

|  |
| --- |
| **Assignment Cover Letter**    **(Group Work)** |

**Student Information**: **Surname**  **Given Names Student ID Number**

1. **Tandra** **Vincentius** **2301894804**

2. **Tanjaya** **Ravel** **2301890320**

**Course Code**  **: COMP6571**  **Course Name**  **: Data Structure and Algorithms**

**Class** **: L2AC**  **Name of Lecturer(s)** **:** **Andreas Kurniawan**

|  |
| --- |
|  |

**Major**  **: CS**

**Title of Assignment** :

(if any)

**Type of Assignment**  **: Final Project**

**Submission Pattern**

**Due Date**  **:** 18/06/2020   **Submission Date**  **:** 18/06/2020

The assignment should meet the below requirements.

1. Assignment (hard copy) is required to be submitted on clean paper, and (soft copy) as per lecturer’s instructions.
2. Soft copy assignment also requires the signed (hardcopy) submission of this form, which automatically validates the softcopy submission.
3. The above information is complete and legible.
4. Compiled pages are firmly stapled.
5. Assignment has been copied (soft copy and hard copy) for each student ahead of the submission.

**Plagiarism/Cheating**

BiNus International seriously regards all forms of plagiarism, cheating and collusion as academic offenses which may result in severe penalties, including loss/drop of marks, course/class discontinuity and other possible penalties executed by the university. Please refer to the related course syllabus for further information.

**Declaration of Originality**

By signing this assignment, I understand, accept and consent to BiNus International terms and policy on plagiarism. Herewith I declare that the work contained in this assignment is my own work and has not been submitted for the use of assessment in another course or class, except where this has been notified and accepted in advance.

Signature of Student: (Name of Student)

1. Vincentius Gabriel Tandra
2. Ravel Tanjaya

**Project Name:**

HashDB

**Introduction:**

Our project mainly focuses on finding best data structure for storing specific non uniform data inside a NOSQL database. We chose this topic because we felt like it was a problem that we could apply to the real world where the market for the nosql database continues to expand.

**Problem Description:**

One of the ways to store non uniform data is by using a nosql database because this kind of database does not use a predefined structure which means all of the data that is stored inside the database may be unique.

In recent years, the NOSQL paradigm of databases’ popularity is increasing greatly, this is caused by the fact that data nowadays is not structured and uniform enough which means that storing this data on a traditional RDBMS (relational database management system) may not be efficient and that’s why NOSQL database is a very hot topic these days with a lot of NOSQL paradigm emerging on the market, like Redis, MongoDB, Firestore etc.

With the large variety of NOSQL paradigms currently on the market, users may not be able to deduce the most efficient one out of the bunch. This is why we will choose a certain condition and then evaluate the best paradigm for that condition.

**Condition:**

    In this project, we will be evaluating which data structure that is best suited for a food delivery application using a benchmarking applicatoin (written in JavaScript object format):

|  |
| --- |
| data = {     Customer: [         Budi: 128000,         joko: 50000,     ],     Orders: [         Fsdfkkjgkfdgjfkd: {             Food: [                 "pizza", "donut"             ],             Price: 12500,             status: "waiting"         },         fsdagfdgfdgfdhh: {             Food: [                 "pizza", "donut"             ],             Price: 12500,             status: "waiting"         },     ] } |

**Data Structures Chosen:**

For the problem we have chosen, we had to narrow down to a selection of various data structures. In this case, we were to use structures that could be used to implement a nosql database. These included trees or hash based structures. We were able to test many structures in the limited time we had and were able to evaluate the best structure from the group. Several structures were tested but there were some that we failed to implement.

During the time we took to develop an app for the project we used the previous condition to benchmark a number of data structures to find the most effective one for our implementation. For our tests we were testing the insertion and deletion functions of each structure.

Data Structures Tested:

* Hash Table
* Hash Tree

Hash Table: A hash table is a data structure which stores data in an associative manner. In a hash table, the data is stored in an array format, where each data value has its own unique index value. The data inside a hash table is stored in key value pairs. It uses a technique called hashing to generate the indexes where these key value pairs are then stored. An advantage of using a Hash Table is that they are efficient for inserting and searching data. However, when running into collisions they become quite inefficient.

Many databases now implement the hash table for database indexing, we decided to test it to see why it is such a popular choice for handling data. Its average time complexity for its insertion and deletion algorithms in Big O notation are O (1).

Hash Tree: A hash tree is a hash-based data structure composed of a tree of hashes where the leaves of the tree are hashes of data blocks. It is a generalization of a data structure known as a hash list. One of its advantages is that it is very useful in verifying data.

A good example of where a hash tree is used is in the digital cryptocurrency, Bitcoin where transactions using bit coin are stored in blocks known as a blockchain, leaves of the hash tree are typically hashes of single blocks, and is used to verify changes within the blockchain. We tested this structure in order to see whether or not it could compare to the other structures we were testing. Its average time complexity for its insertion and deletion algorithms in Big O notation are O (log2n).

**Benchmarking Application:**

Each of these data structures were tested in a benchmarking application we developed in order to compare the relative speeds of each of their functional algorithms. In our case, these are their search and delete algorithms. The application functions as follows:

It uses a Nodejs based build system to automate the build process, a hyperfine based benchmark system and Random Data generation based on faker.js and was developed under a unix based system.

**NodeJS build system**

Currently to test the speed and efficiency of the data structure nodejs is used to automate the whole process including the building, generation of random data and benchmarking.

In order to use these scripts please make sure you have installed the following:

* NodeJs
* npm
* Hyperfine

To use this script please first, change directory to the benchmark folder with:

cd benchmark

the use npm to install all of the required depedency.

npm install

to install all required npm depedencies.

**Automatically build all of the codes**

To automatically build all of the available codes, we can run this command:

npm run build

The script will change directory to each of the codes and will use gcc to build all files with c++17 standard.

**Generate the random data**

The random data that is generated is of JSON(Javascript object notation) these data will later be loaded by the programs and used as a sample data to measure the speed of each data structure.

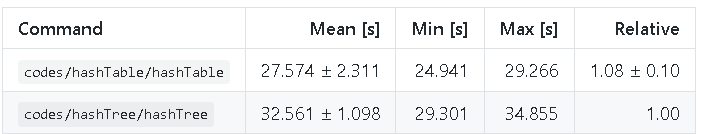
To generate/regenerate these random data please run:

npm run generate

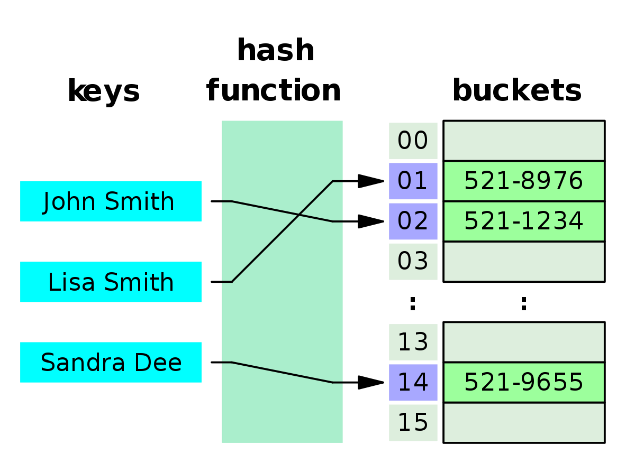
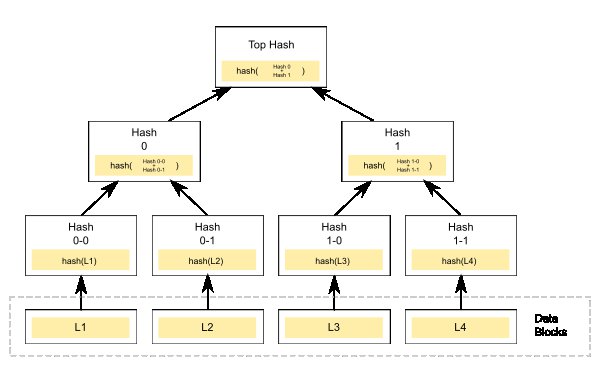
**Benchmarking**

In order to benchmark, please make sure you have already build for the latest changes and generate the json data,then run:

npm run benchmark

**Benchmarking Results**

After benchmarking each structure, we found the **hash table** to be the most efficient data structure for the choice. Here we see the hash table head to head with the hash tree and as expected from the differencess in time complexity, it beats it out in speed even while handling the large amounts of data. So through these results we can see the hash table being faster but besides the differences in time complexity, we can attempt to figure out why this is exactly the case.



Here we see a respresentation of both structures, the hash table assigns a key value pair to a bucket where they are stored while a hash tree is structured similar to a binary search tree in which they have a branching factor of 2 (the number of children at each node). However, unlike a binary search tree where generating the root node is simple, with a merkle tree you would have to compute and store a hash at every node which adds up as the tree gets larger and larger. This combined with its branching factor makes the hash tree substantially slower than the hash table when handling large amounts of data.

One of the advantages a tree would hold over the hash table is its ability to store data in an organized and ordered structure. However, our use-case describes the best way to store non-uniform data in a nosql database and so, the non-ordered data makes this advantage less favorable for the hash tree and better shows why the hash table is faster. The hash table is a structure that functions relative to its function and only slows down when running into collisions which are always expected to happen, especially with larger data sets even with the ideal hash function. However, with proper collision resolution using open-addressing methods we are able to resolve these collisions so the hash table works at its best.

**HashDB Demo**

After we benchmarked each respective structure, we used the hash table as the data structure for our implementation of a NOSQL database. The hash table implemtation we are using is quite special, as mentioned before we are using an open adress hash table, this hash table contains a linked list data structure inside of in which we use as a container for another linked list. Our hash table is able to store two different types of objects, these are **Documents** which would contain regular data and **Collections** which are collections of these documents. In summary, the hash table acts as a collection while the inner linked list acts as a document. We decided not to use GUI but instead established a server-client connection. We can use the application we created now called HashDB in order to access the database and interact with it using what is known as a RESTful API which is an interface that allows us to make HTTP requests.

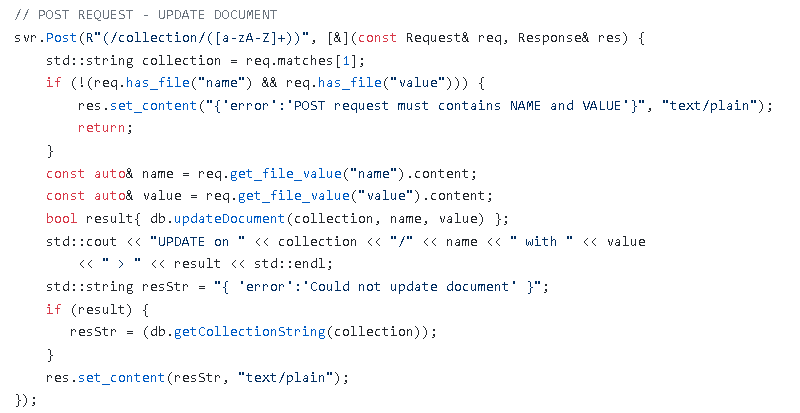
Previously in our benchmarking application, our data was structured in JSON notation. The NOSQL database, unlike a regular relational dabase does not store data in the form of tables filled with rows and columns, instead it stores the data in a document format similarly to the JSON format, it is perfect for storing our random, unstructured, non-uniform data which is already in JSON fomat.

For the creation of our database, we took inspiration from Google’s Firebase Realtime Database, a real-time NOSQL database. However, there are some features of our application that offers some advantages over this database. Firebase establishes an immediate connectoin to their cllients, while our database requires a webserver connection first. We are also able to make pure HTTP requests with no prior knowledge of a quey language. We can make requests to the server using an HTTP tool such as Postman. Our REST api allows us to make a variety of requests towards the server, these include **Get**, **Delete**, **Post** and **Put**. Each of these types of requests interact with the document and the collection objects.

In an example using our code, this is how you would make a get request for a collection or document. It will then print out the selected collection or document depending on the request.



In this example, we can see the code we use to make a request in order to delete either a document or a collection.

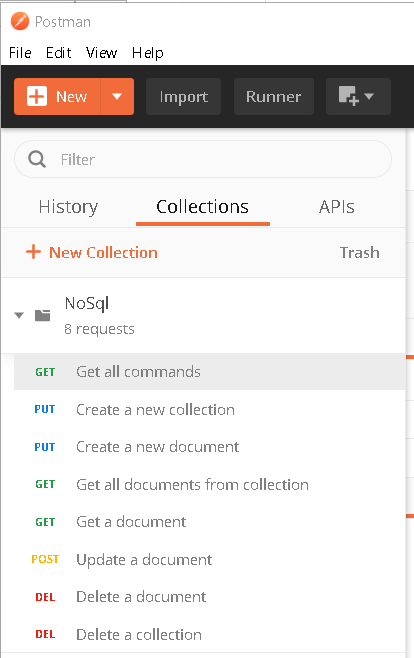


In this example, we can see the post request being used to update a pre-existing document with a different value. This means that you cannot update a document if it is not valid.

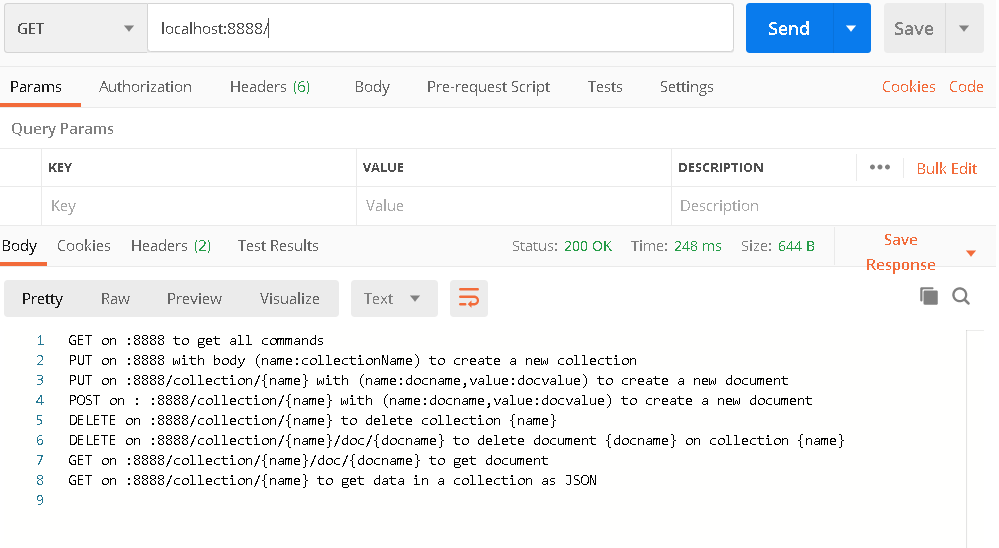


In this example, the put code is shown. The put request allows the client to make a request to create a new document or a collection. This code needs to be run first in order to use the other commands. Now that we have shown the code, we can show the code in action.

Once the app is opened, a connection is established and we can start using our HTTP tool to make requests to access the database. In our case we are using Postman and this demo is being done on a Windows machine.

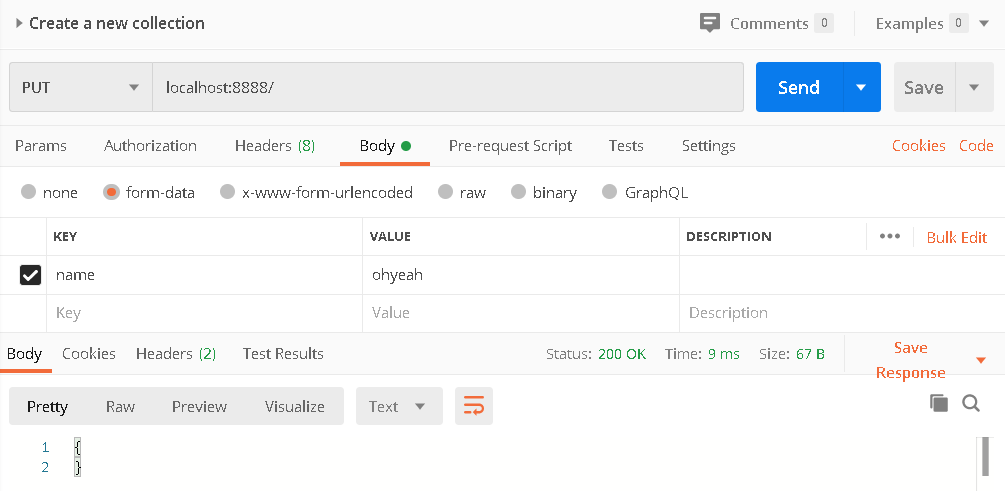


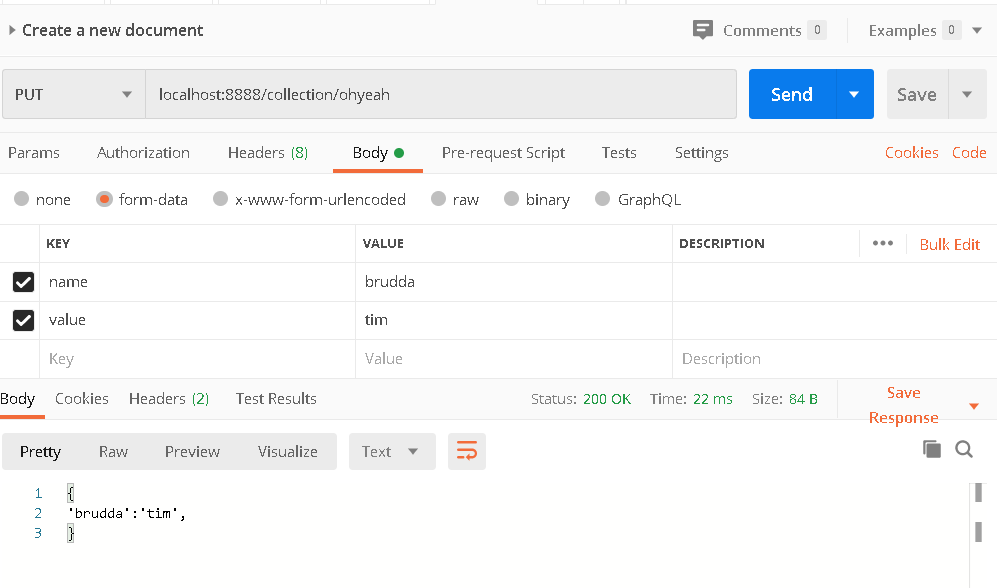
After creating a request, we can import the collection file we are using in order to execute any of these respective requests. The first request which can be made immediately is the get request.



As seen here, a response from the server is returned and it prints out these lines explaining how to use each command manually through the Postman client.

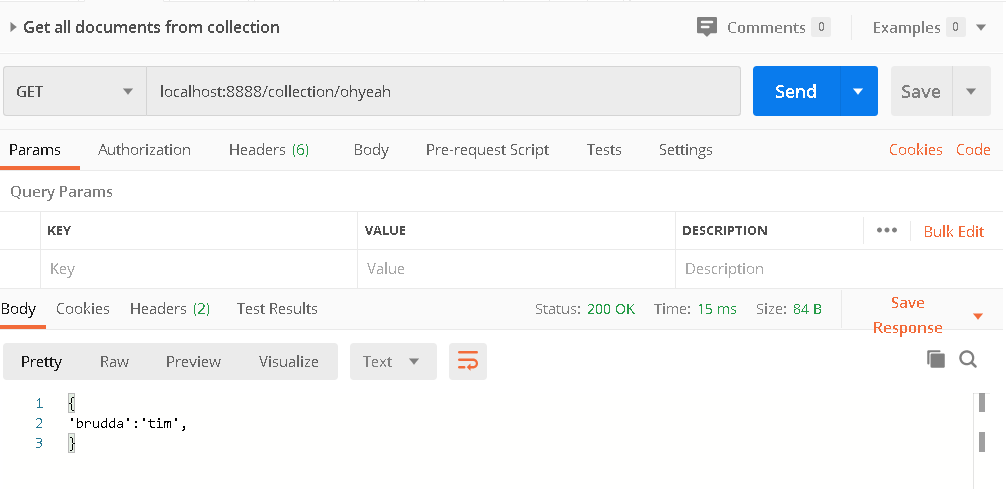
Here we can see more examples of how the app can be used.

This is an example of a put request being used to create a new collection named “ohyeah”

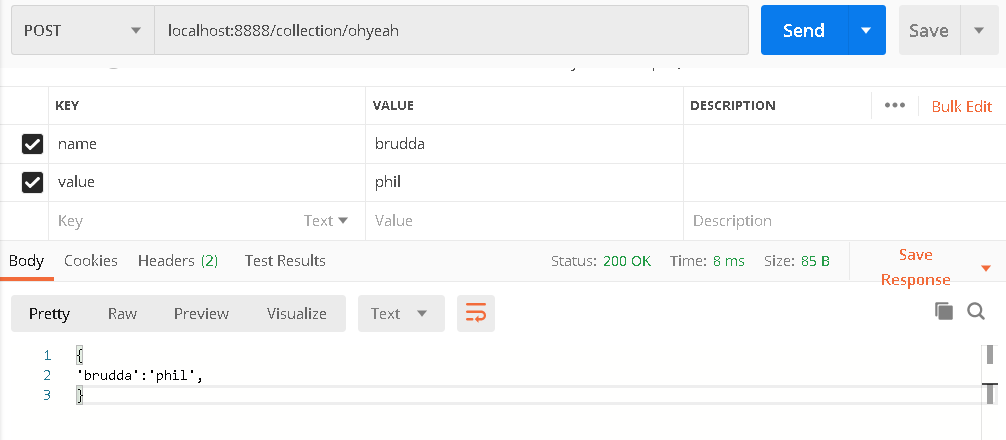


This is an example of a put request being used to create a document named “brudda” with a value of “tim” under the previous collection called “ohyeah”.

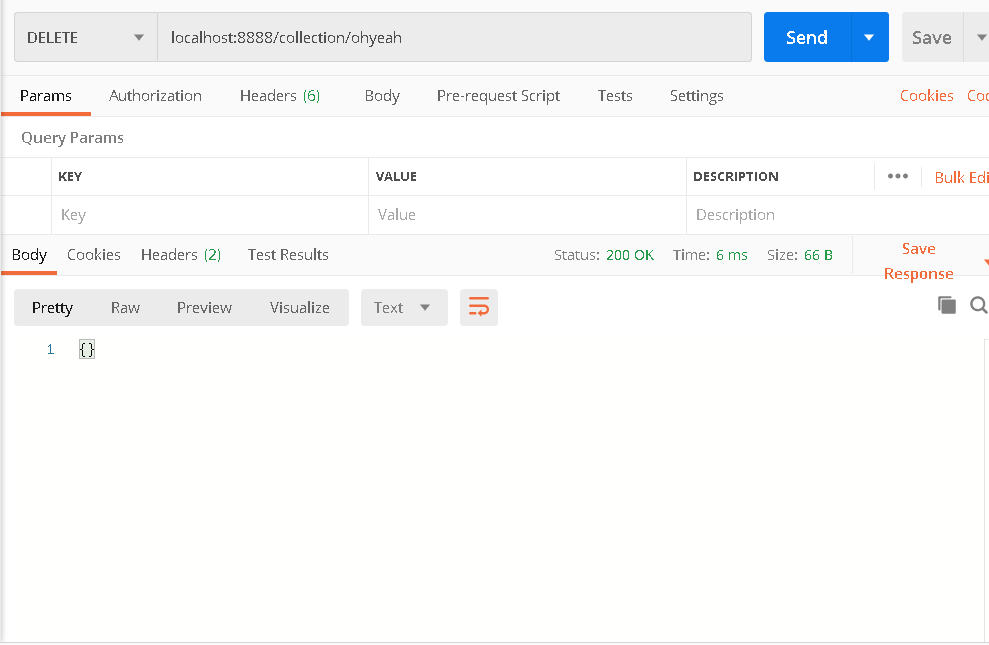
We are also able to get all data in a collection.



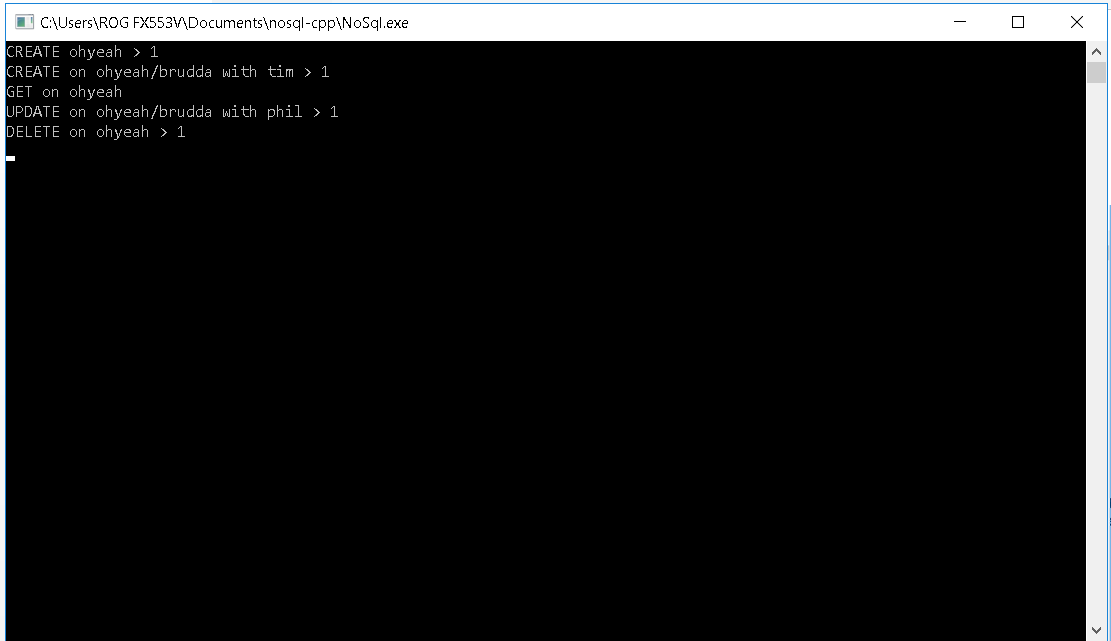
We can also update our data using the post request like this.



And last of all we have our delete command which we can use on either a document in the collection or on the whole collection itself. In this case the “ohyeah” collection and its document is being deleted.



Meanwhile, our HashDB app is also saving logs of each command like this:



**Application Details**

**Background**

We wanted to create a database that is easy to use by taking inspiration from Google’s Firebase database, which focuses more to clients who don’t want to create a backend, our database allows a user to access the database using HTTP requests but this is not an API that we can just put as a webserver and dsitribute to the public as it doesn’t support any authentication and other middleware and so they must be implemented in another webserverm using golang, node or PHP.

So why do we use the port 8888 unlike Firebase’ port 8080 which a user can use to directly use the database as a webserver? The reason why is because Firebase supports security rules that allows access to a database selectively but our database does not support this, Firebase was intded to be a platform for users who don’t want to create a server but we plan to enchance the server creation process by providing an alternative way to connect to the database. To elaborate more on the hash table we are using, we can explain why we use a double linked list in our implementation. Since our hash function is not perfect, when a conflict appears regarding a similar address, we use open addressing to place it in a linked list to resolve these conflicts, it will reduce thewhole memory usage because the the object is created on the heap on request so when the object is not created yet ,it doesn't take any space. Lastly, why did we choose the JSON format in the first place? The simple reason is that despite the many alternative formats available to the API server such as proto format or XML, JSON is by far the most popular data interchange format in use and because it is so popular in today’s market, many are familiar with the use of the JSON format.

**Demo Video**

https://www.youtube.com/watch?v=dou3ZJsuR9U&feature=youtu.be

**Github Link**

https://github.com/ravelgwong/nosql-cpp

Within this github link, all of our code is stored, this includes our main application, the benchmarking application, previous reports and other files which were used to create this project.

**Contribution and Role towards this project**

For a better description of how the workload was dsitributed during the creation of this project, this section of the report will be written out by each group member so they can share their experiences working with one another and they can give their honest take on their contributions towards the final product.

**Group member (Vincentius Gabriel Tandra)**

When i first received the specifications for this project, I was quite taken aback. After all, we had done implementations of several data structures in class before but we had never really applied this knowledge directly for situations such as solving a problen. Being less experienced, at this point I had only followed along my group and agreed on a topic that sounded satisfactory. It was quite difficult at first to follow the concept of what we were creating and what problem it would solve but after some more research online and a better idea of what exactly we were creating,

The aforementioned benchmarking application was something I had only just seen and I had no idea that we could directly test a structure with some data and it would yield results similar to the concept of Big O notation in class. We were to test several data structure to determine the best through this benchmark. Then i received my first task for the project, I had to create an implementation of a data structue known as a B-Tree. It was a common data structure used in databases for things such as indexing and after researching more about it online, I began work. After about an unsuccessful 3 weeks of attempting to implement the structure, I was advised to scrap it and try again with another structure. C++ was full of open source libraries online and it was only a matter of time before we were able to find other data structures to test. Even so, I continued to struggle and was continuously unable to find a library that suited our needs and became less and less motivated to find a solution. Meanwhile, we had to submit our first report and second report and at that time no structure other than the hash table previously made by my group partner was build and benchmark ready.

Approaching 2 weeks from the deadline, I was presssured in order to find a solution and pushed on to find a structure that suited the code we had made thus far and was applicable for our project. A day before our personal projected deadline, I managed to find a working library for a data structure we could use in the project. Finally, I had finished my first task. Up to this point, I had little to do with the code involved and while I was fascinated with our concept and problem, I was less capable of doing the hard coding that resulted in either application. Instead, I leaned more towards creating reports and adding analysis to each data structure. Working with a lot of the data structures I learnt a lot more about each of them and each of their advantages and disadvantages or their uses in the real world. I feel like I have many areas to improve in coding and also in managing my life. Had I been more efficient from the start, perhaps we could have moved a bit faster and in future projects I hope it will help me go forward instead of backwards. I was quick to give up and unwilling to look for the solution right in front of me which made me unreasonable and less focused in general. However, what I lack is made up for by my group partner who created the framework and content for most of the code and was patient enough with me so that at the end of the day, we could enjoy the project and learn something at the very end. Overall, my experience working in this group was enjoyable , I contributed towards the analysis and implementation of data structures and report writing. I aim to become a more productive group member who can work quickly and respnsibly in the future.

**Group member (Ravel Tanjaya)**

**Conclusion**

In conclusion, we were able to successfullly solve the problem we defined for our project, we were able to find the best data structure for our implementation of a NOSQL database using a hash table by comparing structures using our benchmarking application. We were also able to complete an application that we can use to access the contents of the database and also improve upon already existing databases using our own implementation for the database.