PriMera Scientific Engineering
Volume 3 Issue 1 July 2023
DOI: 10.56831/PSEN-02-060
ISSN: 2834-2550

PriMera Scientific Publications

Preventing Insider Attacks in Databases

Type: Thesis
Received: May 30, 2023
Published: June 06, 2023

Citation:
Vijay K Chaurasiya., et al. "Preventing Insider Attacks in Databases". PriMera Scientific Engineering 3.1 (2023): 03-32.

Sourav Mishra and Vijay K Chaurasiya*

Dept. of Information Technology, Indian Institute of Information Technology Allahabad, Prayagraj, India

***Corresponding Author:** Vijay K Chaurasiya, Dept. of Information Technology, Indian Institute of Information Technology Allahabad, Prayagraj, India.

Abstract

Many web applications that rely on centralized databases face vulnerabilities to insider attacks. While these systems implement multiple layers of security measures against external hackers, they may overlook the threat posed by employees who are already within these security layers and have access to privileged information. Users with administrative privileges in the database system can potentially access, modify, or delete data, while also manipulating corresponding log entries to erase any evidence of tampering, making detection nearly impossible. While one approach could involve developing methods to detect and trace such attacks, along with recovering the original data, this report takes a different perspective. Instead of focusing on detection and recovery, we explore a new direction: ensuring that attacks do not occur in the first place. By establishing a system that comprehensively safeguards data integrity, the need for detection, tracing, and recovery can be minimized or eliminated. This report investigates the prevention of insider attacks on databases by utilizing Bluzelle, a NoSQL database that offers decentralized database solutions for decentralized applications.

Keywords: Tampering; Centralized Database Systems; Insider Attack; Detection; Recovery; Integrity; Bluzelle

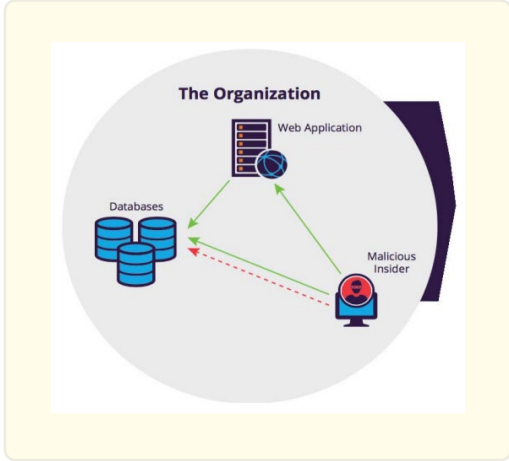
Introduction

Bradley Manning and Edward Snowden who were a part of the US military and were considered trusted people, stole critical intelligence documents and released them. The worst came later in 2015 when the Office of Personal Management of the US disclosed that someone had acquired personal information of 22 million federal employees, including employees with the highest level of security clearance. Whoever the thief, they put the security of an entire country at risk. Did they get caught by the security measures in place? No.

Here we are talking about preventing attacks coming from people we trust. This can be achieved by monitoring behavior of users and how they interact with the organization's database, analyze the query logs which are records of questions that people ask to the database and the way they ask them. One may think that we are overreacting, we should trust our employees which is completely correct, but it is better to be prepared than being sorry later.

An insider attack is launched by an authorized personnel who is familiar with the network architecture and is aware of the system policies and procedures. There is usually less security against insider attacks because many organizations focus on protection from external attacks. Insider attacks can affect security elements of the system, steal sensitive data, inject trojan viruses in the system, affect availability of the system by overloading storage or processing capacity which will ultimately lead to system crash. Internal intrusion detection systems (IDS) protect organizations against insider attacks but their deployment is far from easy. Employees must be made aware of certain rules and regulations which ensure that no false alarms are raised.

"In 2008, an incident occurred involving Terry Childs, a network engineer employed by the San Francisco Department of Telecommunications and Information Services. During this incident, Childs modified the network passwords, effectively restricting access to FiberWAN for a duration of 12 days. Subsequently, Childs was convicted of felony network tampering. The city of San Francisco incurred significant costs in the process of regaining control of the affected systems, amounting to $900,000. Moreover, the insider attack had far-reaching consequences, impacting approximately 60 percent of the city's services.

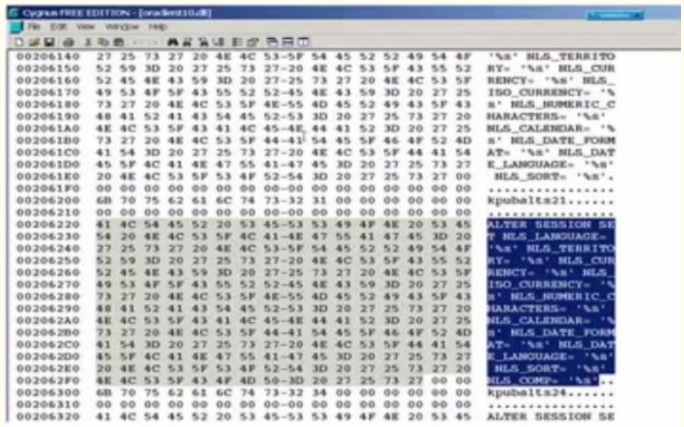


A lot of web or mobile applications that we use today rely on centralized database systems. Normally database servers are secured behind layers of firewalls and certain access control policies are implemented which protect the database from external attacks but do not fare well against insider threats. This is because these policies provide privileged access to certain users thereby enabling them to modify the database entries and remove the corresponding database and system logs which makes detection impossible.

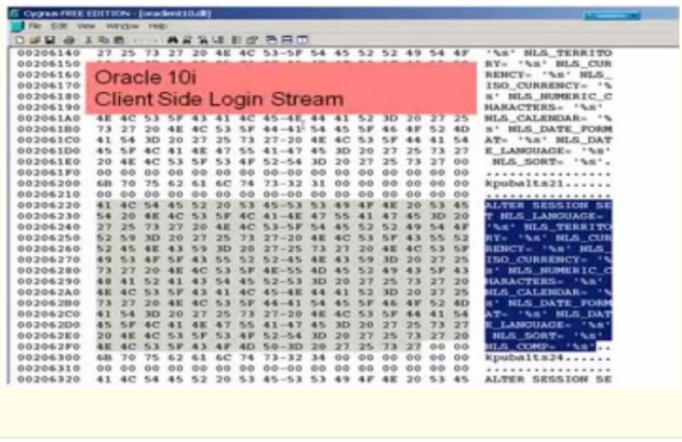
Client Side Database Protocol Attack

Each database vendor essentially has a proprietary network protocol and they use this for communication and commands. This is generally a highly complex and unfortunately very obscure protocol and they often change. This makes them prone to security vulnerabilities and some of the consequences of this are unauthorized access, manipulation. There are ways to mitigate such attacks : one is protocol validation engines which can address even the unknown vulnerabilities i.e. we let normal client generated messages get through and anything that has hidden qualities or features we drop them and the other one is reactive protocol validation measure which is going to address known vulnerabilities and this will check for specific known attacks.

This attack is a client-side vulnerability. A hex editor is used for looking at some Oracle information.

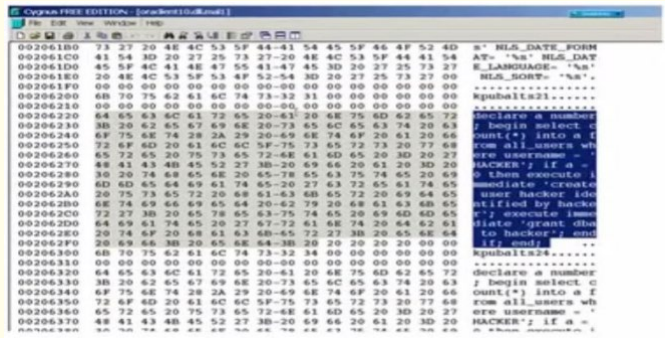


This is some DLL information that happens to be in hexadecimal format. The information highlighted on the right this is just a simple query and it says ALTER SESSION SET NLS_LANGUAGE etc etc. This query this is actually the login stream for a sql query from a client. The attacker takes advantage of this. He uses his sql client when he logs in to the database and launches the vulnerability and creates a new user for himself.

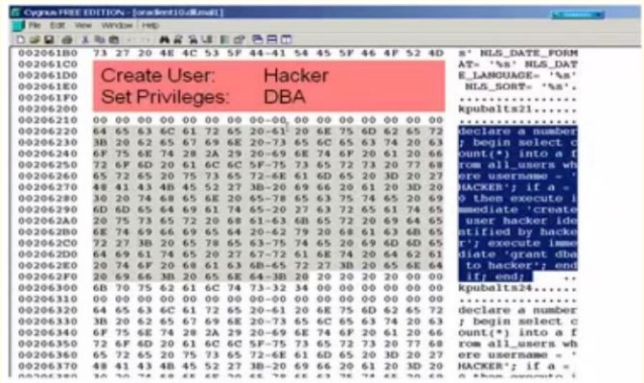


By modifying information on the client the attacker creates a new user on the database. This is very interesting because the actual user login does not have to even be successful. This can be a completely unsuccessful login. It is the process of logging in that will actually take advantage of the vulnerability.

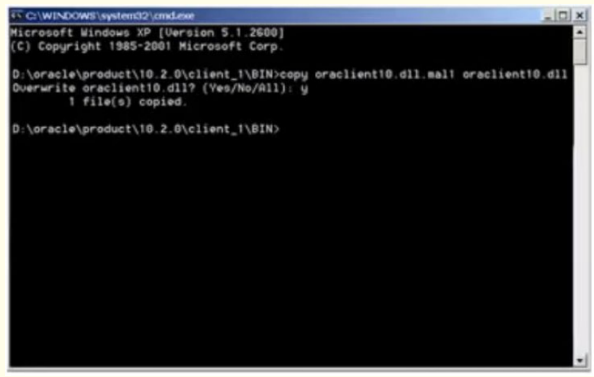
Now this works in the unpatched version of Oracle 10 because the database user in Oracle actually operates with DBA privileges. So if the attacker causes that user to fail via say a buffer overflow for example then he can inject code inside this to actually create a new user or do any number of things that DBA might want to do. The attacker makes some changes here by taking this Oracle 10 file and this client log on stream information and it brings over here.



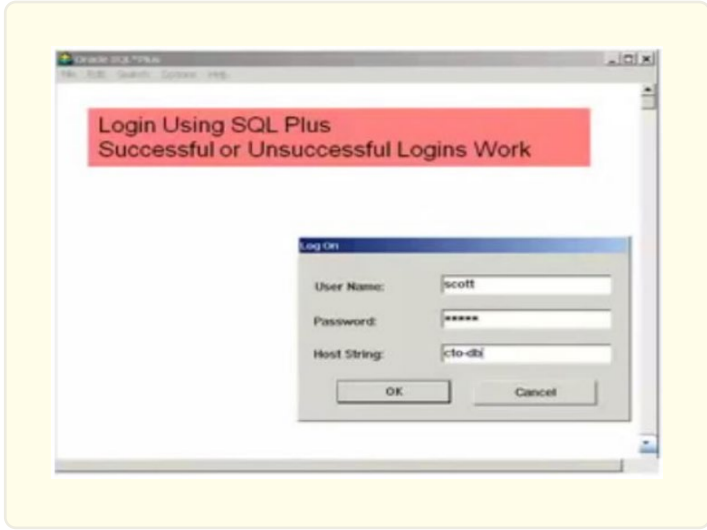
He then edits this to create a user called hacker and set that user's privileges to DBA.



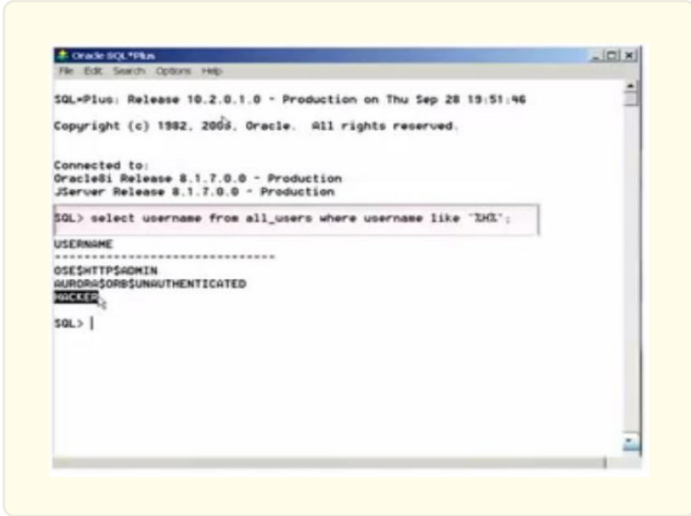
On the right side, in the highlighted area you can see a query that the attacker is executing to grant DBA to "hacker". So in short the attacker created a user called hacker and set it's privileges to DBA. He placed this in the sql login stream on client side for Oracle client 10 DLL (which is just a file that happens to be on client side).



Then he goes into his operating system and modifies the clients. He uses a copy command to overwrite his hacked version over the pre-existing version.



Then he logs in as the user Scott. Whether the login attemp is successful or unsuccessful doesn't really matter at this point. Then using oracle sql+ and as scott he runs the command to select username from all users whose usernames essentially has a "H".



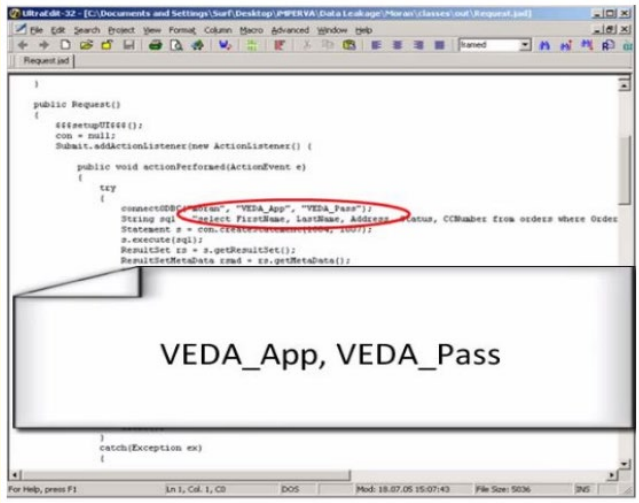
He gets the list of user names and you can see highlighted at the bottom we actually now have the user called "hacker".

Insider Database Privilege Abuse

This example uses a java application this is a standard order form. Many organizations have applications that access the database directly through this means as opposed to a web front- end. Sometimes this is called the fat client or thick client and generally they're installed on the end users desktop.

They connect directly to the database. Thus no web application is required to serve as a bridge between user and database. Depending on how well coded this application is, there could be several vulnerabilities that are actually specific to the application and the backend database. Here we have a malicious insider that has legitimate access to the Java client due to which the simple fact of them(attacker) using it is not going to send up any bells or alarms. So he downloads an application from the web, in this case it is the fast java decompiler.

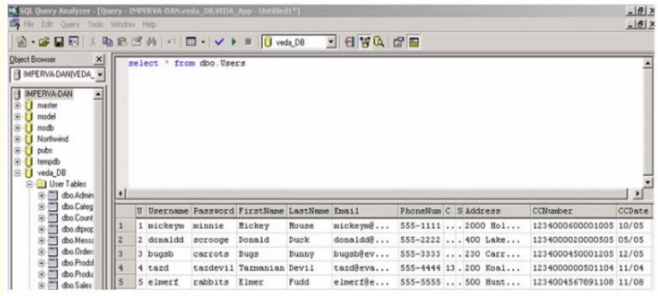
This is going to allow him to decompile the application that resides on his desktop. This decompiles the java application and the source code is revealed. There are certain things here that we might find interesting. First we see the *ODBC connect* and then we see *VEDA_App* and *VEDA_Pass*.



It is actually the database user that this application is coded to operate as within the database. For most clients, their apps will use a single database user to connect to the database. This is very much like pooled user accounts between a web application in a database where you might have 10,000 users on an application but they are really interacting with the database is a single pooled user, Oracle shared accounts, sql shared account something of that nature.

The user authenticates themselves with their own credentials when they leverage their thick client their application connecting to the database but the connection is usually operating as the database user that is coded in here i.e. *VEDA_App*, *VEDA_Pass*. That application is going to maintain all the user rights and all the user restrictions.

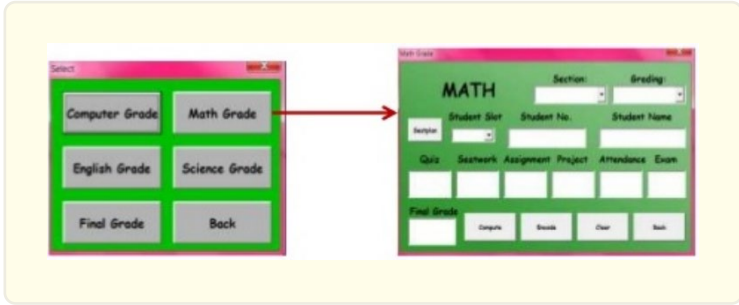
Now the application sits on the desktop, the attacker authenticates against that app but that app is interacting directly with the database. If the attacker operates outside of the application and goes directly to the database then the restrictions the application would have applied are no longer there. He can operate with the privileges of the database user account i.e. *VEDA_App*, *VEDA_Pass* that gives us access to the application.



Now this malicious insider opens up a tool called query analyzer. This tool that ships with Microsoft sql so logging in using *VEDA_App* and *VEDA_Pass* the attacker is able to perform a simple query, in this case a SELECT * FROM DBO USERS which gives him very critical information like username, password, first name, last name, email etc. He can even get credit card information. He can execute any

query and it would run as long as the user that he is leveraging i.e. *VEDA_App*, *VEDA_Pass* has those privileges. In fact this database user account has the privileges to do pretty much anything within this database if we are able to pull up information this sensitive. So it must be remebered that even thick clients need to be monitored even if there are no front-end web applications and this is because monitoring even the raw sql traffic is important to understand how users are interacting with data. This was a user that had legitimate access but who happened to be malicious as well. They were taking advantage of a poorly coded of the Java fat client as well as the poor security restrictions on the database back-end so that they could leak much more sensitive information than they would have if they you had used the thick client for its intended purposes.

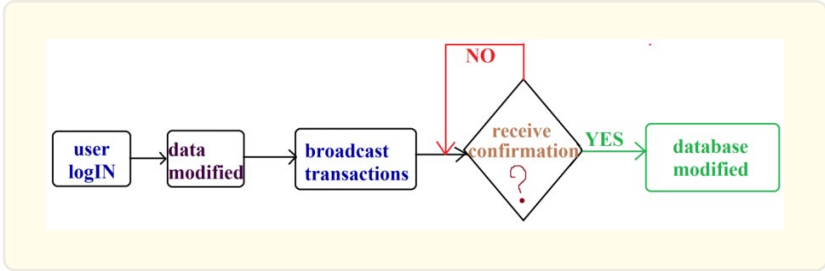
Problem Definition



Let us consider a database application of a university that allows legitimate users to view and/or update grades of students in various courses offered by the institute. The users are required to use their login credentials (user id and password) to access the database. The type of access provided varies according to the user. Students can only view their grades in various subjects where as faculty members can view along with enter/update the grades of students. However there are certain IT administrators who have privileged access to the database which not only enables them to edit the grades of students but also remove the corresponding database and system logs. If this happens then the students may find their grades changed but they won't be able to prove that such changes were not legitimate. Moreover any faculty member also won't be able to attribute such changes to any specific administrator.

Literature Survey

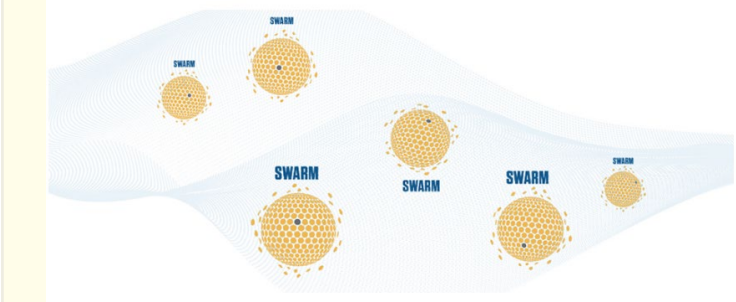
- The paper [2] discusses how data and trust management degrade the efficiency of Intrusion Detection Systems (IDS) and proposes an intersection of blockchain and IDS to protect data integrity and ensure process transparency.
- The paper [4] presents SafeWS, a system that uses cryptographic and software engineering techniques along with data management principles to prevent and detect website hijacking and unauthorized access.
- The paper [1] uses the immutability and tamper-resistant property of blockchains to detect such attacks. They store some metadata corresponding to each database entry on the blockchain. This metadata has a very small overhead but helps verify the database entries when accessed. Whenever a user wishes to update data in the database, the request is broadcasted across the entire network thereby making it public knowledge. If the transaction is confirmed by all the stakeholders then the database is modified otherwise the request is cancelled and the database remains unchanged.



However, tackling the problem of data recovery in the event of the attacker deleting the entire database along with the log files, requires a protocol where the entire blockchain network is constantly monitored by the nodes in a completely random manner. First one node chosen at random monitors the network and is always on the lookout for any change in the blockchain. Then after say 6 seconds another node chosen at random does the same and after say 1 second another node takes its place and after 9 seconds another and so on. The time interval between the nodes' switching should be totally random thereby making it completely impossible for the attacker to exploit this loophole. But while such a protocol may be theoretically sound, it is by no means practical.

Proposed Methodology

We store the data on bluzelle which is a noSQL database that provides decentralized storage of data to dApps as a secured alternative on blockchain. It consists of two tokens: ***BLZ and BNT*** i.e. external token used for investment and internal token which powers up the network and gives the owners of masternode a reason to stay operational. BLZ is publicly traded ERC-20 token while BNT are private and only meant for bluzelle network. There are two parties in a bluzelle network: ***Producers and Consumers***. The producers are like miners who temporarily provide their computing resources to the network for tokens (BLZ or BNT). This producer economy is called a swarm of nodes or meta swarm.



A swarm is a collection of nodes that store the same set of data. Bluzelle network consists of many swarms that can store different sets of data. The consumers are dApps who require an immutable platform that provides decentralized storage. They access the bluzelle resources and pay in tokens (BLZ and BNT). The two tokens can be exchanged in accordance to a 1:1 ratio. A unique index system monitors the network, imposing penalty on the producers that break rules. Producers who perform well are given a chance to become swarm leaders.

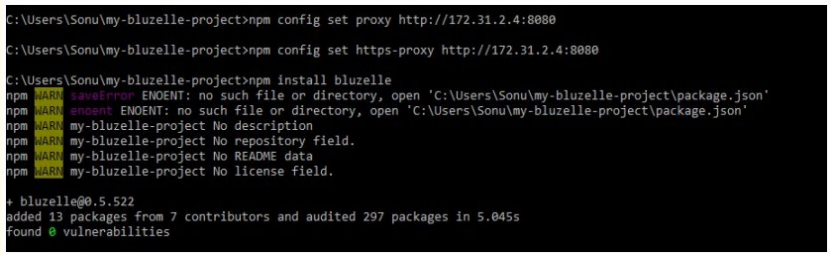
Bluzelle has interconnected servers, each of which replicates the same data, all over the world. The data request of a customer is fulfilled by the server which responds first. Even if a server goes down there are plenty of others to handle the request. Deployment of new servers can be done at any time to replace existing ones or to increase geographic availability.

The singleton metaswarm refers to the entire swarm framework while the virtualized metaswarm is a colony of leaf swarms. Data stored by each leaf swarm is shared with the virtualized metaswarm. The data that network gets from a consumer is split up and stored as shards between each leaf swarm. This ensures data replication and security so that no data loss occurs in the event of a single node failure in the swarm. When a consumer wants to retrieve his data, he provides the private key corresponding to its hash value. The encrypted information on bluzelle ensures that no one can access data without its private key.

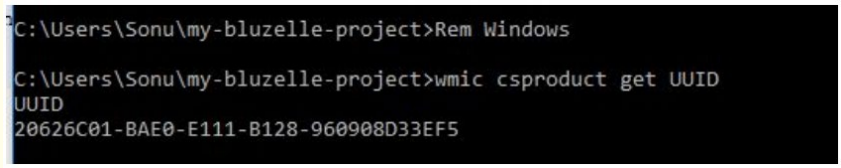
Implementation of Proposed Work

Installation and Start

To get started we first install the Bluzelle JS library by typing *npm install bluzelle* in our terminal.



Then we proceed to type *wmic csproduct get UUID* in the terminal to get our system's UUID as it will be required in the codes further.



Heroku

We can use Heroku with bluzelle add-on which automatically creates environment variables, in our application to enable connection to our testnet and provides a dashboard to manage data that our heroku Application committed to testnet.

First we install the Heroku CLI and log in.

```
$ heroku login
```

Next we deploy an application on heroku and attach the Bluzelle DB add-on to the application as follows:

```
$ heroku addons:create bluzelledb:test -a YOURHEROKUAPP
```

Next in our project root we include:

```
const bluzelle = require('bluzelle');
```

To connect to a server we give the host and port of a Bluzelle node, and supply a unique *uuid* for our database.

The following is our Bluzelle JavaScript API:

```
const bluzelle = require('bluzelle');

bluzelle.connect('ws://testnet.bluzelle.com:51010',

'20626C01-BAE0-E111-B128-960908D33EF5');

bluzelle.create('myKey', 'myValue').then(() =>
{
    bluzelle.read('myKey').then(value =>
    {
        console.log(value); // 'myValue'
    }).catch(e => console.log(e.message));
}).catch(e => console.log(e.message));
```

Then we use the three configuration variables in the code of our previously deployed heroku application to connect to the Bluzelle database.

```
$ process.env.BLUZELLEDB_ADRESS (ex. ws://testnet.bluzelle.com)
$ process.env.BLUZELLEDB_PORT (ex. 51010)
$ process.env.BLUZELLEDB_UUID (ex. d5ffc87e-b447-43cd-980e-53feed3b1afe)
```

After that we locally replicate configuration variables.

```
$ heroku config:get ADDON-CONFIG-NAME -s  >> .env
```

Then we use our configuration variables in our application by replacing the line we use for connecting to testnet i.e.

```
bluzelle.connect('ws://testnet.bluzelle.com:51010','45498479–2447–47a6–8c36-efa5d251a283');
```

With environment variables that were set by the add-on:

```
bluzelle.connect(process.env.BLUZELLEDB_ADDRESS +':'+ process.env.BLUZELLEDB_PORT,process.env.BLUZELLEDB_UUID);
```

For Node.js applications we add an entry into the package.json file as follows:

```
npm install --save bluzelle
```

```
var BluzelleClient = require('bluzelle');

const bluzelle = new BluzelleClient(
       process.env.BLUZELLEDB_ADDRESS + ':' + process.env.BLUZELLEDB_PORT,
       process.env.BLUZELLEDB_UUID
    );

await bluzelle.connect();
```

Cryptographic Keys

We need to provide our own private key to use the database. We install OpenSSL and generate a new key by typing the following command:

```
openssl ecparam -name secp256k1 -genkey -noout
```

We receive the following output:

The private key is entered into the *private_pem* field of the bluzelle constructor. *bluzelle-js* signs off database operations using this key. The decentralized swarm then verifies our signatures to enforce security and authorization. Bluzelle uses the Elliptic Curve Digital Signature Algorithm (*ECDSA*) on the curve *secp256k1* with an *SHA-256* hash which are all built-in to the system.

Database Permissions

Each database is created by one owner who is the only one who can add or remove writers. Initially the number of writers is zero by default. Any user other than a writer trying to modify the database is denied access.

In ECDSA cryptography, the private key is used to generate the public key. The API function of *bluzelle-js* gives user's the public key. Alternately we can get the public key in this way:

```
openssl ec -in private.pem -pubout
```

Where the file *private.pem* contains the private key.

Caching Code (Setup, Read and Write)

Whenever a client wants to read data he will first try to read a key-value-pair (KVP) from Bluzelle's cache. If the KVP is not in Bluzelle's cache then it is a miss as a result the client will request the data directly from the database.

Firstly we initialize the bluzelle client in our code:

```
const bz = bluzelle({

  entry: 'ws://testnet.bluzelle.com:51010',

  uuid: '20626C01-BAE0-E111-B128-960908D33EF5',

  private_pem:
'MHQCAQEEIFNmJHEiGpgITlRwao/CDki4OS7BYeI7nyz+CM8NW3xToAcGBSuBBAAKoU
QDQgAEndHOcS6bE1P9xjS/U+SM2a1GbQpPuH9sWNWtNYxZr0JcF+sCS2zsD+xlCcbrRX
DZtfeDmgD9tHdWhcZKly8ejQ=='

});
```

We do the following to read a record from the Bluzelle cache and fall back to our existing database:

```
try {
  return await bz.read(key);
}
catch(e)
{
  const value = await db.get(key);
  await bz.update(key, value);
  return value;
}
```

For writing records we write to the cache and to the database:

```
await db.put(key, value)
await bz.update(key, value)
```

Experiments and Results

Integrating Bluzelle Cache

Suppose I have an online game whose database is in India. If suddenly the game takes off in England then I will have to setup another server and replicate everything. Then if the game takes off in China then I have to do the same thing all over again. This incurs overhead in terms of time as well as money. But with bluzelle there is no need to switch out, replace or remove. It is added as an enhancement to the existing code. There is no need to redesign the app. In fact bluzelle can be integrated into any app in under five minutes.

Check out the *client.js* file below:

```
        const temp = appCommandQueue;
        appCommandQueue = [];
        return temp;
    },
    clearCache: async () => sendToServer({cmd: 'clearCache'}),
    uploadImage: async ({id, image}) => sendToServer({cmd: 'uploadImage', id, image}
    requestData: async ({withCache, id, numOfRequests, startRequestId}) => {
        times(numOfRequests, async (idx) => {
            let profile;
            let fromCache = false;

            const start = Date.now();

            // With Bluzelle
            withCache && (profile = await getBZConnection[idx].quickread(id).catch(() => undefined));

            fromCache = !!profile;

            // Without Bluzelle
            profile || (profile = await sendToServer({cmd: 'requestData', id, withCache}));

            appCommandQueue.push({
                cmd: 'updateRequest',
                data: {
                    requestId: startRequestId + idx,
                    delay: Date.now() - start,
                    fromCache,
                    profile: typeof profile === 'string' ? JSON.parse(profile) : profile
```

In the highlighted portion we connect to Bluzelle and quickread a key-value pair that is identified by the key *id* and set the return value to *profile*. So what we are doing here is what is called a cache aside pattern. I am first asking the cache "hey do you have this value?". If the cache has that value it'll quickly return that value, if it doesn't have that value which is called a cache miss it won't return that value. So hopefully we get a cache hit and *profile* has the object.

Now if *profile* does not have it then it is a cache miss. We next go to this line:

client.js

```
25          const temp = appCommandQueue;
26          appCommandQueue = [];
27          return temp;
28      },
29      clearCache: async () => sendToServer({cmd: 'clearCache'}),
30      uploadImage: async ({id, image}) => sendToServer({cmd: 'uploadImage', id, image})
31      requestData: async ({withCache, id, numOfRequests, startRequestId}) => {
32          times(numOfRequests, async (idx) => {
33              let profile;
34              let fromCache = false;
35
36              const start = Date.now();
37
38              // With Bluzelle
39              withCache && (profile = await getBZConnection(idx).quickread(id).catch(() => undefined));
40
41              fromCache = !!profile;
42
43              // Without Bluzelle
44              profile || (profile = await sendToServer({cmd: 'requestData', id, withCache}));
45
46              appCommandQueue.push({
47                  cmd: 'updateRequest',
48                  data: {
49                      requestId: startRequestId + idx,
50                      delay: Date.now() - start,
51                      fromCache,
52                      profile: typeof profile === 'string' ? JSON.parse(profile) : profile
```

The highlighted line represents what our client will typically already have or what we might have already programmed into it where we are basically talking to our existing back-end which is probably an existing database server which might be SQL or some sort of middleware but the point is this is what we already have and all we are doing is adding the single line of code(*line 39*) above it saying ask the bluzelle cache first for that value and see if we can quickly get that data back from those out at the edge before we even bother talking to our middleware.

So the benefit here is pretty obvious. We can get a really quick response to getting our data instead of waiting to talk to our server on the back-end. Instead of burdening our server with all the additional overhead of talking to it every time we need data, we just get it from the cache.

Let us keep in mind that sendToServer (*line 44* of *client.js*) is actually calling ultimately something called request data which is on the server side i.e. *server.js*. Using an existing server side cache called Redis which is wrapped around our existing database we are reading our data here from an SQL server or a Mongo server:

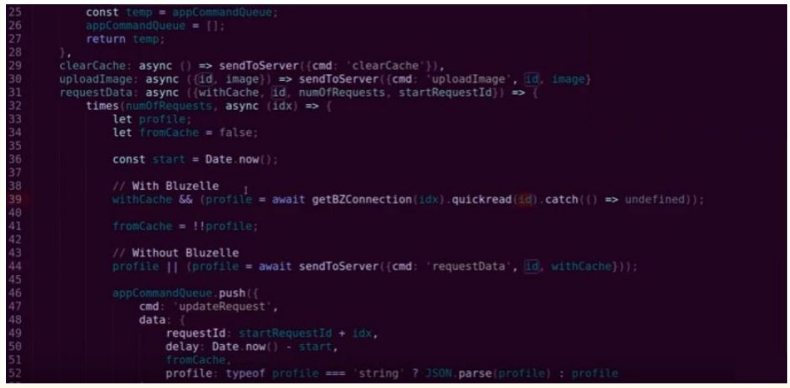
server.js

The highlighted portion of code above represents our existing middleware. Now we ultimately do not want to hit this very often because readFromSQLServer (*line 44*) is slow, redisGet (*line 41*) is slow and there is also latency of talking to the server. But ultimately whenever we do hit this function we want to add the line highlighted below:

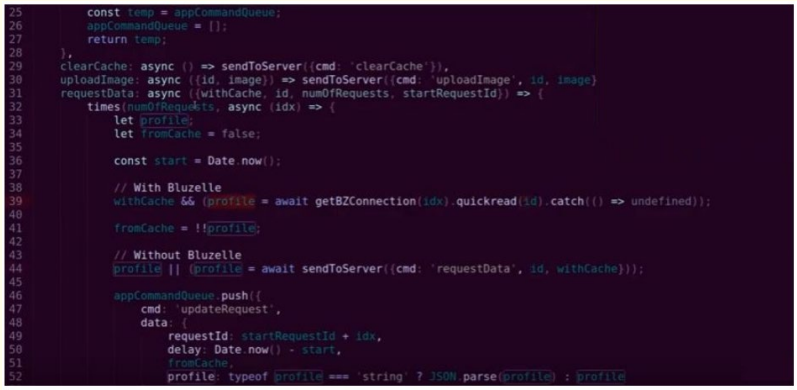
What this line does is it basically tells bluzelle "here is this key-value pair, just store it". Within seconds of calling this function bluzelleset(id, profile) the entire bluzelle network is going to have this id and profile value.

It is stored in the cache and that means within seconds any client that asks for that id value here:

client.js



Is going to get that profile object:



Really quickly. The cache will have been preloaded with that value and now the cache is active for that object.

So for adding data to the bluzelle cache we add this one line to our *server.js* code:

And in the *client.js* we add this one line to read:

So we are not changing any existing code, not removing references to our existing database, not changing references to reading from the existing database. Everything is the same just one extra line of code for bluzelle.

Creating Node.js Application with Bluzelle Database

Firstly we install bluzelle library in our system by typing in *npm install bluzelle*. Then we type in the following code in our editor:

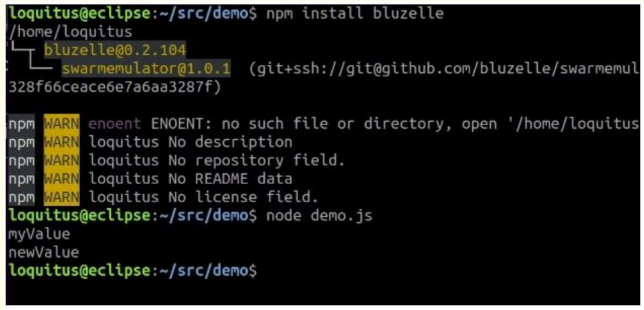
```
1  const bluzelle = require('bluzelle');
2  bluzelle.connect('ws://testnet.bluzelle.com:51010',
3  'javascript_demo');
4  bluzelle.create('myKey', 'myValue').then(() =>
5  {
6       bluzelle.read('myKey').then(value =>
7       {
8            console.log(value); // 'myValue'
9
10           bluzelle.update('myKey', 'newValue').then(() =>
11           {
12                bluzelle.read('myKey').then(value =>
13                {
14                     console.log(value); // 'myValue'
15                }).catch(e => console.log(e.message));
16           }).catch(e => console.log(e.message));
17      }).catch(e => console.log(e.message));
18 }).catch(e => console.log(e.message));
```

In line 1 we use the require command to import the bluzelle library. In line 2 we connect to testnet.bluzelle.com which is a bluzelle lovelace testnet and provide a namespace variable javascript_demo. We then create a key-value pair with the value of the key myKey and value myValue. We then read back "my key" we should get back "my value" and we will print that out. We then update the key-value pair with key myValue to newValue. We then read it again and display it on the screen. Now, if everything works as planned we should first see the value myValue printed on the screen followed by newValue. We save the code as *demo.js*.

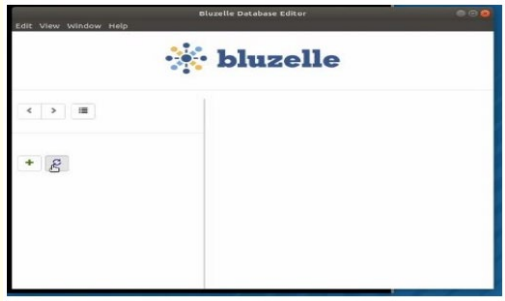
We go to a CRUD client and validate the namespace javascript_demo and see that those key value pairs are not already there.

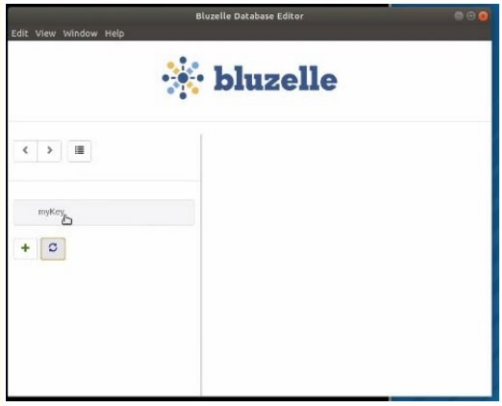
So we find an empty database.

Then we run the code by typing *node demo.js in our terminal*:

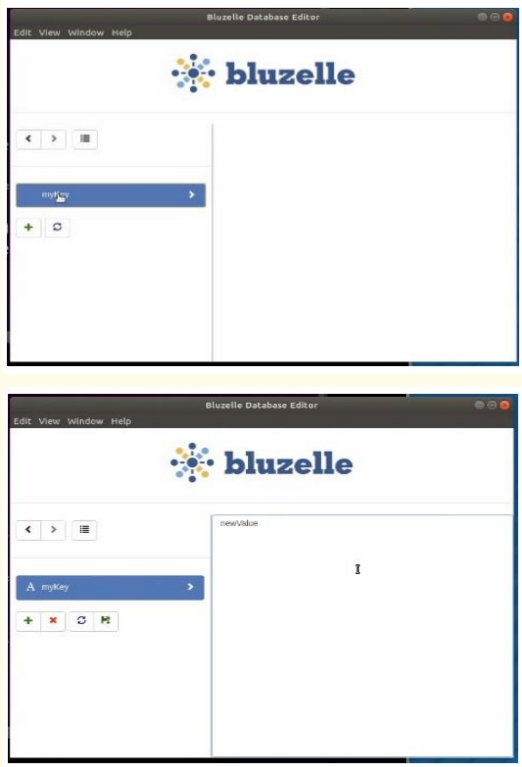


So we see that we got the desired output myValue followed by newValue. Then we go to the CRUD client to verify what appears.





So we have myKey as expected and then we click on it to check its value.



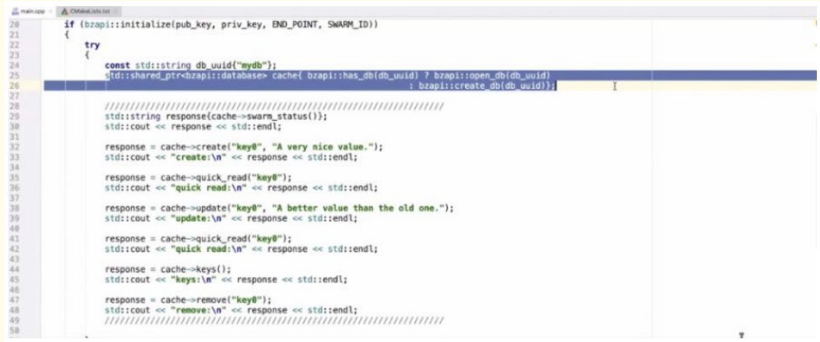
And as expected it has the last value was set to which was newValue.

CRUD Operations using Bluzelle APIs

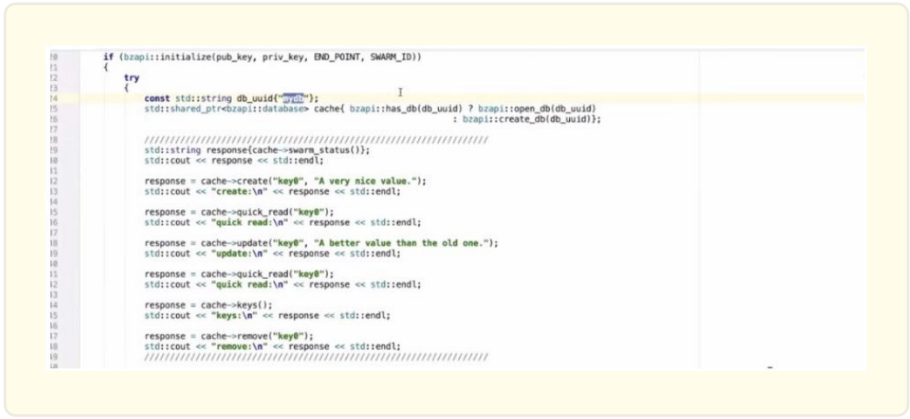
Let us take a look at an application which will spool up a four node swarm on my machine. It will be a local swarm and the application called swarm spooler. We can see that it has created a four nodes swarm and these four nodes are running a locally on our machine.

Now we go to the swarm talker. It uses the Bluzelle API to talk to the swarm I have just created. First thing is I need to connect to the swarm using the Bluzelle API initialize function which takes a public and private key for the cryptography part of creating and reading keys. It also takes an endpoint so that it knows where the swarm is and a swarm ID. The endpoint is my local host address and the port of one of the nodes in the swarm.

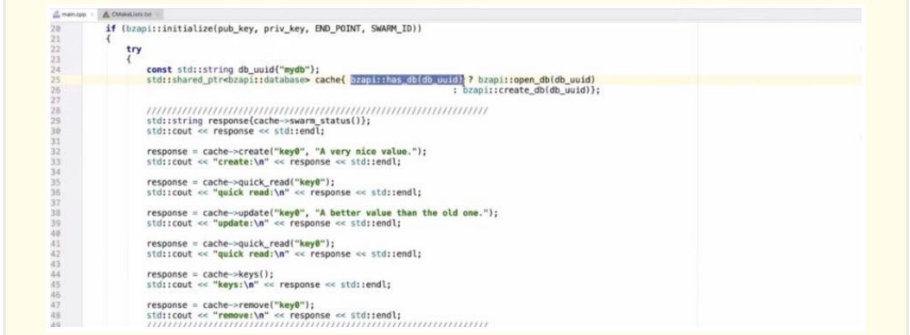
It will ask that one node for a list of the rest of the nodes in the swarm and then it will ask each of those swarms for a status and choose the fastest node to connect to from then on. Every now and then it pulls the swarm to find out which one is fastest and always tries to use the fastest node. Once we've initialized that connection we can create or open a cache and that's what this line here is for:



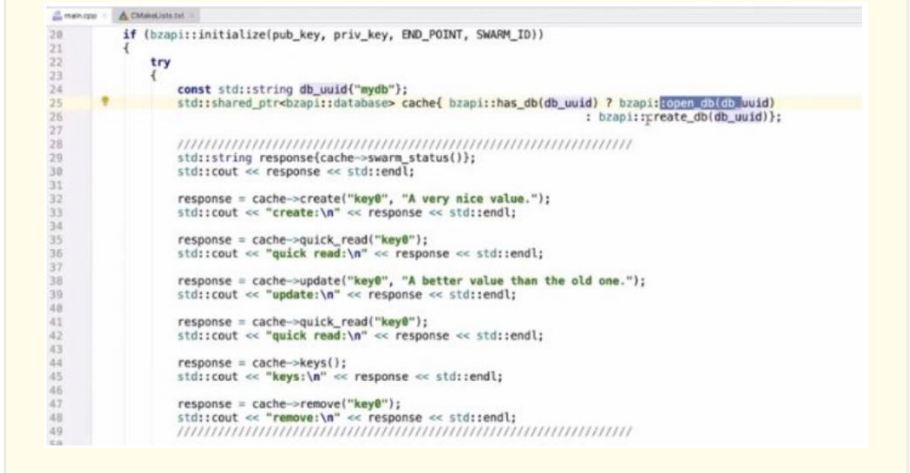
So I have chosen myDB as the identifier for my cache.



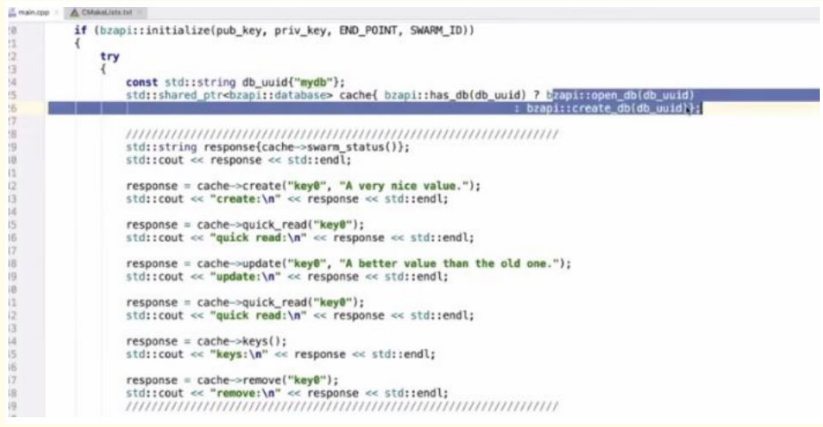
We will first ask the swarm does it have that cache.



If it does we will open the cache. If it does not it will create the cache.



Return value of these two functions is a database object:

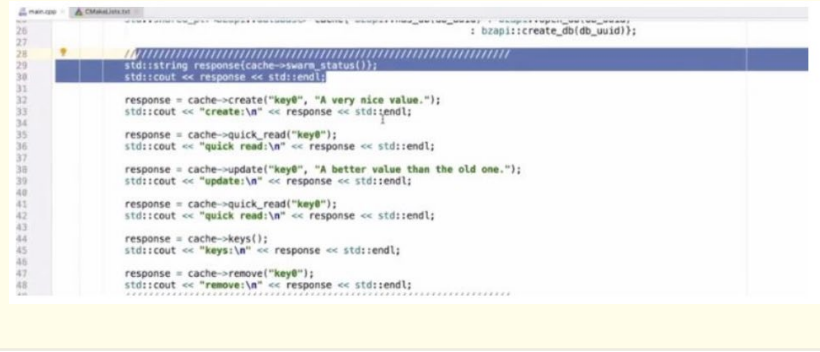


```
if (bzapi::initialize(pub_key, priv_key, END_POINT, SWARM_ID))
{
    try
    {
        const std::string db_uuid{"mydb"};
        std::shared_ptr<bzapi::database> cache{ bzapi::has_db(db_uuid) ? bzapi::open_db(db_uuid)
                                                                       : bzapi::create_db(db_uuid)};

        ////////////////////////////////////////////////////////////////////
        std::string response{cache->swarm_status()};
        std::cout << response << std::endl;

        response = cache->create("key0", "A very nice value.");
        std::cout << "create:\n" << response << std::endl;

        response = cache->quick_read("key0");
        std::cout << "quick read:\n" << response << std::endl;

        response = cache->update("key0", "A better value than the old one.");
        std::cout << "update:\n" << response << std::endl;

        response = cache->quick_read("key0");
        std::cout << "quick read:\n" << response << std::endl;

        response = cache->keys();
        std::cout << "keys:\n" << response << std::endl;

        response = cache->remove("key0");
        std::cout << "remove:\n" << response << std::endl;
        ////////////////////////////////////////////////////////////////////
```

Then we run this and get the following output:

```
[2019-05-30 11:58:44.108681] [0x000070000fb05000] [debug]   (db_impl.cpp:175) - Got response for message 5
[2019-05-30 11:58:44.108692] [0x000070000fb05000] [debug]   (db_impl.cpp:180) - Ignoring db response for unknown or already processed m
348 bytes written
[2019-05-30 11:58:44.121517] [0x000070000fb05000] [debug]   (db_impl.cpp:175) - Got response for message 6
[2019-05-30 11:58:44.121551] [0x000070000fb05000] [debug]   (db_impl.cpp:210) - 1 of 3 responses received
[2019-05-30 11:58:44.122566] [0x000070000fb05000] [debug]   (db_impl.cpp:175) - Got response for message 6
[2019-05-30 11:58:44.122591] [0x000070000fb05000] [debug]   (db_impl.cpp:210) - 2 of 3 responses received
[2019-05-30 11:58:44.123601] [0x000070000fb05000] [debug]   (db_impl.cpp:175) - Got response for message 6
[2019-05-30 11:58:44.123631] [0x000070000fb05000] [debug]   (db_impl.cpp:210) - 3 of 3 responses received
[2019-05-30 11:58:44.123675] [0x000070000fb05000] [debug]   (db_impl.cpp:199) - Done processing db response for message 6
remove:
{
    "result" : 1
}

[2019-05-30 11:58:44.123746] [0x000070000fb05000] [debug]   (library.cpp:61) - Events run: 77
```

For the demo the first thing I will do once I have a cache, is ask for the swarm status and print that out:

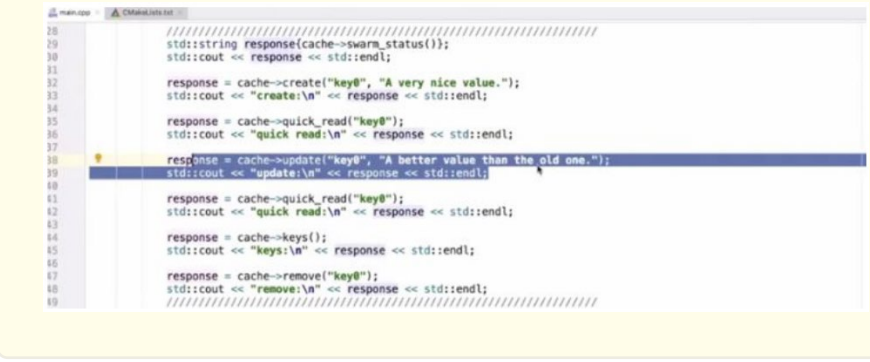


```
                                                                       : bzapi::create_db(db_uuid));

        ////////////////////////////////////////////////////////////////////
        std::string response{cache->swarm_status()};
        std::cout << response << std::endl;

        response = cache->create("key0", "A very nice value.");
        std::cout << "create:\n" << response << std::endl;

        response = cache->quick_read("key0");
        std::cout << "quick read:\n" << response << std::endl;

        response = cache->update("key0", "A better value than the old one.");
        std::cout << "update:\n" << response << std::endl;

        response = cache->quick_read("key0");
        std::cout << "quick read:\n" << response << std::endl;

        response = cache->keys();
        std::cout << "keys:\n" << response << std::endl;

        response = cache->remove("key0");
        std::cout << "remove:\n" << response << std::endl;
```

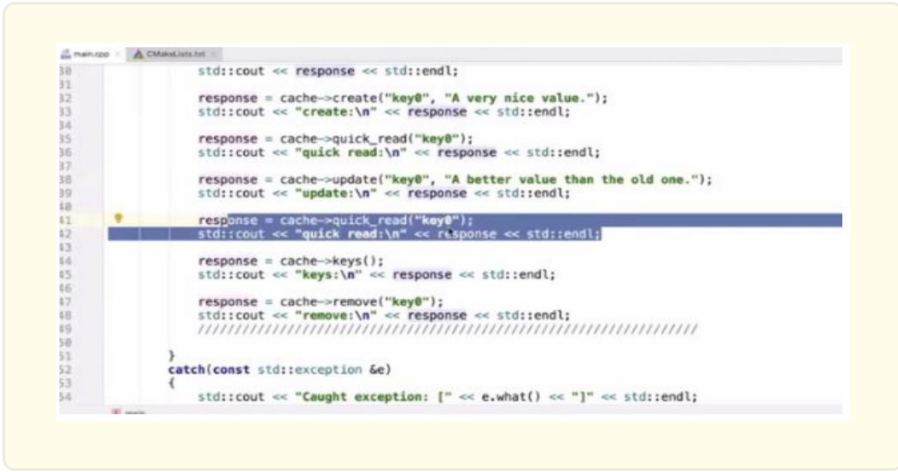
Then I'll create a key value pair and key0, A very nice value:

Then I'll quick read that key0:

Then I'll update the value of key0:

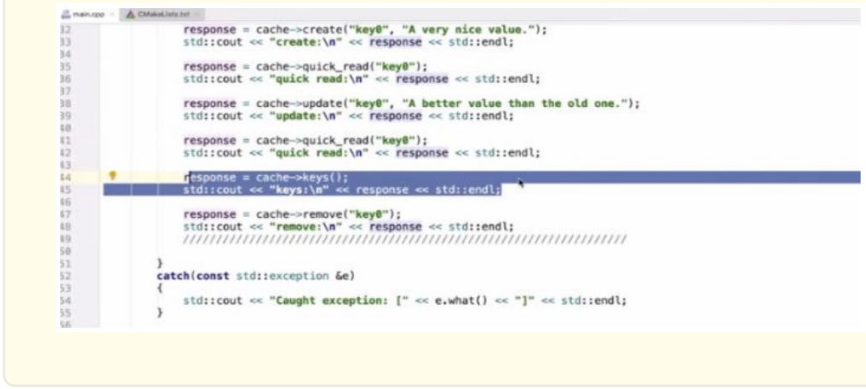


Read it again to be convinced that the value has been updated:



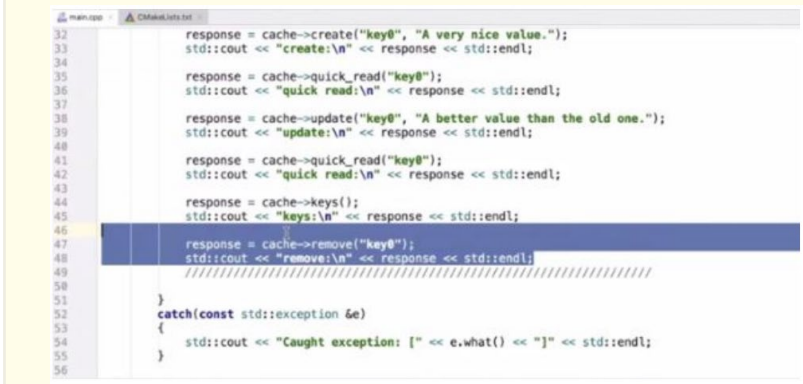
```
        std::cout << response << std::endl;

        response = cache->create("key0", "A very nice value.");
        std::cout << "create:\n" << response << std::endl;

        response = cache->quick_read("key0");
        std::cout << "quick read:\n" << response << std::endl;

        response = cache->update("key0", "A better value than the old one.");
        std::cout << "update:\n" << response << std::endl;

        response = cache->quick_read("key0");
        std::cout << "quick read:\n" << response << std::endl;

        response = cache->keys();
        std::cout << "keys:\n" << response << std::endl;

        response = cache->remove("key0");
        std::cout << "remove:\n" << response << std::endl;
        ////////////////////////////////////////////////////////////////////////

    }
    catch(const std::exception &e)
    {
        std::cout << "Caught exception: [" << e.what() << "]" << std::endl;
```

And then I'll ask for all the keys in the cache:



```
        response = cache->create("key0", "A very nice value.");
        std::cout << "create:\n" << response << std::endl;

        response = cache->quick_read("key0");
        std::cout << "quick read:\n" << response << std::endl;

        response = cache->update("key0", "A better value than the old one.");
        std::cout << "update:\n" << response << std::endl;

        response = cache->quick_read("key0");
        std::cout << "quick read:\n" << response << std::endl;

        response = cache->keys();
        std::cout << "keys:\n" << response << std::endl;

        response = cache->remove("key0");
        std::cout << "remove:\n" << response << std::endl;
        ////////////////////////////////////////////////////////////////////////

    }
    catch(const std::exception &e)
    {
        std::cout << "Caught exception: [" << e.what() << "]" << std::endl;
    }
```

Finally I will remove the key0 from the cache to clean up:



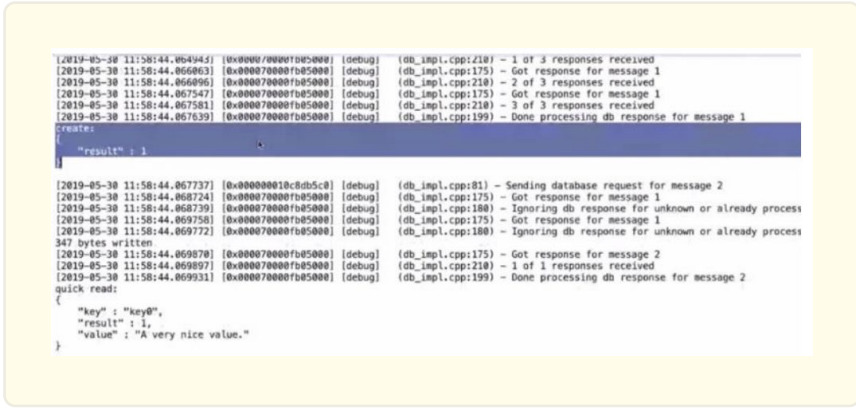
```
        response = cache->create("key0", "A very nice value.");
        std::cout << "create:\n" << response << std::endl;

        response = cache->quick_read("key0");
        std::cout << "quick read:\n" << response << std::endl;

        response = cache->update("key0", "A better value than the old one.");
        std::cout << "update:\n" << response << std::endl;

        response = cache->quick_read("key0");
        std::cout << "quick read:\n" << response << std::endl;

        response = cache->keys();
        std::cout << "keys:\n" << response << std::endl;

        response = cache->remove("key0");
        std::cout << "remove:\n" << response << std::endl;
        ////////////////////////////////////////////////////////////////////////

    }
    catch(const std::exception &e)
    {
        std::cout << "Caught exception: [" << e.what() << "]" << std::endl;
    }
```

And if we look at the output you can see what happened. That is our status output which lists all of the nodes and their connection information:

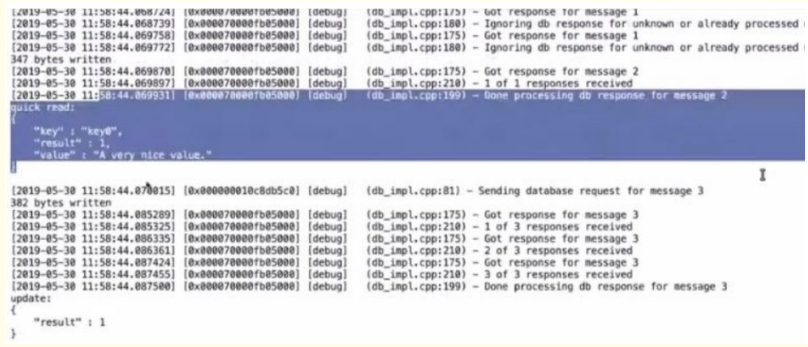
And the time it took to talk to them:

It also gives us the value of the primary node:

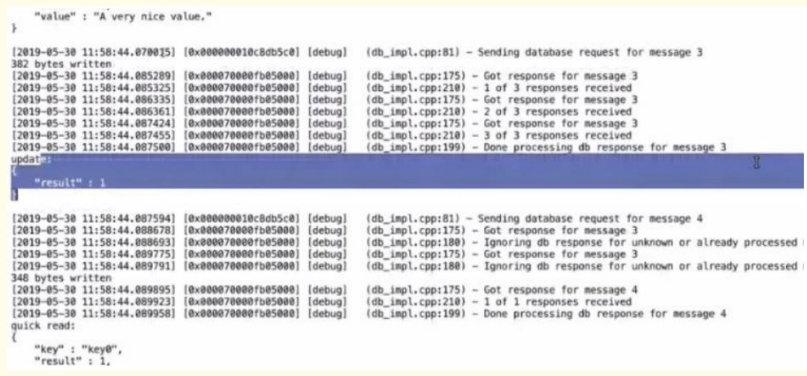
Here is the create response we see that it was successful:



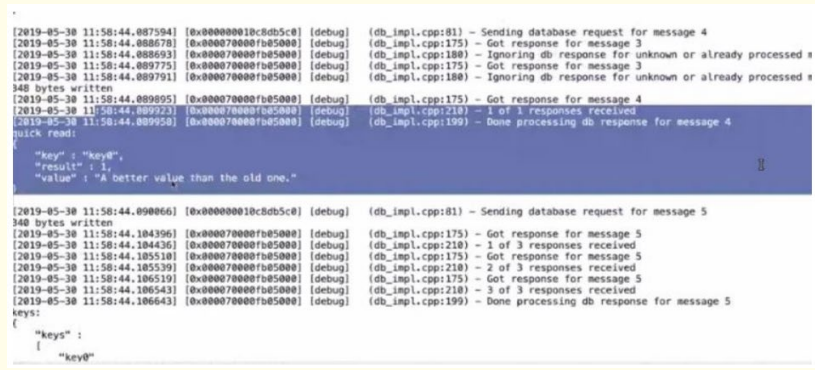
If it were not successful, it would also give us an error message, for example, the key already exists. Here's the result of the quick read. So I asked for key0 and it gave me the value of a very nice value:



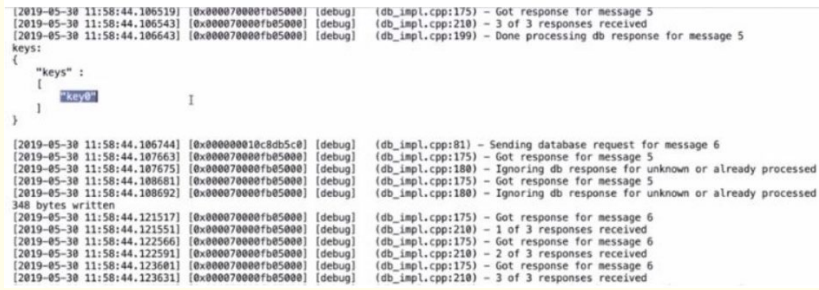
I then did the update or I changed a very nice value for key0 to something else:



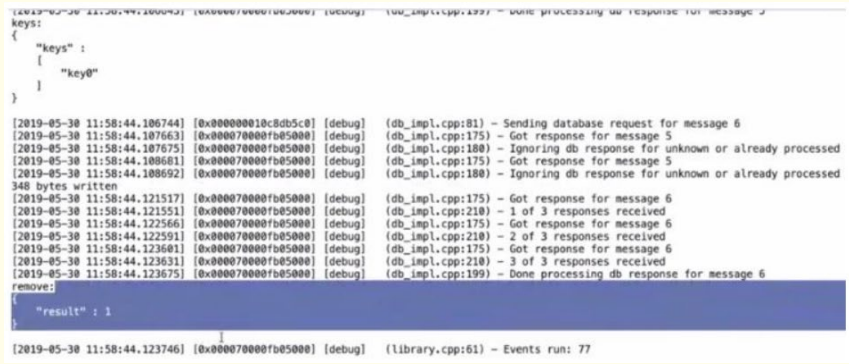
And here it is a better value than the old one:



Finally I asked the swarm for all the keys in our cache. There is only one so it returned an array of one key i.e. key0:



And finally I removed the key to clean up:



Future Work

Bluzelle is not the only player in this game. It has competitors such as MaidSafeCoin, Siacoin, Storj and Filecoin. It is a certainty that one of these come out on top in the near future and investors in the BLZ token are betting that when the dust settles Bluzelle will be the last man standing.

Bluzelle has made sure that it is well-positioned to dominate the market by establishing offices in the west in Vancouver, and in the east in Singapore. It has also made alliances to help it capture market share and become the dominant decentralized database storage solution.

Currently most of its features are in beta phase while some are being scrutinised before being made available. When fully operational its storage capacity will be ten million times that of current corporate IT storage at 0.001 the cost per Tb of storage when compared with current costs. To put it in a nutshell, bluzelle promises all round data security.

Conclusion

Bluzelle is very easy to use and involves writing minimal code to cache our data to it. It has servers available in many data centers across multiple region thereby providing low latencies and high throughputs to applications. Tiering ensures that Bluzelle Cache can match our I/O performance needs. Data within Bluzelle Cache is globally replicated across more than 25 global regions. With its client libraries, one can easily and in a scalable manner store his/her data across bluzelle's servers. The user's data is always available, irrespective of faults, disasters, or attack vectors. There is no overhead for infrastructure deployment as there is nothing new to deploy. Customers only pay for the services they use. Customers are able to use Bluzelle as their primary data store or as a replica-set, depending on how they use bluzelle APIs in their codebase. Bluzelle provides all the coding patterns from which customers can choose. Also there are provisions for quickly adding additional servers to new data centers. Bluzelle uses both symmetric and asymmetric key cryptography to ensure that data can only be modified by its authorized owners/writers. No organization, including Bluzelle, has any mechanism to remove or modify data stored on Bluzelle. This is a level of security that simply cannot exist on a centralized platform including the cloud. It also means that data stored on Bluzelle belongs to and can only be affected by the owner of that data and no one else. Only they can remove it. Furthermore, Bluzelle will also be offering optional symmetric key encryption for data in transit and at rest, to ensure that only authorized parties can read the data.

References

1. Shubham Sharma., et al. "Detecting Insider Attacks on Databases using Blockchains". In proceedings of Workshop on Blockchain Technologies and its Applications, ISRDC, IIT Bombay (2017).
2. Weizhi Meng., et al. "When Intrusion Detection Meets Blockchain Technology: A Review". IEEE Access 6 (2018): 10179-10188.
3. Deepak K Tosh., et al. "Security Implications of Blockchain Cloud with Analysis of Block Withholding Attack". 2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (2017).
4. Tzvi Chumash and Danfeng Yao. "Detection and prevention of insider threats in database driven web services". IFIPTM 2009: Trust Management III (2009): 117-132.