zero-value transactions, which allows machines to communicate and trade resources with each other, a crucial step towards the Machine Economy much like those in the thriving cloud computing and autonomous driving ecosystems.

A critical difference between IOTAs next generation distributed ledger technology, and Blockchain is that instead of grouping transactions into blocks and stored in sequential chains, IOTA stores them as a stream of individual transactions entangled together, hence the Tangle.

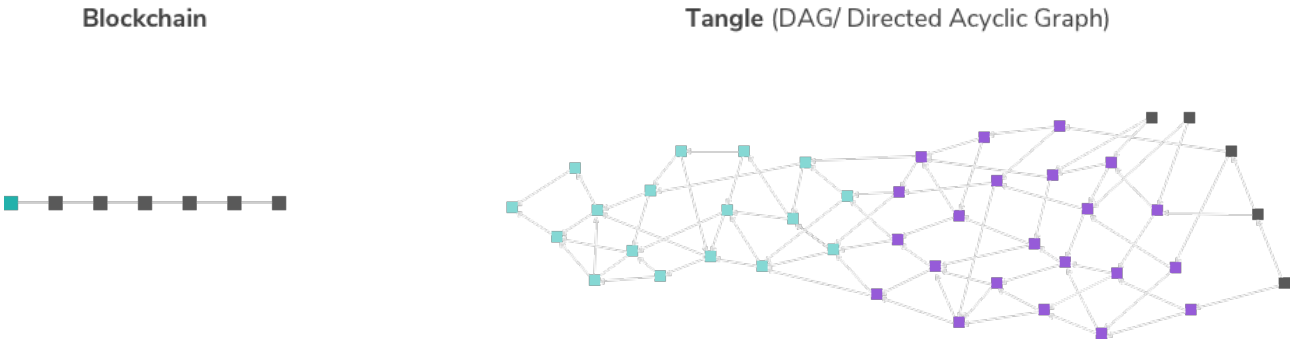


Figure 1: https://www.iota.org/get-started/what-is-iota

1.2 Our Objective

Our objective in this mission is to help developers, researchers and users understand Tangle. A quick Google search will yield splashy SigmaJS-based visualizers that completely dismiss the directed nature of IOTA. [6] To this end, we have developed a research tool (IOTAimperial.com) that will help users understand IOTA's directed network topography, node interactions and provide current statistics and graphs for historical statistics. The tool is fully integrated, allowing users to hover over nodes to glean critical information and dig deeper by clicking which will populate a full page of details about the searched node.

1.3 This Report

As we progress through this report, we will first discuss the specifications that has guided this project. Following that, we will provide a detailed walk-through of the features, implementation and challenges that came from meeting and exceeding those specifications.

2 Specification

While interactive feedback from our project supervisor has allowed us to build many extensions on top of the core product, our core specifications remained largely unchanged throughout the project. The initial specifications were based on another tangle visualizer [6]. While the reference visualizer had its flaws, like how it didn't actually show the directed nature of Tangle's Directed Acyclic Graph (DAG), it did present several critical areas of functionalities we deemed essential. Additionally, non-essential requirements were all implemented. The details of each are summarized below.

2.1 Essential/Status

Essential	Status
Website	Achieved
Transaction information from IOTA API	Achieved
Internal database for caching node data	Achieved
Tangle network map	Achieved

Table 1: Essential requirements

2.2 Non-essential/Status

Non-essential	Status
Node Search	Achieved
Directed Acyclic visualizer	Achieved
Visualizer, 3D animated version	Achieved
Useful, real-time derived data	Achieved
External database and server	Achieved
Global activity heat map	Not Achieved

Table 2: Non-essential requirements

2.3 Software Dependencies

IOTAimperial will be developed using JavaScript and HTML5 for maximum compatibility with popular modern browsers such as Chrome, Internet Explorer, Firefox, and Safari. While our program will not depend on the operating system (Windows, MacOS, Linux), as the differences are expected to be simulated away by the browser, we do recognize the performance and dimensional requirements of mobile browsers and OS (IOS, Android). However, as mobile usage is not our key concern at this point, development will be focused on the average PC. For lower bandwidth and processor capacity scenarios, we will consider reducing the polling rates of our visualizer in exchange for reduced visual fidelity.

2.3.1 Languages

- JavaScript for queuing of the API and advanced styling
- HTML5/CSS for modern web browser compatibility
- NodeJS for back-end integration of the web server to the browser

2.3.2 Libraries and Frameworks

- Vis.js library to visualize both version of the Tangle.
- JQuery framework to support the front-end JavaScript features
- Chart.js to visualize charts for historical statistics
- Express framework to support server-side NodeJS features

2.3.3 External Services

- IOTA API [2] which enables the database to fetch data from IOTA nodes.
- MongoDB [3] to store aggregate statistics in JSON schema
- Neo4J [11] to store IOTA node in a graph database
- Microsoft Azure [7] to deploy and test test our back-end server.

2.4 Hardware Dependencies

IOTAimperial will be browser-based and requires no emulation. The processing requirements are also expected to be within the bounds of most web applications. Therefore, there will not be any local hardware requirements. However, we do depend on an external database server that has reasonable uptime. This will greatly enhance the performance of our platform as a server, instead of our local machines will deal with most of the redundant queuing. In our case, server access has been granted by Imperial's Department of Computing through Microsoft Azure.

2.5 Stakeholders

2.5.1 Users

The likely users and target audience of this product will be those who wish to learn more about the IOTA tangle, aggregate statistics around IOTA and its directed acyclic graph technology.

2.5.2 Core Developers

The developers and maintainers of the project are the 6 members of our group.

2.5.3 Developer Community

We interact frequently with IOTA's developers and received instant feedback and tips on what resources are available. They also expressed their appreciation for our contribution to the platform.

2.5.4 Support

Our supervisors provided resources such as Azure access, specification guidance, implementation assistance, and communication facilitation.

2.5.5 Assessors

Whether we meet our deliverables in an adequate standard will be determined by independent judges from the department of computing and industry.

3 Design Choices

Many design choices such as the style of the visualizer and the choice of library were reached by group consensus, while others were done out of necessity. A good example of design choices borne out of necessity is the modularization of server and database components. Had we not structured our back-end this way, we could not have gotten around the slow IOTA API problem. This and other details of each design choice will be detailed in this section.

3.1 Directed vs Node Graphs

We began our initial graph design using an "explosive" or node-edge demonstration of the network topography. However, while this did "look" impressive, both the direction and acyclic nature of the Tangle would have been overlooked. This was deemed non-acceptable both by our supervisor and the team. Thus, a directed version with arrows and layers was introduced. The legacy version based on nodes and edges is still supported through a toggle.

3.2 Vis.js vs SigmaJS

Initially, we implemented the SigmaJS [9] graph drawing library as was discussed in the first report. [4] However, after learning more about the IOTA data structure, we thought Vis.js [1] would be a better fit.

1. Edges and Vertices: Although SigmaJS is great at drawing charts and diagrams, Vis.js was specifically designed to draw node graphs. This is important, because IOTA is literally an acyclic node graph. Vis allows us to cull verticies and edges from sets directly to clearly convey concepts critical to the IOTA network. Extensions of this library also allow the user to dynamically interact with each node (or block), which might be a useful feature for potential extensions.

2. Dynamic Redraws: Since the aim of each IOTA transaction is to update its neighboring nodes, the status of each node is changing constantly. Additionally, there are new transaction broadcasts onto the network every second. We need to find a way to dynamically reflect these changes. SigmaJS was unable to do this. Vis.js allowed us to dynamically add, delete and change the information of each individual nodes.

3.3 Runtime Environment - NodeJS vs PHP

We researched and experimented with both, but progressed with Node for the following reasons.

1. Front-back Integration: Unlike PHP, Node allows the same JavaScript files to be used for both front-end and back-end development. An integrated runtime environment is critical for team cohesion and integrated testing. Using different frameworks for front and back-end development will introduce unnecessary code reconciliation overhead and bugs.

2. Flexibility and Testing: Unlike PHP, NodeJS comes with fewer hard dependencies and rules that has allowed the community to develop many feature-rich libraries. The NPM package manager comes with a wide variety of frameworks and test packages like Mocha. The integrated nature of Node would also make our system testing more efficient.
   There are drawbacks to Node such as single-threading and nested callbacks that are not present in PHP. Nonetheless, since we are not building high IO-based applications, single-threading will actually let us avoid many concurrency bugs.

3.4 Database Structure - MongoDB vs MySQL

Many of our team members have had previous experience working with SQL and MySQL databases. However we decided on MongoDB because SQL databases have restrictive rules. This required a lot of upfront investment in structuring the database. Since the IOTA data structure and API were still unfamiliar to us, flexibility and optionality were critical for us. This gave us more room to make mistakes, thus spending less time on fixing database problems and more time on other important aspects of our project.

3.5 Database Structure - Neo4j vs MySQL

As the project progressed, we adopted Neo4j as recommended by our supervisor. It can be considered an ideal database for visualization tools, because we can see the link structure of the transactions visually in the Neo4j GUI. Moreover, the feature of the database enables us to find associated transactions rapidly. By comparison, to model 2 edges and 3 nodes in SQL, we would have to build tables with a primary key, and two foreign keys pointing to the same primary key. Under such condition, the query time will grow exponentially as more data is added, hardly a scalable data structure.

3.6 Web App vs Desktop App

Our choice to develop a website instead of a desktop program is 3 fold: Support, Maintenance, and Accessibility.

- Support: The IOTA API fully supports JavaScript, partly supports Python, and offers no support in other languages. This effectively limits our option to a web-based platform if we are to utilize resources from official IOTA channels.
- Maintenance: It is easier to maintain a web-page since we can introduce bug-fixes with no interaction with the client. Simple web applications like ours are also largely operating system independent, making it ideal for rapid deployment.
- Accessibility: Web-Applications can be accessed anywhere by users.

Though it has to be noted that desktop applications are superior in terms of performance and can work off-line. Nonetheless, since IOTA is network-based to begin with, there is no value in off-line operations. The graphical capacity of browsers is also adequate for the scope of our project. Because of this, we believe the pros of a web application far outweighs its cons.

3.7 Database

Near the early stages of the project, an external database did not appear necessary as a web page can directly queue the IOTA API. However, this became increasingly infeasible as we discovered that servers supporting the IOTA API were unstable, and a cache of data was necessary for our product to function smoothly. Further, it is not efficient and scalable. A well-designed server side service can help improve overall performance and reduce redundant bandwidth usage.

3.8 Server

Initially, our database and server functioned as one. Nonetheless, after reaping of both Neo4j, and MongoDB, we had two different databases and we had to separate the structure into several components. Not only is modularization good software engineering practice, it also helped us find bugs, that would have been lost in one big code chunk. An overview of our product structure is as follows.

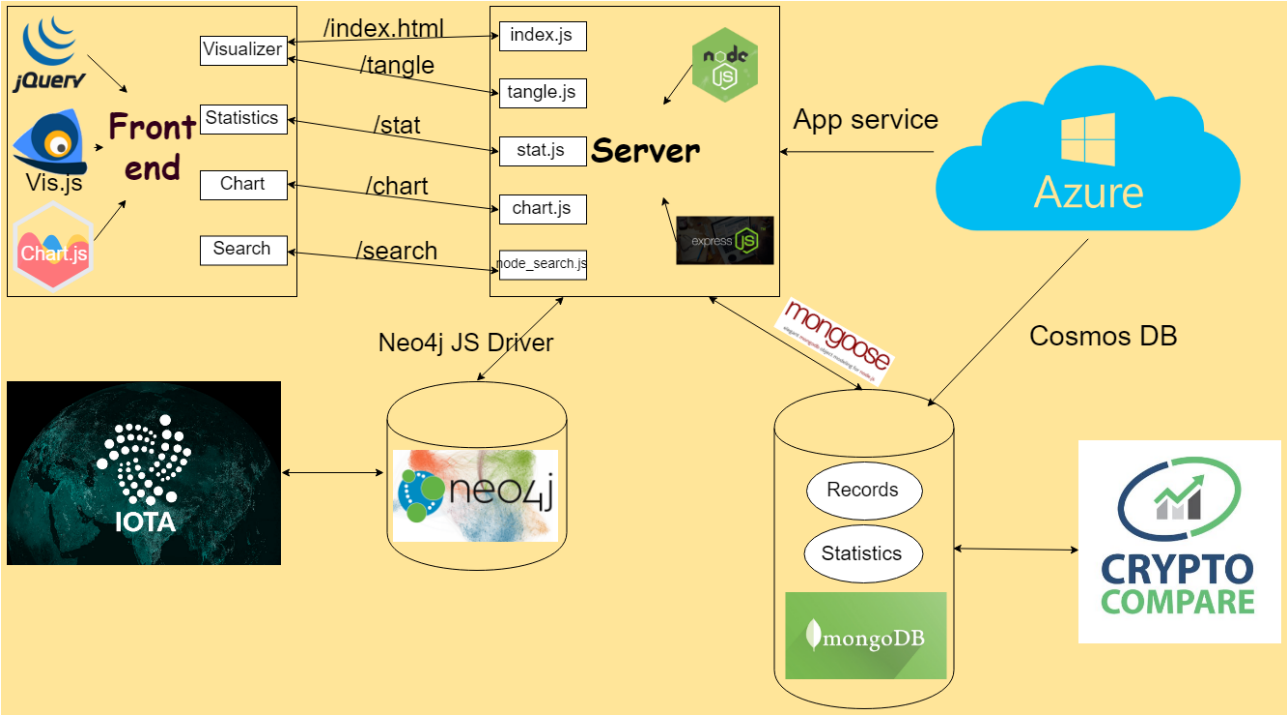


Figure 2: Overview of interactions between each product component

Considering the many tasks here, it is a good idea to divide them into sub-problems, modularize our product, and allow concurrent contributions from developers. In that spirit, we will divide the following methodology section into four parts, reflecting the core features of our product.

4 Essential Feature: Visualizer

4.1 Terminology and Features

The core functionality of this product is the visualizer. There are three types of transaction nodes in IOTA: Confirmed, Unconfirmed and Tip. Note that transactions and nodes refer to the same thing. Transfers usually consist of 4 different transactions, similar to the key exchange process in symmetric cryptography.

1. Confirmed nodes (in green) are transactions that have been confirmed by multiple other nodes. The user can decide how many confirmations are necessary to be considered safe for transacting.

2. Unconfirmed (in red) are transactions that have been referenced by other nodes but is not judged adequately confirmed by the user.

3. Tip nodes (in grey) are transactions that were recently broadcasted onto the IOTA network and thus have not been approved by any other node.

4.1.1 Directed vs Node Representations

We display the Tangle in both Directed and Node-edge graph modes, which can be toggled on the top right. The server will select several new nodes that are connected to the original ones to update graphs in both modes. In Directed mode, the transactions will be put into several horizontal layers, with one layer's transactions confirming another layer's transactions. Arrows will indicate the direction of the confirmation. Clicking each node will highlight all incoming and outgoing arrows (direction of confirmation). Hovering over each node brings up transaction-specific statistics such as value, host address, and hash. In node-edge mode, these relationships are broken down and will display as a ball showing no directions. Both illustrate the same number of nodes but the website defaults to the directed representation, since that most accurately portrays the Tangle.

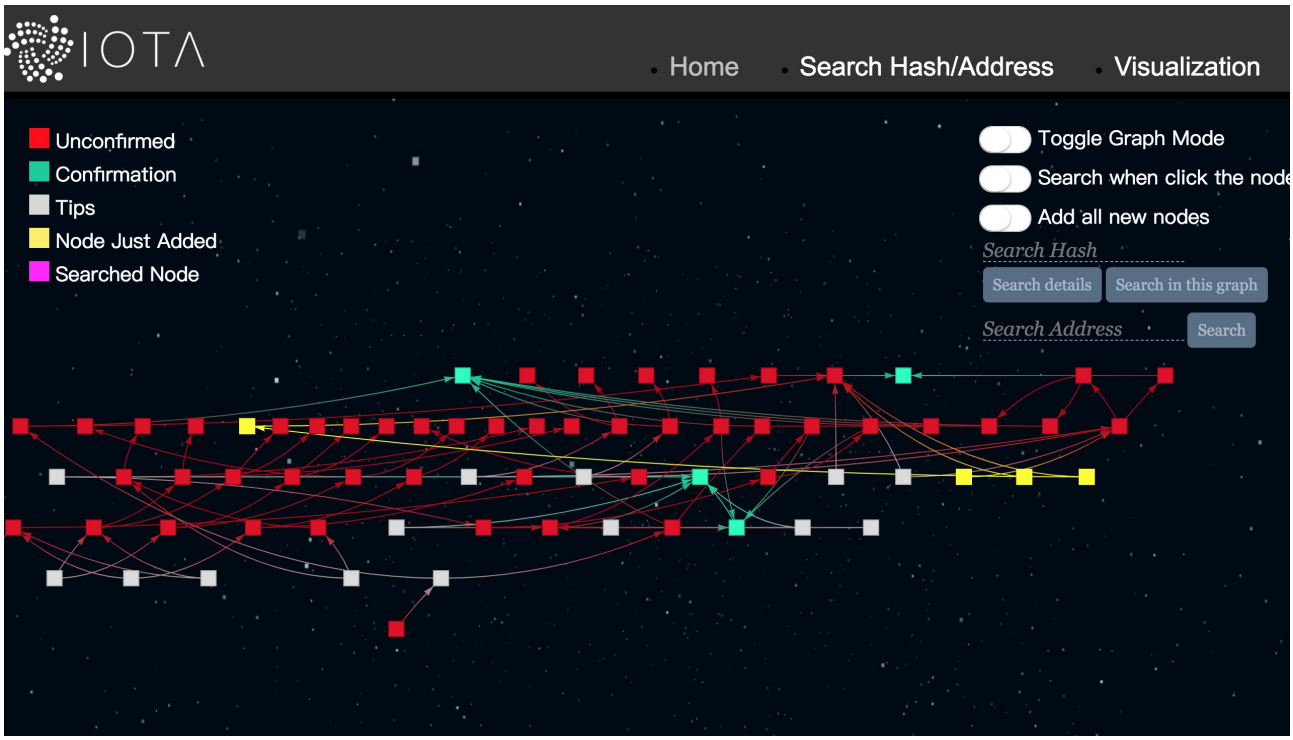


Figure 3: Directed Representation

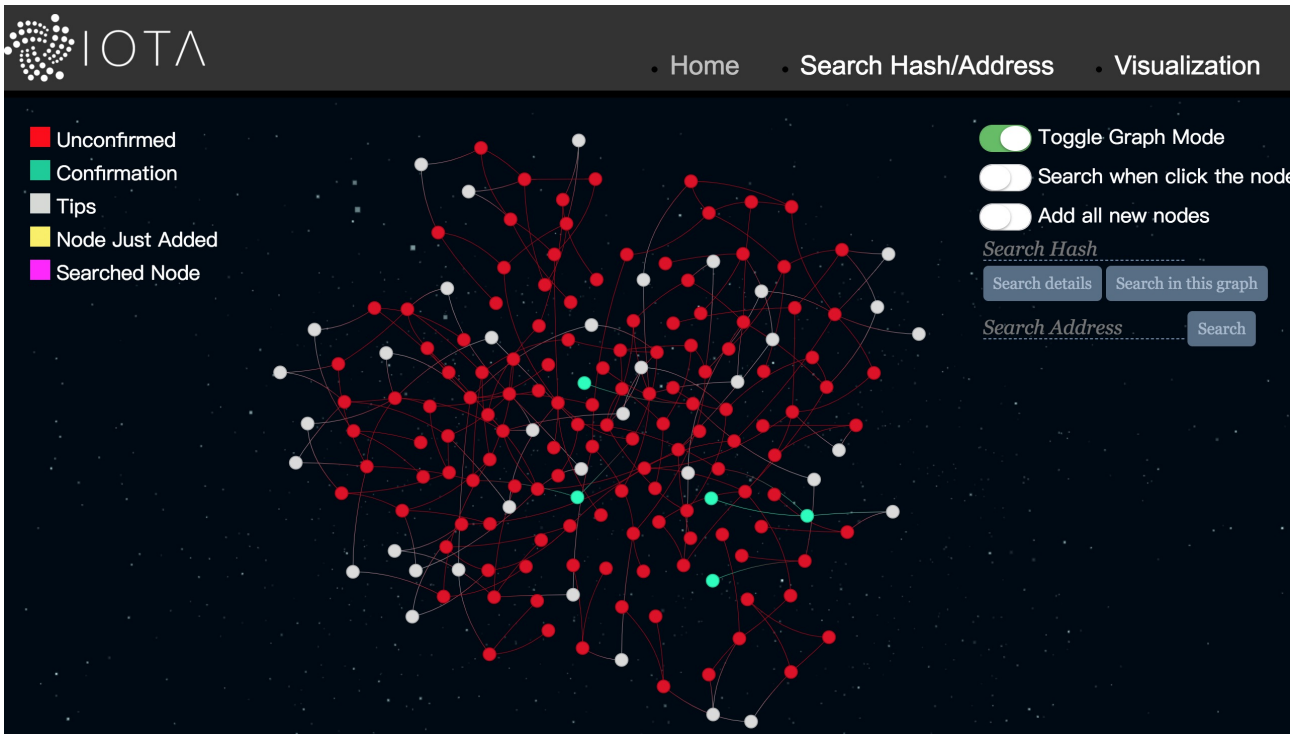


Figure 4: Node-edge Representation

4.1.2 Add All Connected Nodes

Adds all nodes connected to the current ones on display. Note that this has huge performance implications, as new nodes added scales exponentially. Recommended to be turned off after 2 iterations.

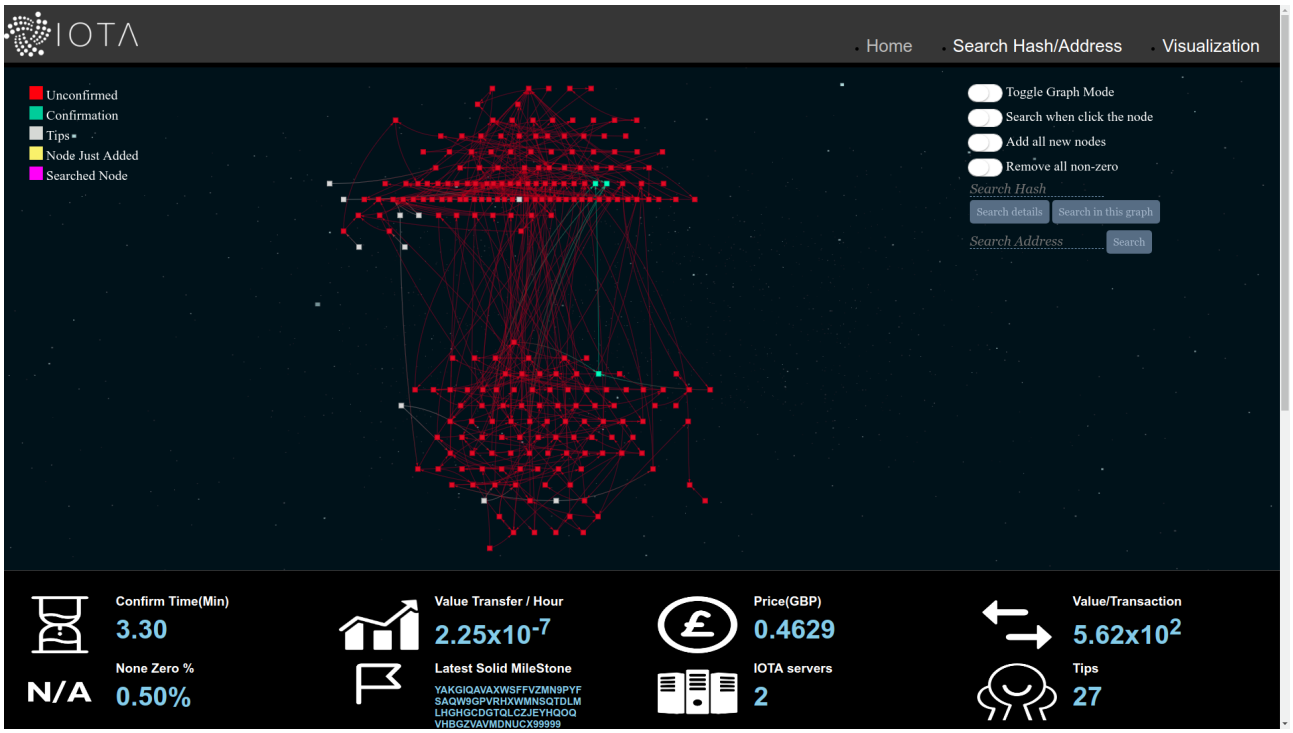


Figure 5: Result after adding all connected nodes after 2 iterations

4.1.3 Remove Zero-Value Nodes

Removes all zero-value nodes from the graph. This effectively leaves only money-transfer transactions. This is interesting, since most value-transfers only have 2 nodes (sender and receiver) with minimal external confirmation.

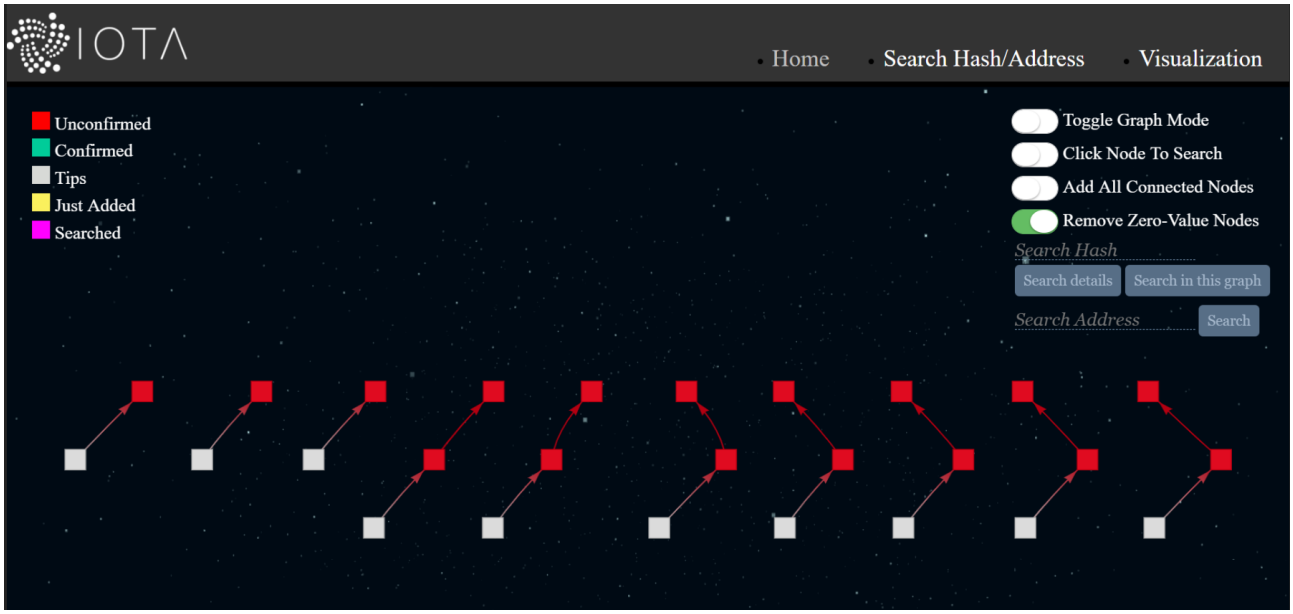


Figure 6: Result after removing zero-value transactions

4.2 Front-end Implementation

The visualizer represents each transaction with a node and the relationships between them as arrows. It is implemented with the Vis.js JavaScript library.

4.2.1 Front-end Implementation, Challenges and Solutions

Initially, in order to dynamically update the graph, the web browser had to sift out nodes that required updating, and query the server for information. Once new data was fed from the server, the graph was then redrawn. Because we had approximately 500 transaction data points, this overloaded the browsers and slowed them to a crawl. The first problem was that the sifting drained a lot of the browser's computing power. The second, was that after each redraw, the canvas was wiped clean which made it hard to track the dynamic growth of nodes.

1. For the first problem, we moved the sifting to the server alleviating some pressure from the browser. The max number of nodes added for each update is capped at 50. The web-page also enables the users to remove this constraint by toggling.

2. To accommodate the second problem, a single node/edge is stored in a dictionary with unique IDs. For each IOTA transaction, it has a unique hash ID. The edge is uniquely identified by its outbound and inbound transaction ID. All other information including the transaction type (tips, confirmed or unconfirmed), value, creation time etc. are also stored in the corresponding dictionary. On start-up, the front-end will first query initial transaction nodes from the server and store them in the data base. Following this, the front-end will query the server in 20 second intervals, but the contents will differ from the initial query, as it includes a parameter which indicates which node need to be updated.

4.2.2 Querying Neighbors

In some cases, the user may want to know about the neighbors of a certain node. To implement this, the user should type in the transaction hash ID needed. The front-end will query the server for

all nodes that have connection to the designated node then visualize them.

4.2.3 Node Details Upon Hovering

When the cursor is hovering over certain nodes in visualizer, brief information about this node will be displayed. To obtain detailed information about certain nodes in the graph, a callback function is defined for the onclick event. The callback function will redirect to the information search page with the ID of the node (hash ID) as a parameter.

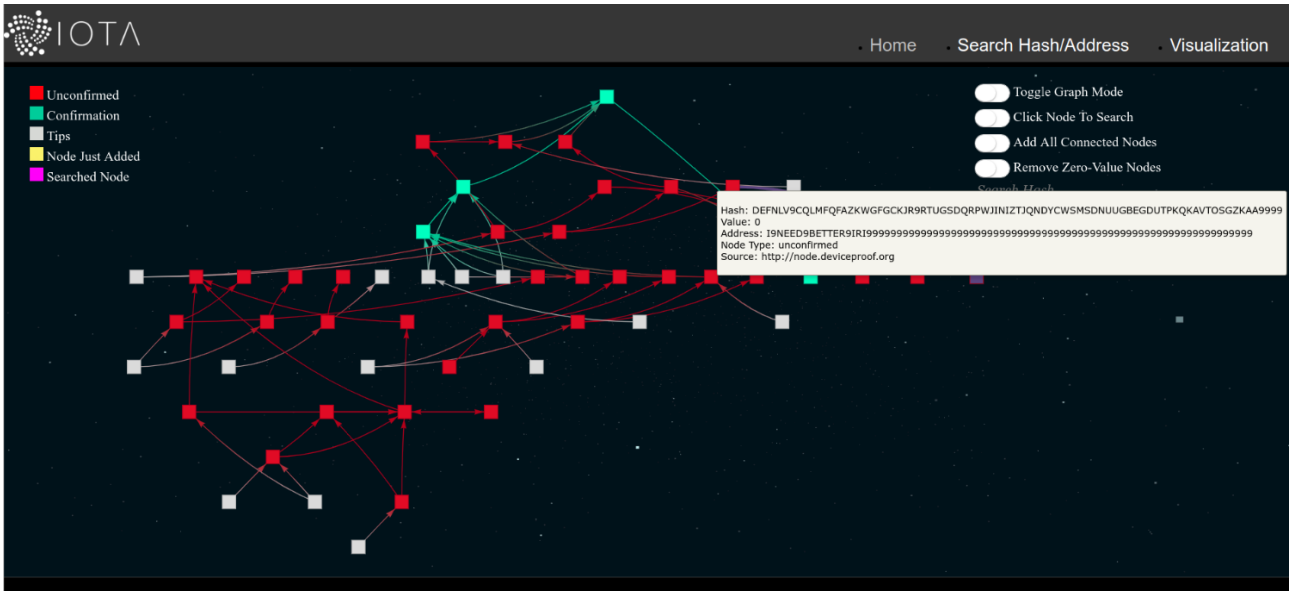


Figure 7: Statistics Visualization.

4.3 Server Implementation

The web server is responsible for receiving client requests, querying the relevant data from the database and then sending it back to the client. In other words, it is an intermediary between the database and client. To customize the experience for each user, we indexed several routers. Each routed user will receive an initial data batch (approximately 100 data points for Directed and 500 for Node graphs) from our database collection. A graph will then be constructed based on these data. We use cookies to save user information, and determine if updates are necessary. Cookies are removed from the client browsers and our database when the session ends.

There are two main functions implemented in the tangle.js file:

```
function initial(req, res, next, amount)

function update(req, res, next, amount)
```

"initial()" is used to send the first batch of transaction data to the client. It takes four parameters: "req" is the request variable, including all information sent from the client. "res" is the response variable. Once the function is finished, it will use "res" to send data to the client. "next" is the Javascript asynchronous handling variable. "amount" means how many tips will be sent. When executed, the function will first call the Neo4j driver. We accomplish this by first finding the latest tips ordered by time, then we find the transactions that are confirmed by these tips (usually up to 4 edges long). "Initial() then receives results from Neo4j database and packages these results into JSON objects. "update()" takes the same input parameters. The only difference is that this time "req" parameter will contains all hashes that the client already has, so the server only sends what the client doesn't have.

4.3.1 Server Implementation Challenges and Solutions

1. In many of our server-side processes, we needed to query data from the official IOTA API. However, due to limited throughput and slow status change times, the IOTA API updates very slowly. This was not anticipated, and forced our graph to appear less dynamic than it was. The limitations of the official API was beyond our control. We are currently looking at ways to increase the dynamism by loading older nodes first and use current ones as "updates." However, this is only a temporary solution that will result in the same problem eventually as current nodes are used up. In the meanwhile, we are also exploring other solutions to ameliorate the problem like folding our datasets so we can display them in rotation.

2. Because not all the nodes queried from server has a linkage (directly or indirectly) with each other, the Directed graph drawn scatters into several smaller trees. To solve this problem, those nodes whose parent nodes are not in the queried dataset are removed and linked to a common hidden root node. This resolves the problem of displaying too many unconnected clusters.

4.4 Database Implementation - Neo4j

Neo4j Database Driver We built a modularized database separate from the server to acquire data, store transaction information, and provide sampled data to the user. It is implemented using the Neo4j graph database with a NodeJS driver. The database updates itself on a regular basis to add transactions, establish relationships between transactions, modify the state of transactions, and delete the outdated transactions. The process above is considered independent of the rest of the project. The task can be divided into the following four parts:

1. Database Initialization: Truncate all data in the database, set hash as primary key, and create index based on the creation timestamp.

2. Transaction addition: Acquire the hashes of transactions and get transaction objects via IOTA API, and insert those objects into the Neo4j.

3. Transaction update: Select all tips and unconfirmed transactions from Neo4j database, query their states via IOTA API, and update the changed states. If states change to confirmed, record the confirmation time.

4. Transaction deletion: When total amount of transactions in the intermediate database reaches the pre-determined upper bound, delete the earliest-added transactions until the amount of transactions fall below the cap. (FIFO)

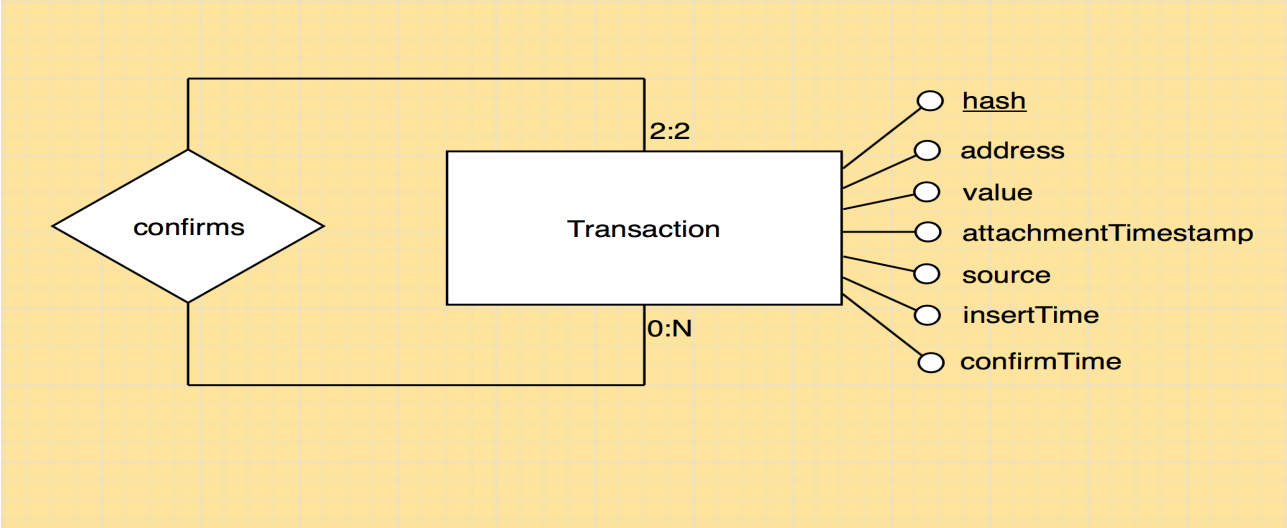


Figure 8: ER Diagram.

4.4.1 Neo4j Infrastructure Layers

Our database follows a three-layer infrastructure model. The outer layer encapsulates the above four tasks into four functions, while the inner layer serves as the Data Access Object, directly transmitting the cypher query to the Neo4j database. The intermediate layer calls the IOTA API for data, and links the outer layer and inner layer by connecting the functions in the inner layer to several function modules and presenting them for calling from outer layer. The three-layer infrastructure follows some frequently-used software framework.

1. Building and manipulating the components in the DAO layer.
2. Combining those operations in Service layer together with external interface to perform some meaningful tasks.
3. Breaking them into several sub-problems in Action layer for call executions.

The advantage of the infrastructure is that each layer performs specific tasks, making the code clear and easy to reuse and modify. It is also a good way to ameliorate nested callback chains in JavaScript.

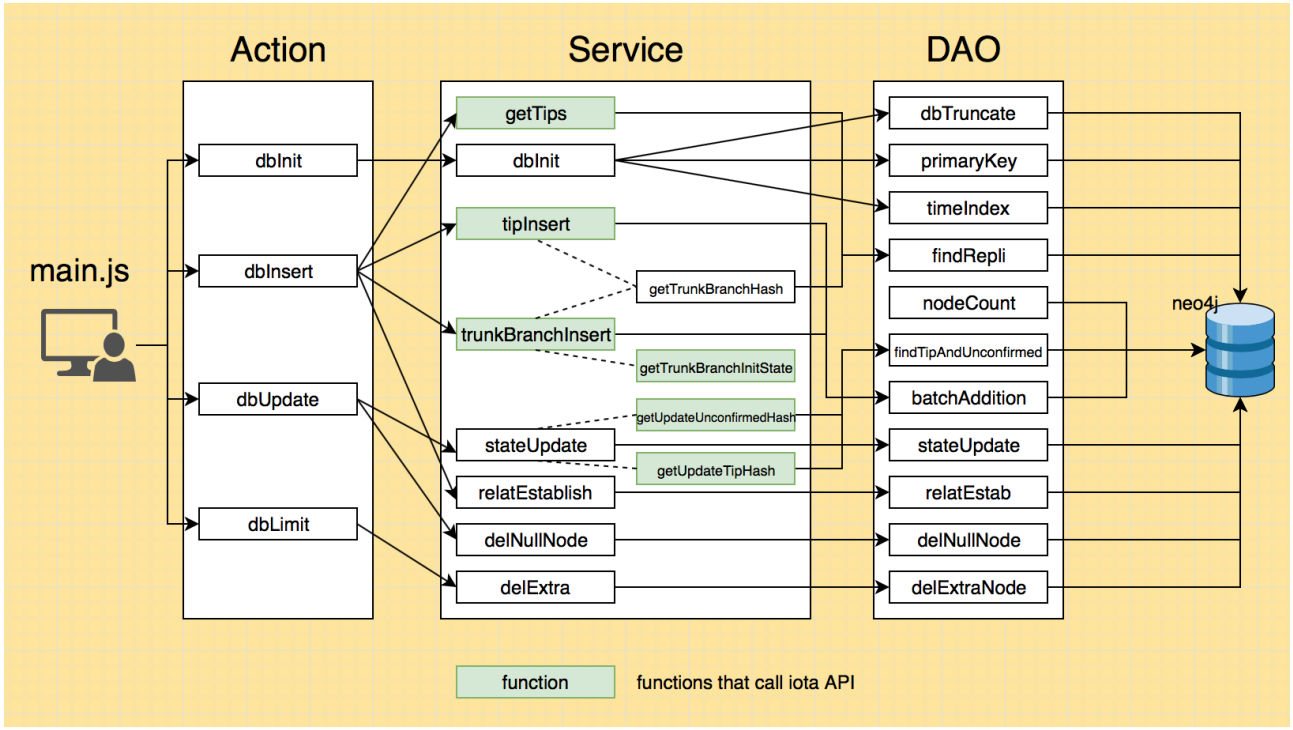


Figure 9: Infrastructure.

4.4.2 Neo4j Implementation Challenges and Solutions

Without dealing with queries cautiously, the database update process can take more than a minute. Speedups can be achieved by performing batch tasks and avoiding hard parses, more specifically, the UNWIND clause in cypher and parameter passing. In addition, the primary key on the hash and the index timestamp helps reduce the query time by avoiding looping through all nodes in the database. This is similar to how a hash table works. Another problem is that we need to consider the scalability and efficiency of queries to the Neo4j database, since we need to query over 100 transactions out of more than 200000 transactions and track their status changes. Initially, it took about 3 seconds to finish the initial query. For example, when we query data, we need to sort the data by their insertion time to get newest active transactions. In this case, we index the attach time in Neo4j database. Although this operation will slow down the process of adding the data into database, it would make it much faster to get data out of database order by time.
An ideal intermediate database should synchronize the data from IOTA database. However, considering our limited resources and the lack of IOTA database permissions, it is almost impossible for us to implement database synchronization. Therefore, we acquire the data from IOTA API instead.

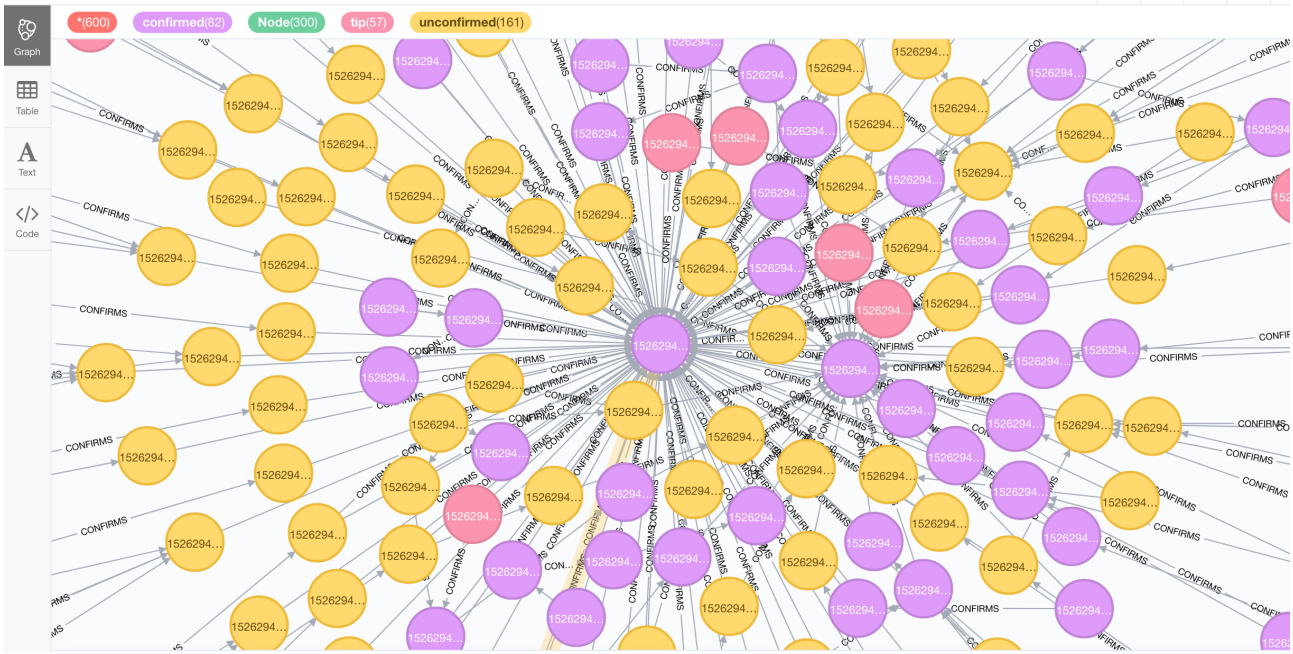
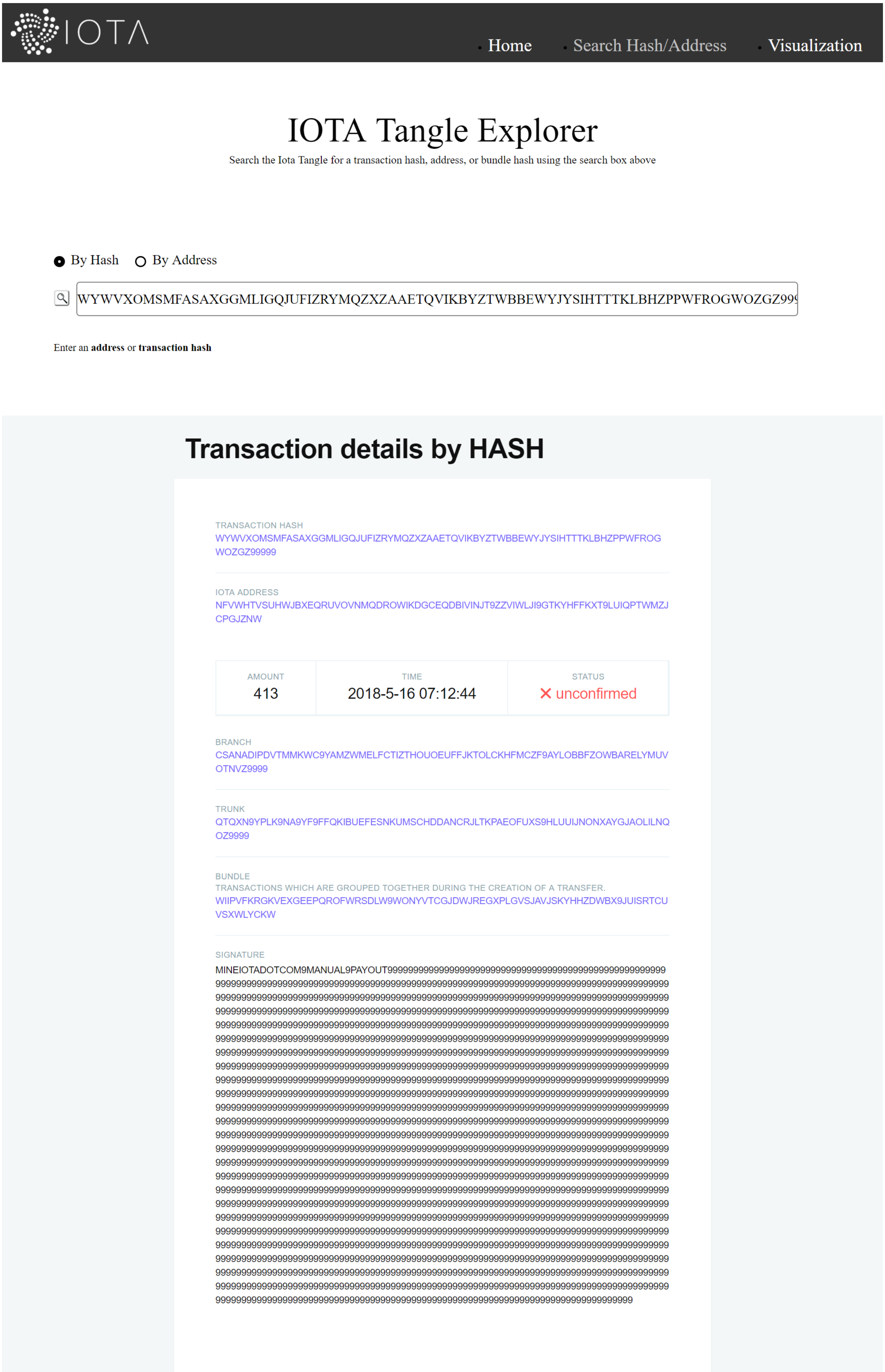


Figure 10: Neo4j Database.

5 Non-essential Feature: Search

For detailed information we also provide several search features, namely "Click Node To Search", "Search Hash" and "Search Address". These allow the user to search by hash, or wallet address, or from a node they see on the screen directly. The result will return the relevant hash, address, amount, time, status, branch, trunk and bundle information. The user can search for the information of certain transaction by Address or Hash ID. These two modes are differentiated by sending server separate URL addresses.
Search by Hash Example: http://iotaimperial/search?hash=XXX
Search by Address Example: http://iotaimperial/search?address=YYY

- Hash: Unique ID for each transaction
- Address: Unique ID for the sender or receiver (aka "wallet")
- Amount: Number of IOTAs, not value
- Time: Timestamp of transaction/node creation
- Status: Confirmed, Unconfirmed, Tip
- Bundle: Collection of 4 transactions that will form one transfer, required to sign transactions
- Branch: 1 of 2 transactions this node has to confirm (random)
- Trunk: 1 of 2 transactions this node has to confirm (recently added to the network)


IOTA
Home
Search Hash/Address
Visualization
IOTA Tangle Explorer
Search the Iota Tangle for a transaction hash, address, or bundle hash using the search box above
By Hash
By Address
WYWVXOMSMFASAXGGMLIGQJUFIZRYMQZXZAAETQVIKBYZTWBBEWYJYSIHTTTKLBHZPPWFROGWOZGZ99
Enter an address or transaction hash
Transaction details by HASH
TRANSACTION HASH
WYWVXOMSMFASAXGGMLIGQJUFIZRYMQZXZAAETQVIKBYZTWBBEWYJYSIHTTTKLBHZPPWFROGWOZGZ99999
IOTA ADDRESS
NFVWHTVSUHWJBXEQRUVOVNMQDROWIKDGCEQDBIVINJT9ZZVIWLJI9GTKYHFFKXT9LUIQPTWMZJCPGJZNW
AMOUNT
413
TIME
2018-5-16 07:12:44
STATUS
unconfirmed
BRANCH
CSANADIPDVTMMKWC9YAMZWMELFCTIZTHOUOEUFFJKTOLCKHFMCZF9AYLOBBFZOWBARELYMUVOTNVZ9999
TRUNK
QTQXN9YPLK9NA9YF9FFQKIBUEFESNKUMSCHDDANCRJLTKPAEOFUXS9HLUUIJNONXAYGJAOLILNQOZ9999
BUNDLE
TRANSACTIONS WHICH ARE GROUPED TOGETHER DURING THE CREATION OF A TRANSFER.
WIIPVFKRGKVEXGEEPQROFWRSDLW9WONYVTCGJDWJREGXPLGVSJAVJSKYHHZDWBX9JUISRTCUVSXWLYCKW
SIGNATURE
MINEIOTADOTCOM9MANUAL9PAYOUT

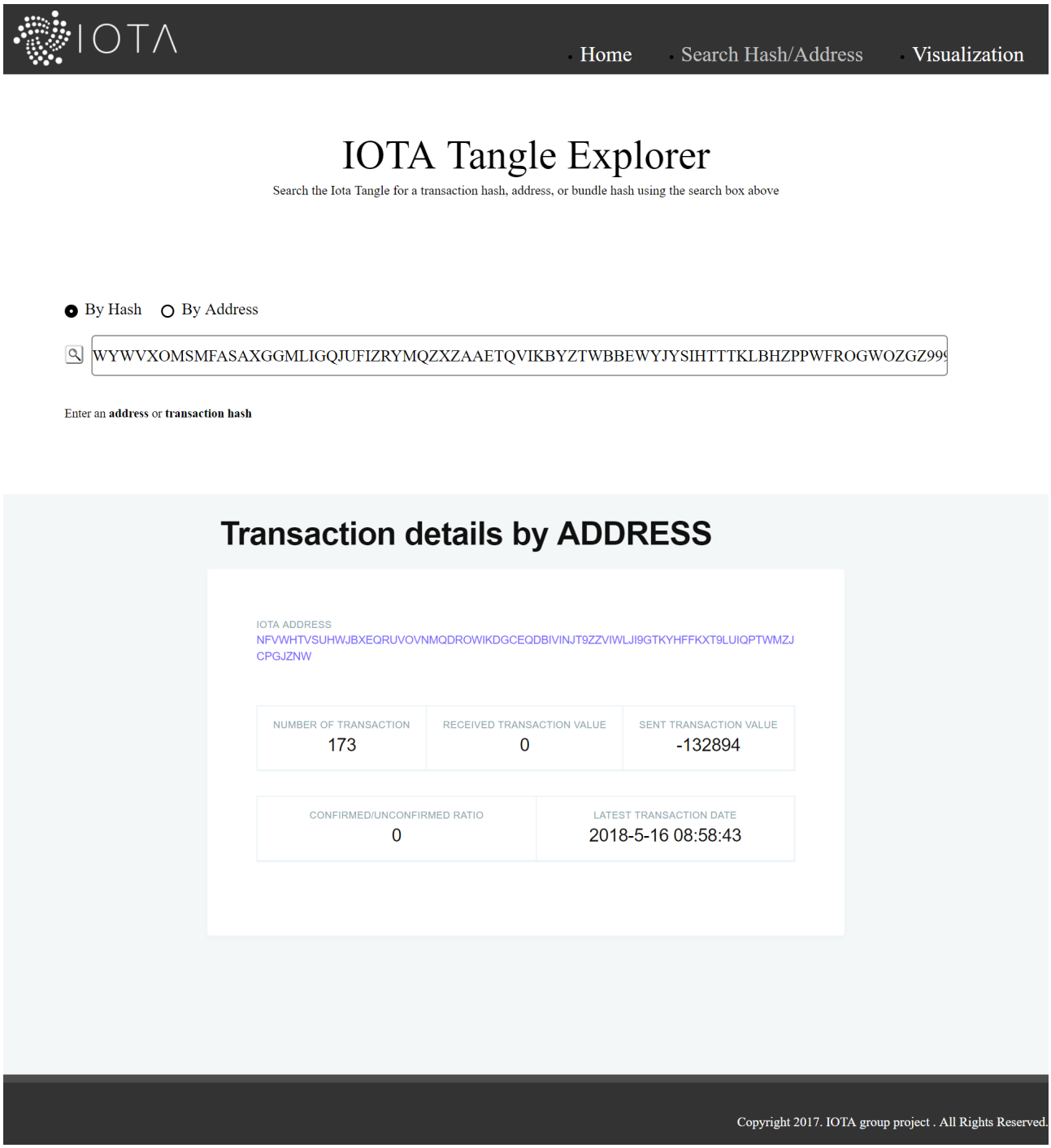


Figure 12: Result after search for address

5.1 Server Implementation

The server will receive a request in a specific end-point router (/node-search) containing the query string which is the hash or address required to search. For hash searching, we will call IOTA API.

```
iota.api.getTransactionObject([hashes], callback)
```

This function receives hash as its input to return details of this transaction in callback function, including address, amount, time, status, bundle, trunk, branch and signature. The server will integrate all these information into one JavaScript object and send it back to front-end via JSON. For address searching, we will show the number of transactions related to this address, total transaction value received, and total transaction value sent.

```
iota.api.findTransactionObject({address:[]}, callback)
```

This function takes address as its input and return all transactions related to this address. We can then derive transactions sent from this wallet address. Summing the values from these transactions and dividing it by the number of confirmed transactions will compute the Confirmation Ratio. Again, we send these information back to front-end as a JSON object.

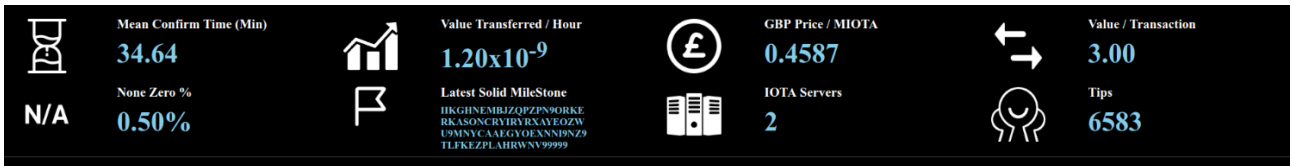
5.2 Server Implementation Challenges and Solutions

When the server needs to find a specific hash, it just searches this transaction from database. But when we test our searching service, we came across a problem where the transaction exists in the IOTA network but does not in our database. Since we cannot store all historical IOTA nodes in our database, we changed our design so that the server will directly query from IOTA server. However, the speed is compromised since IOTA servers will only allocate limited resource to our call. If a hash we search does not exist, we will send a "notValid" marking attribute back to the web page to inform the front-end to show corresponding message telling users that the transaction does not exist.

6 Essential Feature 2: Current Statistics

The transaction statistic summary table contains information including the confirmation time, price, value per transaction. This data is directly queried from the server and updated each 10 seconds. The latest transaction information table contains 9 latest transaction information. A query is sent to the server every 10 seconds and if there are new transactions, the table is updated by removing the oldest transaction and appending the new coming transaction.

- Mean Time (in minutes): The average time for new transactions to get confirmed by the IOTA community. This reflects the level of activity in the IOTA server
- Value transfered / hour: The number of non-zero IOTAs being sent through the system per hour. Reflects roughly the amount of money going through IOTA.
- Price (in GBP): Current GBP value per IOTA
- None Zero Pct: percentage of non-zero value transactions
- Latest Solid Milestone: Percentage of non-zero value transactions visualized
- IOTA Servers: Number of servers we are sourcing information from. Can be specified freely
- Tips: Number of tips currently stored in our database for updating future nodes



Hash	Node Type	Value	Create Time
EGUJKTN9PGJVIFMPTPFZKGSENRAWXRLUBHLO KZPWNMUOYMXYXLKDSNCYFAPMXONQHEKCL KEOQ9NLA9999	tip	0	16/05/2018, 04:15:01
KNUHWRPAYSRBGUFREOBNGSTX9NVVEZOYVU DRTJXCRWRWLHAVVLITQFAMGCKZBHUCMQKW FLYSR9RP99999	tip	0	16/05/2018, 04:15:02
HHZN9TMTXHPEENCF9UYGUYUHRPMAROHTGQ GEQZNYXBAUENNNFJAPXHTPWAXXSOJWGDXLT TMJFIJX99999	tip	0	16/05/2018, 04:15:03
JHWIZDNDLTRPSTVRNIELSHPEELBHQVBGBFVTL VNGOVAXATLWKESYVFVXXRGKUQDZOJVVDHV SNFSS99999	tip	0	16/05/2018, 04:15:04

Figure 13: Statistics and Information Table.

6.1 Server Implementation

We calculate these statistics every 15 minutes, integrate them into one document with the current timestamp, and then store it in MongoDb collection. To differentiate between the current information displayed on the front page and the aggregate statistics on the visualization page, 'Current' collection is used to store latest statistics while 'History' collection is used for history visualization. We monitor the calculation of 8 statistics and when one of them is updated, we update it in this collection. We will store statistics of 'Current' Collection with time tamp into 'History' Collection in fixed interval.

Now we will introduce how we calculate these statistics. For mean confirmation time, We have the function:

```
function updateMeanCon(callback)
```

In the function body, we pick 30 transactions which are latest to be confirmed by the IOTA network. Since we have each transaction's creation and confirmation time, we can compute how long confirmation takes. Then we average the 30 transactions, which gives the mean confirmation time for our displayed nodes. For transfer value per hour, we will query all transaction inserted into network in the last one hour from database, get the sum of their values and divide the summation by the number of these transactions. In the same way, we can also compute the percentage of non-zero value transaction of current network by searching for all transactions with non-zero transfer value, and divide it by total amount.

`iota.api.getTips(callback)`

We call the IOTA API to get all tips that can be detected by the connected server and store the amount of them into database.

`iota.api.getNodeInfo(callback)`

To get latest solid milestone, we send HTTP end-point request to the CryptoCompare API interface to get current price of MIOTA coins.

7 Non-essential Feature: Historical Statistics

One statistic does not make trend, so our group decide to create a graph to demonstrate the trend of all the statistics recorded by our server so we can glean some additional insight into recent trends. Visitors can get access to the visualization web page by click the Visualization tab in the heading of main page or click the corresponding statistics data and it will redirect to visualization page. Charting is done by Chart.js. Here are some terminologies

- Daily price: Price of IOTA incremented by day
- Minute Price: Price of IOTA incremented by minute
- Mean confirmation time: The average time for new transactions to get confirmed by the IOTA community. This reflects the level of activity in the IOTA server
- Value transfered per second: The number of non-zero IOTAs being sent through the system per second. Reflects roughly the amount of money going through IOTA.
- Value per transaction: Average value per transaction, computed after removing zero-value nodes, or the average will be near zero.

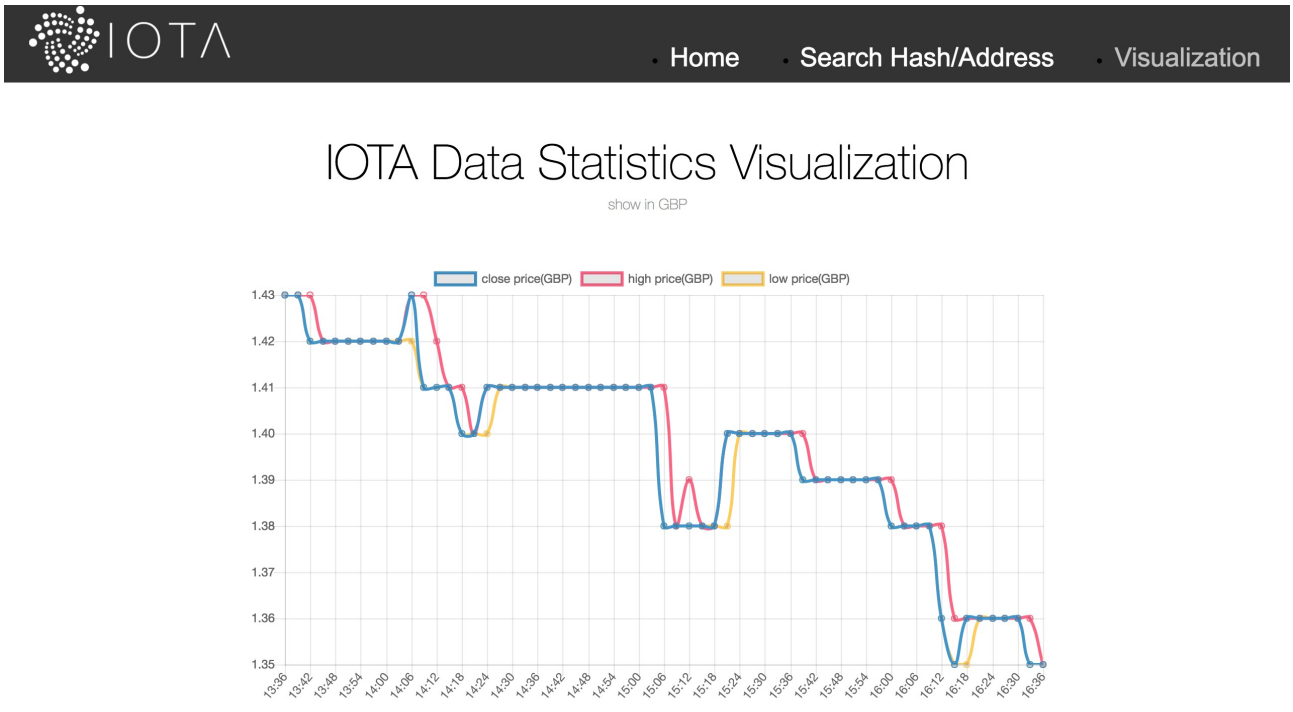


Figure 14: Statistics Visualization.

7.1 Server Implementation

While the server implementation is similar to current statistics, its challenges are not. Since we source our pricing info from CrypoCompare, we are subject to risks from their server and data. In fact, we found that the data before 2018-05-04 from their database was missing. Further, the API provides free updates only every 3 days. To alleviate these risks, we intend to connect to multiple IOTA price servers and eventually move away from CryptoCompare.

7.2 Database Implementation - MongoDB

As is discussed in 6.1, we have used our method to compute all available statistics required in this project, including historical statistics in 'History' collection. With it, we can provide data for front-end to display history-visualization chart. The structure and schema of our MongoDb is shown in table below.

Collection	Schema	Description
Current	TotalTips(str), MeanConTime(str), ValuePerSec(str), NonValuePercent(str), Price(str), ValuePerSec	Stores latest statistics
History	Array	Stores history statistics

Table 3: MongoDB Collection Summary

8 Testing

After the completion of our unit tests, we performed several tests on the integrated system. We used the 'supertest' framework to achieve this. The framework works by sending GET requests to our system and ensuring we get the desired feedback. An example would be the 'addAlltips(tips)'. The function queries the IOTA API, receiving objects in JSON, and is then manipulated and used by many functions before being finally converted into a data structure interpretable by our database schema. If the project expands in the future, we also plan to run compatibility tests with browser drivers beyond just Chrome, Firefox, and Internet Explorer to ensure consistency in user experience. Unit testing is the first stage of our testing process. It will be followed by system testing and a discussion about regression testing.

8.1 Testing Software

1. Sinon.JS [10] was used to generates stubs for unit tests. We used this to full-effect by sending mock XMLHttpRequests to our back-end server so we can retrieve objects from IOTA API to test our written functions.
2. Our JavaScript unit tests were written using Mocha [8].
3. Our front-end was tested using the Selenium Webdriver and ChromeDriver
4. Our coverage report was generated using Istanbul which is integrated with Mocha.
5. Katalon is used to generate User Stories
6. Postman is used to test our server and IOTA API server
7. Neo4j's integrated testing service is used to test the Neo4j database, using Node.js driver
8. Gitlab Continuous Integration and Delivery is used for regression testing.

8.2 Unit Tests

For each helper functions such as extractType and deleteDuplicates, we test its different branches and assert to get corresponding results. We tried to cover every branches of the functions and solved several bugs through unit tests. After that, the coverage performance of the unit test using Mocha and Chai is generated by Istanbul, a Javascript test coverage tool. Note that the queries.js file has lower coverage mostly due to Neo4j's special internal data structures. Regardless, we will still later use Postman to test their outputs, to ensure we are receiving the correct data.

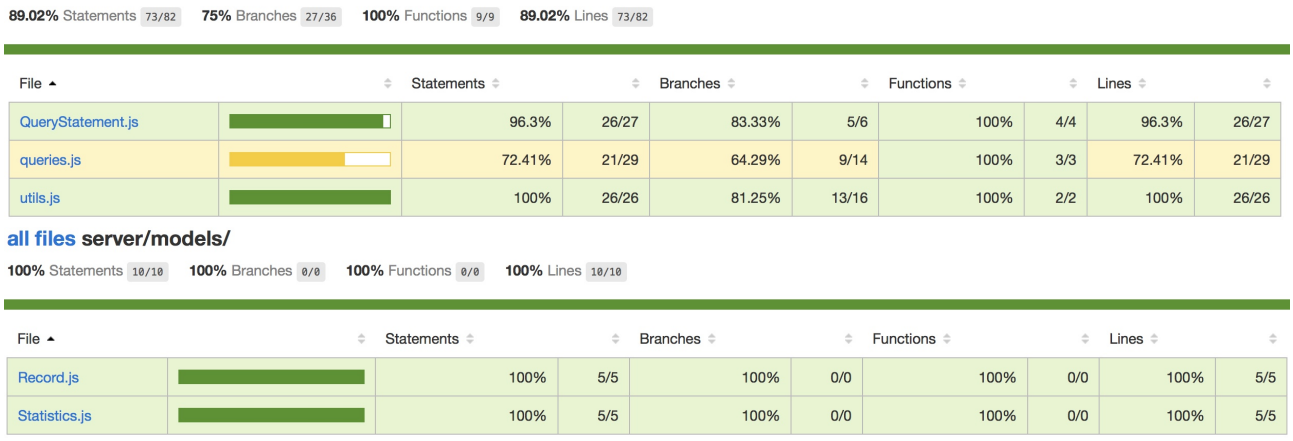
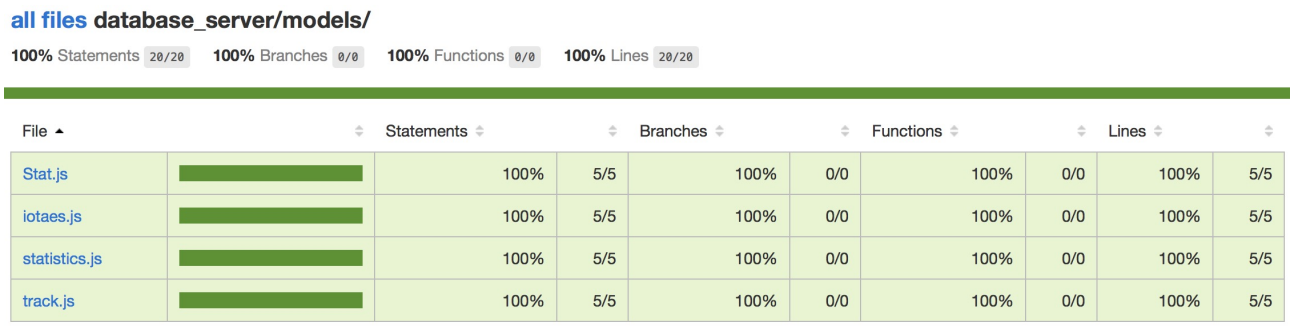


Figure 15: Overall Test Coverage

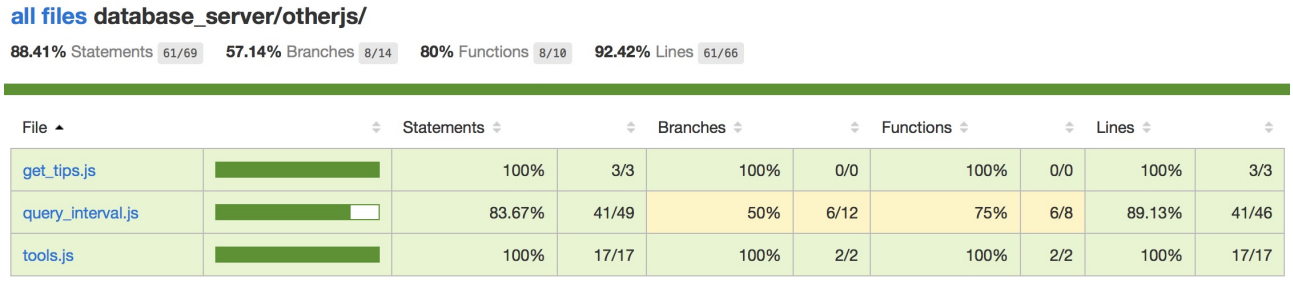


all files database_server/models/

100% Statements 20/20 100% Branches 0/0 100% Functions 0/0 100% Lines 20/20

File	Statements		Branches		Functions		Lines	
Stat.js	100%	5/5	100%	0/0	100%	0/0	100%	5/5
iotaes.js	100%	5/5	100%	0/0	100%	0/0	100%	5/5
statistics.js	100%	5/5	100%	0/0	100%	0/0	100%	5/5
track.js	100%	5/5	100%	0/0	100%	0/0	100%	5/5

Figure 16: Unit Test Coverage Summary for models and computations



all files database_server/otherjs/

88.41% Statements 61/69 57.14% Branches 8/14 80% Functions 8/10 92.42% Lines 61/66

File	Statements		Branches		Functions		Lines	
get_tips.js	100%	3/3	100%	0/0	100%	0/0	100%	3/3
query_interval.js	83.67%	41/49	50%	6/12	75%	6/8	89.13%	41/46
tools.js	100%	17/17	100%	2/2	100%	2/2	100%	17/17

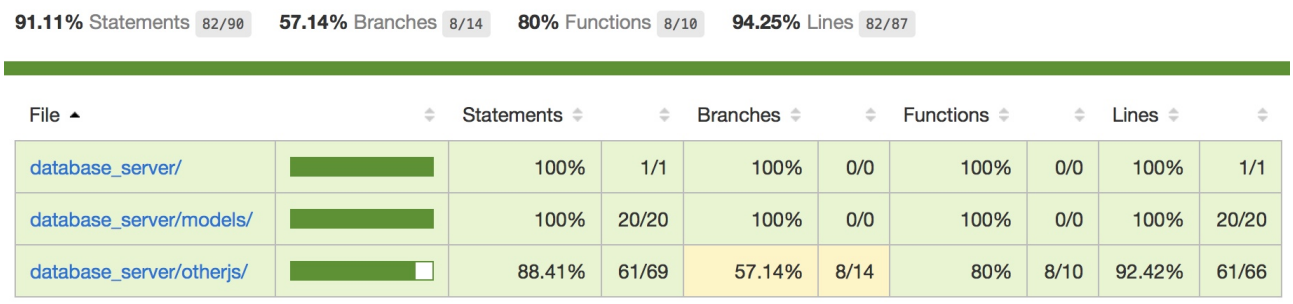
Figure 17: Unit Test Coverage Summary for functions

8.3 Important Test cases

Target	Description	Expected	Actual
Connection	Authentication and timeout	Status Code 200	”
Database Ops	Insert/Delete/Modify/Find	New query reflects changes	”
Schema	Validate entry length, characters, duplicity	filter correct	”
Statistics	Compare with hand-computed results	Results Match	”
Cookie Tracker	Created new collection for one user	Cookies Changed	”
getTips	Partition tested tip hash values	Stub matches output	”
getObjects	Validate object schema	Stub matches output	”
addAlltips	Validate tip schema	Stub matches output	”
queryTip	Correct objects are returned	Stub matches output	”
getLatestInclusion	Correct objects are returned	Stub matches output	Time out

Table 4: Test case summary of functions and operations critical to our project

8.4 Database Coverage



91.11% Statements 82/90 57.14% Branches 8/14 80% Functions 8/10 94.25% Lines 82/87

File	Statements		Branches		Functions		Lines	
database_server/	100%	1/1	100%	0/0	100%	0/0	100%	1/1
database_server/models/	100%	20/20	100%	0/0	100%	0/0	100%	20/20
database_server/otherjs/	88.41%	61/69	57.14%	8/14	80%	8/10	92.42%	61/66

Figure 18: Unit Test Coverage Summary for Databases

The above is the summary of test coverage by branch, functions, lines and statements. It also illustrates how many tests we ran for each script. Since nearly all of our written functions are in models and otherjs, these test results are exhaustive when not considering code from third party libraries. It has to be noted that branch coverage is particularly lacking in otherjs. We have investigated the source and have narrowed it down to IOTA API's getlatestinclusion method. [2] It appears that this function calls getNodeInfo to find out the latest solid milestone hash, and then calls getInclusionStates. This chained API call takes very long and thus times out in our test. (Appendix C) While extending the timeout duration would allow us to pass this test, we believe it is more important to leave it in as the discovery of this bug is a great contributer to our project. Going forward, we have identified this API call as a negative to the performance of our algorithm and will call getNodeInfo and write our own InclusionStates function instead.

8.5 API Testing

Postman is a popular API testing tool we will use to test our server functionalities which all fall under RESTful. In this project, we defined 4 APIs, namely, corsjs, node_searchjs statjs and tanglejs. (Appendix J)

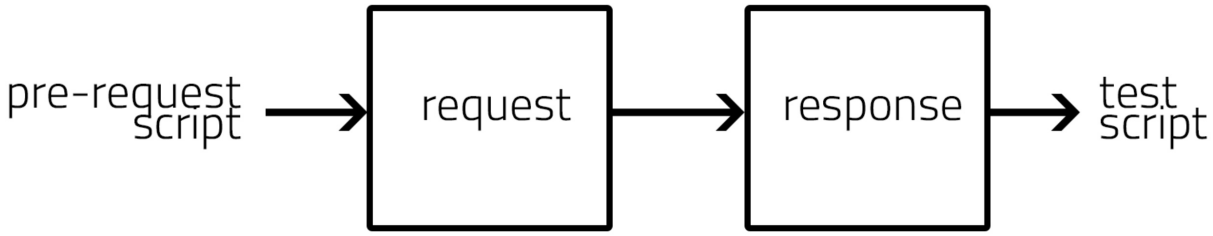


Figure 19: API Tests

8.6 System Testing

8.6.1 Azure Performance Test with 1000 Concurrent Users

Our database server was deployed on Azure. To gauge the performance of our database under stress, we simulated a scenario where 1000 users would be visiting our website simultaneously. (Appendix D) We chose 1000, because usage at this level would exhaust our current Azure credits immediately. Average Response Time of 9.3 seconds, average requests of 280 per second and successful request rates of 97 percent all seemed normal according to performance benchmarks. [7]

8.6.2 Azure Performance Test with 2000 Concurrent Users

However, when we increased the load to 2000, our successful request rate dropped to 88.78 percent, with response times scaling down predictively. (Appendix E) After investigating the successful request issue, it seemed like 502 bad-gateway was the main culprit. (Appendix F) Most of the errors occurred when requesting for JavaScript files with IOTA APIs calls in them. We believe that this is a result of time-outs that occurred when the IOTA API is repeated queried. To tackle this problem, we will likely build a cache to pre-load blocks from the IOTA API, instead of querying it every time a new client navigates to our website.

8.7 Automated UI Testing

We utilized Katalon, a software test suite that records the tester's activities (for example, clicks, hovers, and navigation) to generate a testing script (Appendix G). These tests utilize selenium webdrivers that simulates human actions and HTTP requests. The record allows us to see the interaction

between the front-end UI and our back-end servers, giving us great insight into the integration of our project components. Due to the static nature of these tests, they are generally version and implementation agnostic, meaning that the test results should not change unless new features are added to the UI. A successful test is generally defined as yielding a desired action. These can include:

- A navigation click leading to the correct URL
- A cursor hovering over a block chain, yielding the correct and relevant block information
- Dragged blocks will bounce back to their original position
- Correct statistics being displayed to the user

8.8 Regression Testing

As suspected, testing became a problem with every new iteration of our product. We had to write new tests for newly added features, and oftentimes we did not know whether the test broke, or our code broke. Thus, we adopted GitLab's CI/D integrated environment to run our designated tests after every commit. This allowed us to narrow down which specific commit caused a problem, speeding up our determination of a specific problem, instead of trying to debug an entire system every time. Since the scope of our project is quite small, we believe the cost of re-testing all of our functions during every commit would not be prohibitively high, and would be a great way to alleviate revision risk. (Appendix H)

8.9 Impact and Lessons learned from Testing

- When debugging our code, we were bogged down by certain incorrect date and transaction amounts. Using the Node.js debugger, we discovered that the problems occur only when the node count is high. After analyzing the and narrowing the problem, we concluded that the problem was caused by an overflow in the Neo4j Javascript driver stream. This stream was the bridge between our JavaScript program and the Neo4j database. We overcame this problem by converting the number to string in the database and convert it back to number in Node.js program, since strings cannot overflow. Several lessons were learned in this endeavor. First, we decided to write modularized and non-redundant functions that are maintainable and easy to test. Second, we started writing tests and integrate as we go, so we do not need to discover problems all together. Lastly, we opted to have external developers (or in our case, team members who did not work on the task) to test it to remove test bias.
- While we were processing data from the official IOTA API, the 'getTips' method retrieved seemingly odd hashed blocks like "9999...." Nonetheless, when we used the findTransactionObjects method to retrieve information about that block, it reported a "hash invalid" error. To reconcile this problem, We had to write custom functions to filter out these invalid hashes manually. We subsequently reported this bug to the IOTA foundation. This bug sank a lot of our developer time. We learned to add return values to our functions, instead of writing void functions. Additionally, we noted to our selves to cooperate closely during development so we all know what values are being passed through each function.

9 Team Structure and Software Development Technique

Team members generally pick their own tasks and do not specialize solely in one area as it inhibits learning. Nonetheless, our roles can be divided into front-end, back-end, testing, and team organization. "Everyone" indicates that every team member chipped in a certain task. More details can be seen in the Team Log in Appendix A

Table 5: Front-end

Task	Participants
Web Page	Ruikun, Everyone
Graph Visualization	Yuxiang, Jiangbo
Search	Ruikun
Statistics Visualization	Ruikun, Dongxiao

Table 6: Back-end

Task	Participants
MongoDB	Ruikun
Neo4j	Xiao
Server	Ruikun

Table 7: Testing

Task	Participants
Unit Tests and CI test	Dongxiao
User Story	Ao

Table 8: Organization

Task	Participants
Report Writing	Ao, Ruikun, Everyone
Coordination	Ao

9.1 Extreme Programming

Extreme Programming (XP) is a software development technique with the goal of flexibility in mind. In the traditional Waterfall model, it is expensive to change requirements, especially near the later stages of a project as everything is planned from the start. We can achieve XP Programming by following the mantras below.

- Pair Programming: An experienced programmer is coupled with 2 less experienced ones to share knowledge, spot mistakes and bounce ideas. This has improved our teamwork and knowledge.
- Test driven development: We write tests for new programs as they are committed allowing us to add new features efficiently.
- Collective code ownership: Since we do not delegate fixed responsibilities to anyone, every takes responsibility in each other's work.
- Modularization: Though not a specific XP Programming Technique, we modularize all of our components allowing for parallel programming.
- Small Releases and Continuous Integration: We integrate our components at every major commit, to prevent bugs from accumulating and become undecipherable.

9.2 Version Control and Communication

- We initially used Github as our preferred version control system. However, the integrated testing environment provided by Gitlab prompted us to switch. This is where each commit will be tested. Additionally, we also keep a bug list on the repository.
- We use Slack and emails to communicate with our supervisors and core IOTA developers, to enlist their help, and to facilitate meetings and discussions.
- While we do a significant amount of scheduling and information sharing on the platform, we prefer to debug and test our programs in-person, where communication is more precise.

10 Final Product

Our final product consists of 4 features: Tangle Visualizer, Node Search, Current Statistics, and Historical Statistics. All four components are encapsulated into one website, and separated into different web pages. Visit us at www.iotaimperial.com!

10.1 Key Features Implemented

Each feature and its implementation is explained in detail in previous sections. Here is a summary of key features.

- Directed acyclic representation of the Tangle
- Node-edge representation of the Tangle
- Option to manipulate nodes displayed (Add/Remove/Non-Zero)
- Search by clicking nodes in the graph
- Search by hash
- Search by address
- Current Statistics (Confirmation Time, Milestones, etc)
- Historical Statistics Graph (Transactions/value per second, Price, etc)

10.2 Features Not Implemented

A feature not implemented was the "Global activity heat map" and warrants a brief discussion. This was actually a self-proposed feature, and was initially thought of as feasible. However, as we deepened our understanding of the IOTA DAG network, it became apparent that there was no way to keep track of a meaningful number of nodes across the world to produce a representative global heap map given IOTA API's supported functionalities and our bandwidth allowance. We later noticed that there were on-line discussions of this feature and why it was not yet feasible. From this, we have learned to research features why a seemingly useful feature has never been implemented. Only when we can overcome those known challenges, should we then move forward with it using our team's resources.

10.3 Problems Overcame

Not surprisingly, most of the implementation problems came on the back-end, where the actual work is done. Details of these problems were listed in the implementation challenges section of each feature. A summary of problems we have overcome are as follows.

Problem	Solution
Browser lag from redundant querying	Neo4j database set up to cache data
Slow Neo4j Query	Indexing data points by time stamp
Updates overwriting previous entries	Server sending back index values that will be exempted
Connected graph devolving into separate smaller ones	Sift out non-common parents
Non-valid nodes messes up graph nodes	Built exception handler to find and remove these
CryptoCompare data not reliable	Diversify our API to accept multiple information sources

Table 9: Problems we overcame

10.4 Unresolvable Problems

1. A statistical problem arises from the fact that we only take small number of transactions to reflect some characteristics of the whole network. With a small sample size, the variance or bias will be high. Accordingly, we do observe the numbers of some statistic have great fluctuations. One of the solution of this problem is to increase the amount of samples. However, there is a trade-off between accuracy and resources spent. We simply do not have the resources to use information from every single transaction to compute our statistics.

2. We have yet to solve the problem of long-wait time when searching by address. Addresses that has made many transactions will take disproportionately long to yield search results. This creates a search lag when users interface with our search UI. We have contacted the IOTA core developers. It would appear that other IOTA sites with search function faces similar problems. [5] We believe that reading the core IOTA data stream, divorced from the API is necessary to discover a solution to this problem.

10.5 Evaluation

Our product solves the unique and novel problem of representing the directed acyclic visualization, which most visualizers do not represent. The product itself has met all specifications set by our supervisor. In fact, 2 of the 4 features listed were initially considered non-essential. This does not even include the modularization of our back-end, which has greatly improved the scalability of our product going forward.

Additionally, we also provide several derived statistics for IOTA network, which is innovative and not seen anywhere on the market. We also have developed high-performance and scalable Neo4j database with the capacity of storing more than 200,000 transactions. Moreover, our visualizer could show transactions queried from multiple IOTA servers, making it resilient to server failures (more details can be seen from the testing section). It would be very easy for other projects to build on top of our modules, and for us to build on top of others.

10.5.1 User Stories

While the unit, system and regression testing can attest to how reliable our product is, the extent to which our product satisfies the specification depends entirely on the user experience and if they can find what they need. To this end, we also ran several user story tests based on scenarios generated by humans. Some are from ourselves, while others are from our supervisors and peers. The goal of these tests is to ensure that our outputs are not only functionally correct, but also suits the requirements and specification of our supervisor and users. For example, a user wishing to search and visualize which nodes he is connected to would have the procedure broken down as follows.

1. User searches for own node using hash

2. User receives a visualization of local nodes

3. User clicks on a visualized node to highlight relevant inbound connections

4. Correct aggregate statistics being displayed to the user

These manual actions on our UI are then scripted and tested. While the automated tests are more exhaustive, the user stories allow us to focus on frequently used paths, that are more relevant to the end user. (Appendix K) By breaking a user demand into separate executable tasks, and ask the user for feedback on whether his or her intension is satisfied, we can ensure that our product truly satisfies their specification.

Appendix

A
Task Log

A.1 Ao Shen: Back-end, Tester, Document Editor

01/12/2018 (60min): Designated as Document Editor, will initiate report one and build logbook
01/15/2018 (60min): Report One initiated, logbook built, took notes in group meeting
01/22/2018 (120min): Completed Report One
01/29/2018 (60min): Learned back-end JS and Mongo from Cao, will write test code for UI
02/05/2018 (60min): Learned front-end JS from Bo, completed testing the UI
02/21/2018 (120min): Will complete 50pct of Report Two, write unit tests
02/29/2018 (120min): Completed 50pct of Report Two, wrote 8 unit tests
03/05/2018 (240min): Complete Report Two
03/12/2018 (240min): Getting front-end user stories
03/19/2018 (240min): Testing front-end user stories
03/26/2018 (240min): Researching Database Testing methods
04/02/2018 (240min): Executing Database Testing methods
04/09/2018 (30min): Meeting with Dominik about potential extensions
04/16/2018 (360min): Final Report
05/14/2018 (360min): Final Report

A.2 Yuxiang: Front-end, Report Contributor

01/17/2018 (60min) Complete the layer design with Jiangbo
01/25/2018 (60min) Learned the vis.js library online
02/05/2018 (120min) Drew the first version of sphere graph node
02/19/2018 (120min) Complete the redraw function of the tree graph
02/26/2018 (120min) Complete the dynamic effects in displaying all the indexes
02/28/2018 (120min) Complete the switch between sphere graph and tree graph node
03/02/2018 (120min) Report two and start to optimize the whole front-end next week
03/27/2018 (120min) Enable the dynamically update of the tree and sphere graph with Jiangbo
04/04/2018 (120min) Design the search page
04/18/2018 (180min) Implement the transaction search page with Jiangbo
05/11/2018 (180min) Optimize all other functions in our website
05/13/2018 (240min) Final report
05/13/2018 (300min) Final report
05/14/2018 (180min) Final report

A.3 Ruikun: Front-end, Back-end, Report Contributor

01/17/2018 (60min) Made first version of front-end webpage with AO and Dongxiao.
1/25/2018(120min) Completed first version of our main server
2/02/2018(120min) Deployed the MongodDb in Azure
2/15/2018(60min) Used mongoose driver store IOTA transaction data into MongoDb
2/25/2018(120min) tried to calculate statistics for IOTA
03/02/2018(120min) get more kinds of statistics, like price
03/12/2018(120min) tried to begin to use Neo4j database
03/22/2018(120min) add search function into server
04/01/2018(120min) add history visualization function into server
04/11/2018(120min) begin to query data from Neo4j to server
04/21/2018(120min) improve efficiency of the server
05/01/2018 (120min) try to give last version of server and deploy in Azure

A.4 Jiangbo: Front-end, Report Contributor

01/17/2018 (60min) Complete the layer design with Yuxiang
01/23/2018 (60min) Learned the vis.js library online
02/12/2018 (120min) Drew the first version of tree graph node
02/19/2018 (120min) Complete the redraw function of both the sphere graph node
02/26/2018 (120min) Found the bugs that causes the failure of the toggle switch
03/02/2018 (120min) Report two and start to complete art design of the website next week
03/27/2018 (120min) Enable the dynamic update of the tree/sphere graph with Yuxiang
04/04/2018 (120min) Enable display of transaction information when hovering over nodes
10/04/2018 (120min) Design and implement the interface for search by click node function
18/04/2018 (180min) Implement the transaction search page together with Yuxiang
11/04/2018 (180min) Optimize all the functions in front-end page
12/04/2018-15/04/2018 Write the final report

A.5 Xiao: Database, Report Contributor

12/25/2017 (60min) Learned the fundamentals of IOTA online
01/16/2018 (180min) Completed Requirement Document version1.0
01/22/2018 (60min) Report One
03/05/2018 (10min) Prepare for implementation of an intermediate neo4j database this week
04/02/2018 (360min) Design the data model of intermediate database
04/03/2018 (360min) Design the infrastructure of database driver
04/04/2018 (360min) Implement the database and driver
04/05/2018 (360min) Implement the database and driver
04/06/2018 (360min) Implement the database and driver
04/07/2018 (360min) Unit testing and bug fixing of neo4j database
04/08/2018 (360min) Optimize the data querying and updating efficiency
05/13/2018 (360min) Final Report
05/14/2018 (240min) Final Report

A.6 Dongxiao: Back-end, Tester, Report Contributor

01/13/2018 (60min): Learned web development skills like markup, attributes and url of HTML
01/14/2018 (60min): Learned web development skills, started to develop front-end next week
01/17/2018 (60min): Learned server and looked for suitable database next week
01/23/2018(120min): Learned MongoDB development and improved the performance and fixed bugs
02/10/2018 (120min): Tested the server and browser and tested the other parts next week
02/15/2018 (120min): Tested MongpDB connection and try to create coverage report
02/25/2018 (120min): Report Two and change the started to learn Neo4j Database next week
03/05/2018 (120min): Server end Test
03/13/2018 (120min): Visualization history design
03/20/2018 (360min): Implement visualization history using JQuery and Chart.js
04/02/2018 (120min): Test MongoDB and neo4j database
04/10/2018 (120min): Test helper functions and feedback
04/17/2018 (180min): Server end optimization
04/23/2018 (180min): Test database and server end
05/11/2018 (180min): Testing and CI tesing
05/12/2018 (180min): Final report
05/15/2018 (180min): Final report

B Schedule

Deadline	Description	Progress
Jan 15	Gathering Requirements	Completed
Jan 22	Website (See Appendix Figure 2)	Completed
Jan 29	Report One	Completed
Feb 5	Queuing the API using JavaScript/JQuery	Completed
Feb 12	Choosing and Implementing a Visualization Library	Completed
Feb 19	Implementing the back-end database	Completed
Feb 26	Prototype Testing	Completed
Mar 5	Report Two	Completed
Mar 26	IOTA Block Extraction	Completed
April 9	Statistics Computation	Completed
May 16	Final Report	Completed
May 21	Presentation	Completed

Table 10: Revised timetable

C Unit Test Error

```
1×  function create_father(Collection, tips, results, callback) {
3×    let hashes = [];
3×    for (let i = 0; i < tips.length; i++) {
7×      hashes.push(tips[i]["hash"]);
      }
3×    iota.api.getLatestInclusion(hashes, function(error, values) {
3×      I if (error) console.log(error);
3×      E if (values) {
3×        for (let i = 0; i < values.length; i++) {
6×          if (values[i] === true) {
1×            results = tools.deleteDuplicates(results);
1×            for (let j = 0; j < results.length; j++) {}
1×            Collection.remove({})
                .then(() => {
                  setTimeout(function() {
                    Collection.create(results, function(error, docs) {
                      if (error) console.log(error);
                      else {
                        console.log("update finish");
                        callback();
                      }
                    });
                  }, 1000);

                });

1×            return;
            }
          }
```

Figure 20: Unit Test Error

D

Azure Performance Test 1000

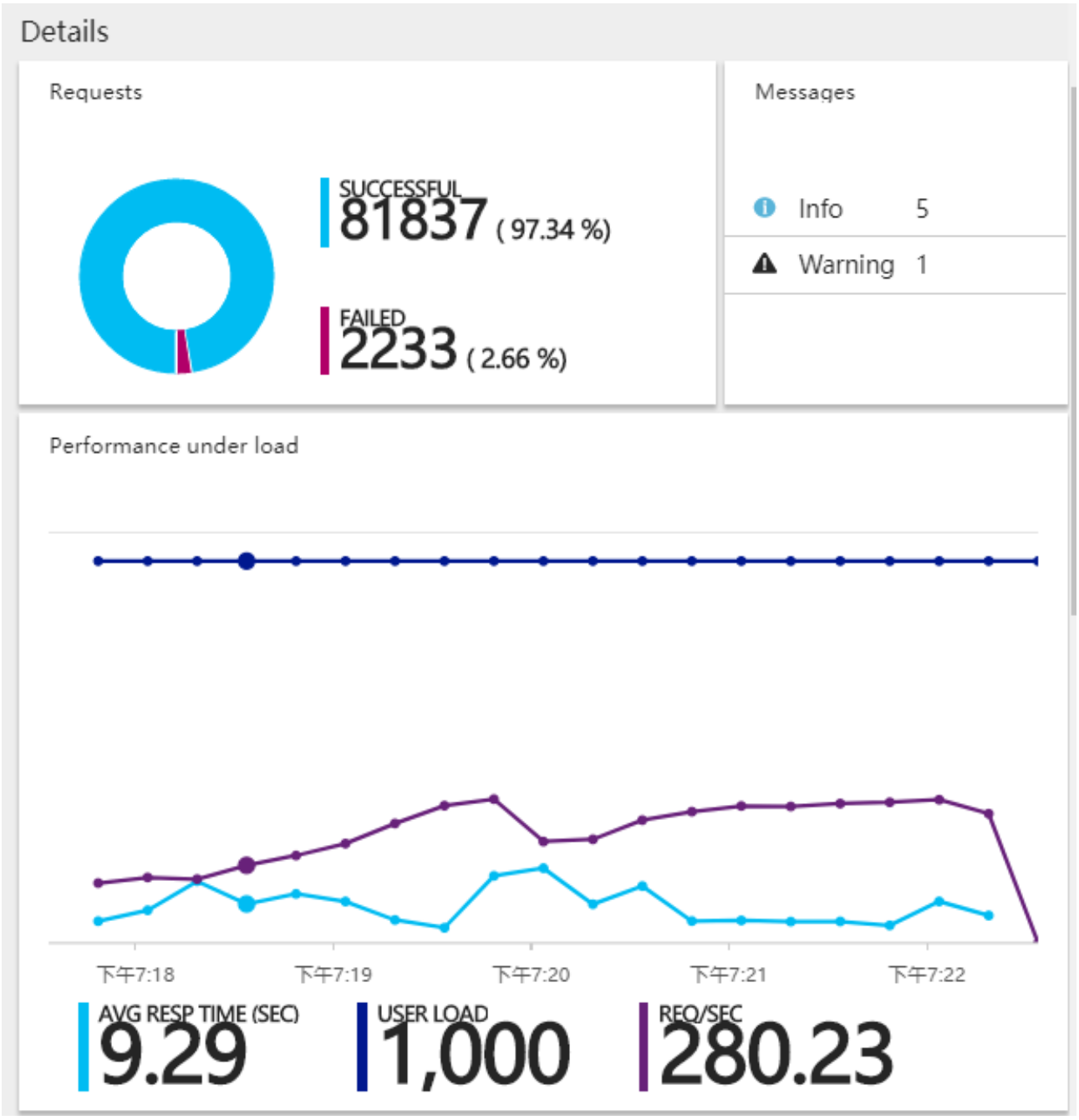


Figure 21: Azure Performance Test Results with 1000 users

E Azure Performance Test 2000

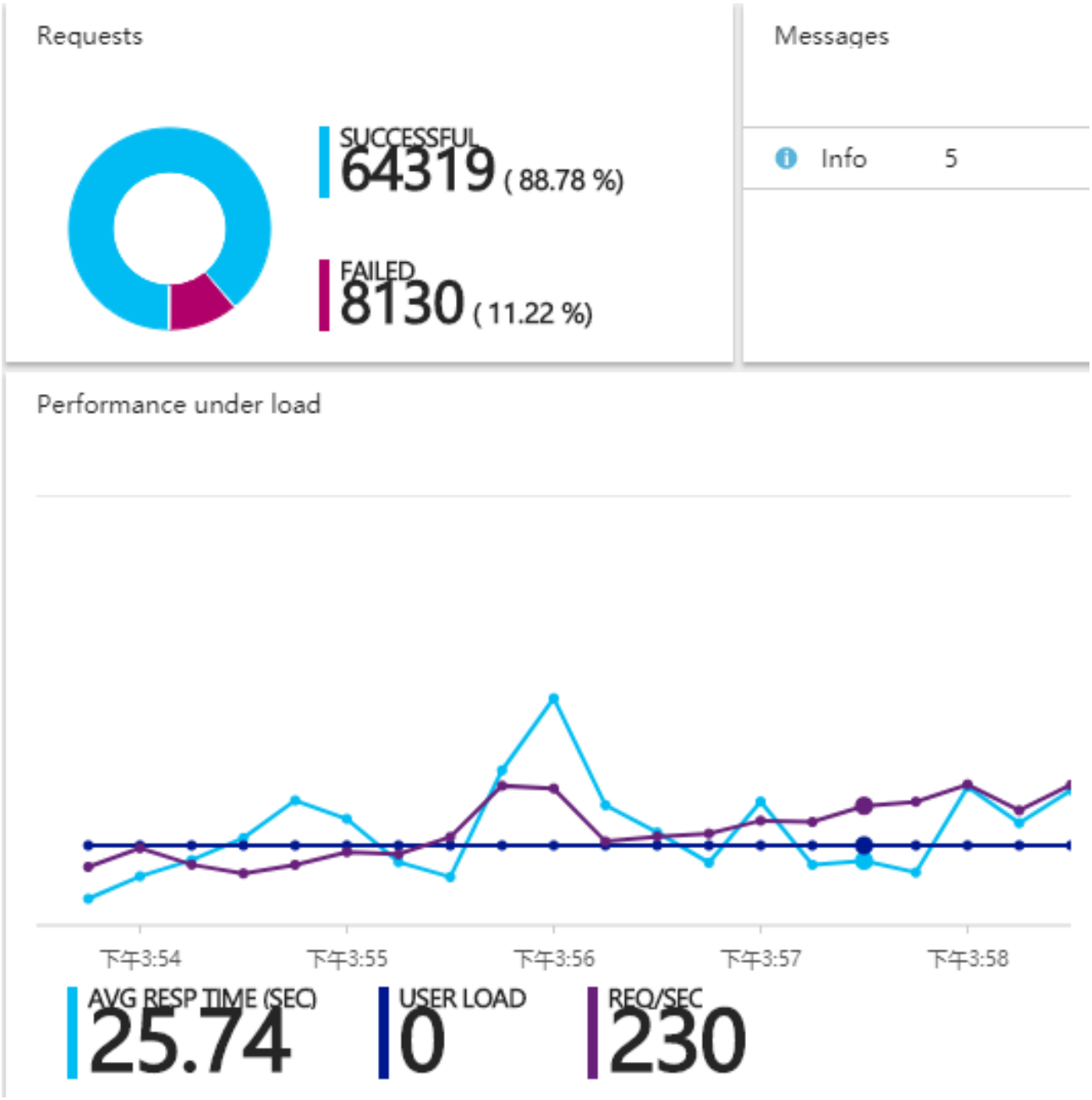


Figure 22: Azure Performance Test Results with 2000 users

F Azure Error Details

Request Failures

REQUEST	TYPE	SUBTYPE	COUNT	LAST MESSAGE
-	Exception	LoadTestErrorL...	2	More than 100...
http://iotaserv....	HttpError	502 - BadGate...	151	502 - BadGate...
http://iotaserv....	HttpError	502 - BadGate...	141	502 - BadGate...
http://iotaserv....	HttpError	502 - BadGate...	137	502 - BadGate...
http://iotaserv....	HttpError	502 - BadGate...	7	502 - BadGate...
http://iotaserv....	HttpError	502 - BadGate...	150	502 - BadGate...
http://iotaserv....	HttpError	502 - BadGate...	7	502 - BadGate...
http://iotaserv....	HttpError	502 - BadGate...	147	502 - BadGate...
Request1	HttpError	502 - BadGate...	81	502 - BadGate...
http://iotaserv....	HttpError	502 - BadGate...	156	502 - BadGate...
http://iotaserv....	HttpError	502 - BadGate...	5	502 - BadGate...
http://iotaserv....	HttpError	502 - BadGate...	8	502 - BadGate...
http://iotaserv....	HttpError	502 - BadGate...	5	502 - BadGate...
http://iotaserv....	HttpError	502 - BadGate...	2	502 - BadGate...
http://iotaserv....	HttpError	502 - BadGate...	1	502 - BadGate...
http://iotaserv....	HttpError	502 - BadGate...	1	502 - BadGate...
http://iotaserv....	HttpError	502 - BadGate...	1	502 - BadGate...

Figure 23: Azure Performance Test Results with 2000 users: Failures Details

G Katalon Test Script

05/03/2018 Script1519756363940.groovy.txt

```
import static com.kms.katalon.core.checkpoint.CheckpointFactory.findCheckpoint
import static com.kms.katalon.core.testcase.TestCaseFactory.findTestCase
import static com.kms.katalon.core.testdata.TestDataFactory.findTestData
import static com.kms.katalon.core.testobject.ObjectRepository.findTestObject
import com.kms.katalon.core.checkpoint.Checkpoint as Checkpoint
import com.kms.katalon.core.checkpoint.CheckpointFactory as CheckpointFactory
import com.kms.katalon.core.mobile.keyword.MobileBuiltInKeywords as MobileBuiltInKeywords
import com.kms.katalon.core.mobile.keyword.MobileBuiltInKeywords as Mobile
import com.kms.katalon.core.model.FailureHandling as FailureHandling
import com.kms.katalon.core.testcase.TestCase as TestCase
import com.kms.katalon.core.testcase.TestCaseFactory as TestCaseFactory
import com.kms.katalon.core.testdata.TestData as TestData
import com.kms.katalon.core.testdata.TestDataFactory as TestDataFactory
import com.kms.katalon.core.testobject.ObjectRepository as ObjectRepository
import com.kms.katalon.core.testobject.TestObject as TestObject
import com.kms.katalon.core.webservice.keyword.WSBuiltInKeywords as WSBuiltInKeywords
import com.kms.katalon.core.webservice.keyword.WSBuiltInKeywords as WS
import com.kms.katalon.core.webui.keyword.WebUiBuiltInKeywords as WebUiBuiltInKeywords
import com.kms.katalon.core.webui.keyword.WebUiBuiltInKeywords as WebUI
import internal.GlobalVariable as GlobalVariable
import org.openqa.selenium.Keys as Keys

WebUI.openBrowser('')

WebUI.navigateToUrl('http://51.140.113.215:3000/')
WebUI.click(findTestObject('Page_IOTA Node Visualizer/img'))
WebUI.click(findTestObject('Page_IOTA Node Visualizer/a_Map'))
WebUI.click(findTestObject('Page_IOTA Node Visualizer/a_About'))
WebUI.click(findTestObject('Page_IOTA Node Visualizer/a_Contact Us'))
WebUI.click(findTestObject('Page_IOTA Node Visualizer/a_Map'))
WebUI.click(findTestObject('Page_IOTA Node Visualizer/span_handle'))
WebUI.click(findTestObject('Page_IOTA Node Visualizer/div_limit to 4k'))
WebUI.click(findTestObject('Page_IOTA Node Visualizer/button__simple-switch-track'))
WebUI.click(findTestObject('Page_IOTA Node Visualizer/button__simple-switch-track_1'))
WebUI.click(findTestObject('Page_IOTA Node Visualizer/html_IOTA Node Visualizer'))
WebUI.click(findTestObject('Page_IOTA Node Visualizer/html_IOTA Node Visualizer'))
WebUI.click(findTestObject('Page_IOTA Node Visualizer/th_Hash'))
WebUI.click(findTestObject('Page_IOTA Node Visualizer/td_dongxiao huang'))
WebUI.doubleClick(findTestObject('Page_IOTA Node Visualizer/canvas'))
WebUI.doubleClick(findTestObject('Page_IOTA Node Visualizer/div_Total Tips      10027'))
WebUI.click(findTestObject('Page_IOTA Node Visualizer/html_IOTA Node Visualizer'))
WebUI.click(findTestObject('Page_IOTA Node Visualizer/body_Map'))
WebUI.click(findTestObject('Page_IOTA Node Visualizer/html_IOTA Node Visualizer'))
WebUI.closeBrowser()
```

file:///homes/as5017/Desktop/Script1519756363940.groovy.txt 1/1

Figure 24: Automated Test Script

H Gitlab

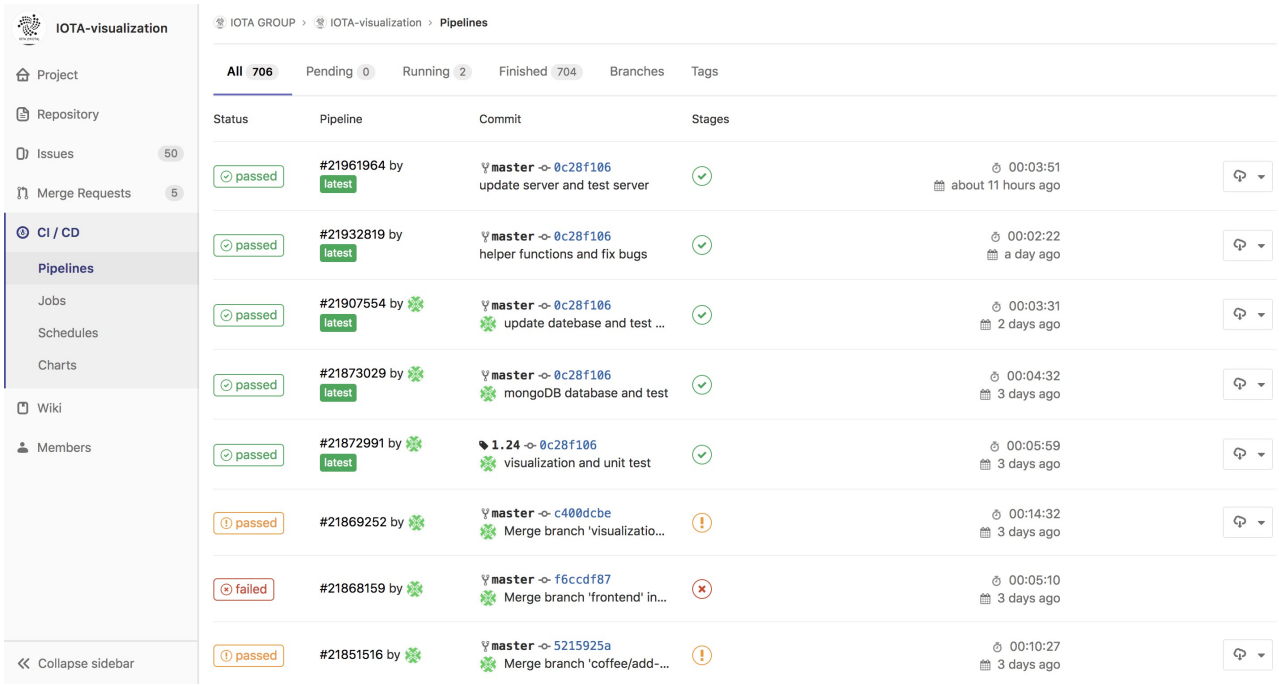


Figure 25: Gitlab Integrated Regression Testing

I User Story

1

Information

ID Test Suite/1
Description

Start 2018-02-27 18:34:27 **End** 2018-02-27 18:34:35
Elapsed 8.044s
Status PASSED

Details

#	Description	Elapsed	Status
1	User opens the Browser with Chrome, Firefox and Internet Explorer Browser is opened with url: ''	3.535s	PASSED
2	User Client connects to Server Navigate to 'http://51.140.113.215:3000/' successfully	1.642s	PASSED
3	User clicks on the node, highlighting relevant information Object: 'Object Repository/Page_IOTA Node Visualizer/img'	0.221s	PASSED
4	User hovers over a node, prompting a popup displaying node details Object: 'Object Repository/Page_IOTA Node Visualizer/node' displays tip	0.711s	PASSED
5	User drags node out of place Object: 'Object Repository/Page_IOTA Node Visualizer/block' is dragged	0.294s	PASSED
6	User searches for node using hash Object: 'Object Repository/Page_IOTA Node Visualizer/search' hash	0.317s	PASSED
7	User views aggregate statistics Object: 'Object Repository/Page_IOTA Node Visualizer/stat' displays +2	0.258s	WARNING

Figure 26: User Story test results

J API Testing

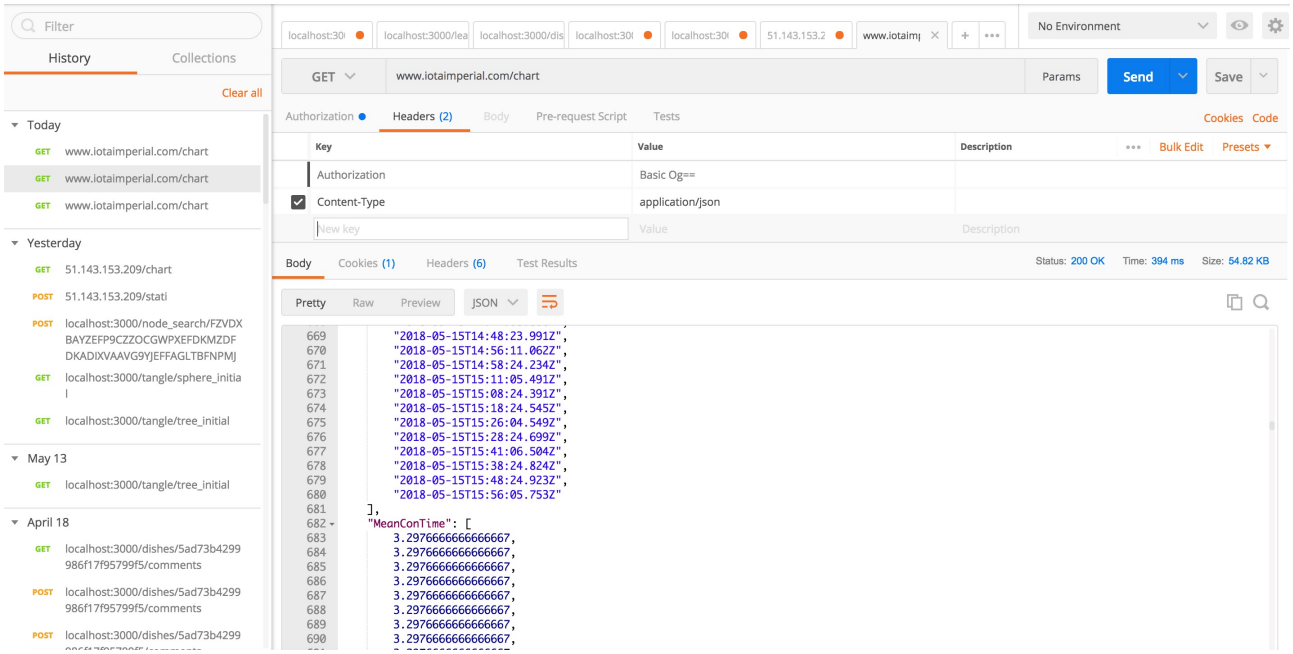


Figure 27: User Story test results

References

[1] Almende B.V. *Node Graph Drawing Library*. 2018. URL: `http://visjs.org/`.

[2] IOTA Foundation. *IOTA API*. 2018. URL: `https://iota.readme.io/v1.2.0/reference`.

[3] MongoDB Inc. *MongoDB*. 2018. URL: `https://www.mongodb.com/`.

[4] Team IOTA. *Report One*. 2018. URL: `https://www.overleaf.com/read/fpqkmjzfnpkd`.

[5] iotasear.ch. *iotasear.ch*. 2018. URL: `https://iotasear.ch/`.

[6] kryptonauten. *the tangle*. 2018. URL: `http://tangle.glumb.de/`.

[7] Microsoft. *Performance Test*. 2018. URL: `https://docs.microsoft.com/en-us/vsts/load-test/performance-reports`.

[8] MochaJS. *JavaScript Tester*. 2018. URL: `https://mochajs.org/`.

[9] SigmaJS. *Graph Drawing Library*. 2018. URL: `http://sigmajs.org/`.

[10] SinonJS. *Stub Generator*. 2018. URL: `http://sinonjs.org/`.

[11] Neo Technology. *graph database platform*. 2018. URL: `https://neo4j.com/`.