Designing and Implementing Multi-user Distributed Text Editor

# Introduction

It is required to design and implement a multi-user distributed text editor, which will allow several users to collaborate in reading and editing text documents in a real-time environment where changes at any user will be broadcasted to all the other users at the same exact time of the changes. It is more like we need to design and implement a clone of Google Documents application.

# Project Description

This project describe to design and implement the multi-user distributed text editor using different components to collaborate in reading and editing text documents. We firstly made the client interface code that will appear to the user using quill and web sockets. Meanwhile, we implemented different servers to handle the clients’ requests to edit or insert data. The servers are connected together with mongo Database. The implementation of the database required several database, as the First one is the primary and there are 2 secondary databases, all of them are connected together to the have the place of the primary if the primary fails.

Also we made a load balancer which will be replaced between the client and server which refers to the act of distributing network traffic across multiple services.

This makes sure that there's not too much load on a single server which could cause it to crash.

A load balancer acts as a ‘reverse-proxy’ to represent the application servers to the client through a virtual IP address (VIP).

# Beneficiaries of the project:

This Multi-User Distributed text editor system can be used in various fields and aspects, and for various users with different backgrounds either technical or non-technical.

The benefits of this system is shown in offering a user-friendly yet reliable interface for Shared Document/s between multiple users or editors. The concept of using shared documents between users is important as it reduces the time consumed in merging different versions among the multiple users enrolled in editing the same document, it also reduces the conflicts resulting from the clashes between different versions of the document.

Simply, it is better for all the enrolled users for editing a specific type of document to edit one version of a document with up to date edits of the other users all saved in database.

**Beneficiaries of the project** may appear clearly In any Organization of documenting important information which needs to be frequently updated by multiple users in various timings around different places (remotely or not), the system will be a perfect tool for handling all the requirements needed to maintain the functionalities required in the organization.

# Roles of the team

|  |  |
| --- | --- |
| Name | Role |
| Eman Khaled | * Part of client Code * Project Documentation * Redis Publish Subscribe Implementation |
| Farah Amr Abd El Fattah | * Part of client Code * Project Documentation * Database Implementation |
| Shehab Eldin Adel | * Part of Server Code * Implementation Of Load Balancer * Redis Publish Subscribe Implementation * Database Replication Database Implementation |
| Zyad Ahmed Yakan | * Part of Server Code * Implementation Of Load Balancer * Database Implementation and Database Replication |

# Task Breakdown structure

## Frontend

Implementation of Client Code by using Quill and web sockets offering an interface for the user to be able to write on the document by different tools and options .This is a pure example of and implement a clone of Google Documents application.

## Backend

Implementation of Server Code by using web sockets and …………….

## Redis Publish Subscribe

We implemented the redis as Pub/sub messaging can be used to enable [event-driven architectures](https://aws.amazon.com/event-driven-architecture/).

## Database

This is connected to the servers upon request processed from the client side to the server side so that each and every change will be inserted and saved on the database side.

## Data base Replication

We implemented the database replication using mongodb replica dataset , as when the primary database is failed to work there are other secondary database that will overwrite on it and work instead on it and become the new primary database. This replication dataset was implemented so that we can achieve availability and concurrency.

## Load balancer

Load balancing refers to the act of distributing network traffic across multiple services.

This makes sure that there's not too much load on a single server which could cause it to crash.

A load balancer acts as a ‘reverse-proxy’ to represent the application servers to the client through a virtual IP address (VIP).

# System Architecture:

Figure 1: Multi-User Distributed Text Editor System Architecture

# System Design

There are several functional and non-functional requirements that we must follow in order to start our system design plan.

## Functional Requirements

* Users can change a document at the same time without any conflict
* Allow sharing documents between users through a unique document ID and hyper-link

## Non-Functional Requirements

* The system must support multiple clients or autonomous agents like an API for sharing and updating data.
* The system should be distributed across multiple clients or server nodes.
* The system should be robust
* The system should be able to continue operation even if one of the participant nodes crashes
  + If three clients are collaborating on the same document and one client failed, the other two should continue collaboration on the system achieving reliability.
* It should be possible to recover the state of the node following a crash, to continue operation
  + Retrieving the final state of the document when the node goes back online
* The system should maintain multiple replicas for fault tolerance.

## Design Constraints

* Concurrency
  + Since several users are working on the same document
  + Operational Transformation
* Latency
  + Clients are working in different places, and the connection is established through the internet, so there is a latency between each and all clients when they are collaborating on the same document.
  + RESTful vs Publish/Subscribe Architectures.
* Security
* For each type of user, what type of data access restrictions are required.
* For each type of user, what type of update privileges are required.
* For each type of user are there any other Window behaviors which require specific privilege.

* Backup and Recovery Requirements
* Acceptable down time for system.
* Acceptable data and user interface state loss due to system crash*.*

## Operational Transformation

In order to provide real-time and collaborative environment in a text editor, we must consider any conflicts that may arise when more than two nodes are collaborating. For example, if a node inserted some text at position x, and another node deleted the text that exists at position x at the same time. Here, we present Operational Transformation, which is a technology that aims to solve conflicts in real-time collaborative editing environments. In order to do that, we must maintain consistency between local replicas of documents, since each client have its own local copy of the document.

A document will be stored as a sequence of operations in order of execution instead of plain text. So, we need a collaboration protocol to understand when to apply changes. We thought about identifying possible operations into three types:

* Insert Text
* Delete Text

Graphical user interface, application

Description automatically generatedWhenever we edit a document, all the changes are appended to the document saving these operations in one of those three types. In addition to saving operations by each user in a changelog database.

Figure 2 - Operational Transformation Sequence diagram

## WebSocket vs HTTP

1. Web Socket

* WebSocket is a technology that enables bidirectional, full-duplex communication between client and server over a persistent, single-socket connection. This allows for low-latency, real-time updates, and the creation of richer communication and gaming applications. Previously, the web was dependent on requests and responses, which aren’t dynamic enough for those kinds of apps.
* WebSockets generally do not use XMLHttpRequest, and as such, headers are not sent every-time we need to get more information from the server. This, in turn, reduces the expensive data loads being sent to the server.
* WebSocket is an event-driven protocol, which means you can actually use it for truly real-time communication. Unlike HTTP, where you have to constantly request updates, with websocket, updates are sent immediately when they are available.

1. Web Transport

* The newer WebTransport offers secure, multiplexed, realtime transport and already has APIs for sending data both reliably and unreliably. In a reliable data transfer, the sender is notified of the success or failure of the data transmission, and failed transmissions are usually resent until they succeed, after which the next data packet is sent. In unreliable transfer, there’s no confirmation of transmission success, and packets that aren’t received simply don’t get delivered.
* Unreliable transfer is often used for things like streaming videos, where speed is a concern, and minor data loss, such as a few frames of video, is acceptable. Because WebTransport uses both of these methods, there are many use cases for it, such as bidirectional data streaming for multiplayer gaming, interactive live streams, and data transfer for sensors and internet of things devices.
* It avoids the head-of-line blocking delays that WebSocket suffers from, and is less resource intensive when creating connections
* It works with HTTP/3, the upcoming version of the transport protocol used by the World Wide Web. HTTP/3 uses the QUIC protocol for transport layer data exchange, which has several advantages. QUIC can prevent head-of-line blocking delays, improving network performance in many situations. This is a limitation of WebSocket.

# RESTful vs Event-Driven Architectures

# General architecture for collaborative editing applications

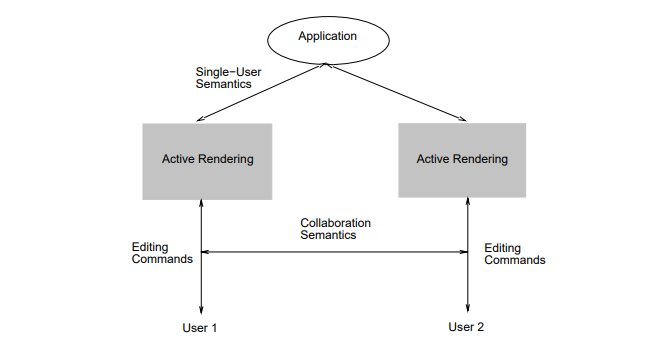


Figure 3 – general architecture for collaborative editing apps

# 1st: RESTful Architecture (representational state transfer)

## Characteristics

REST principles are defined by four interface controls, including identifying resources, managing resources through representations, self-descriptive communications, and hypermedia as the engine of the application state.

View the distributed system as a collection of resources, individually managed by components and these resources can be added, removed, retrieved or modified by remote applications while keeping that these resources provide the same interface and are identified by the same naming scheme.

The messages sent to or from a specific service are fully described and after executing an operation at a service that component totally forgets about the caller.

Has greater **stability** because it restrains component performance. so that each component can’t see further than the immediate layer with which it is intermingling.

REST uses less bandwidth, simple and more flexible making it more useful for internet usage.

Uses http operations (GET, POST, PUT, DELETE, UPDATE, PATCH)

## Guiding principles for REST (constraints)

1. Its **layered system** allows generating a more scalable and flexible application. An application has better security due to its layered system, as components in each layer can’t interact outside the successive layer. Also, it balances loads and offers shared caches for stimulating scalability.
2. **Code on demand (optional):**

A REST API definition permits extending client functionality by downloading and implementing coding in the form of applets or scripts. This restructures clients by decreasing the number of features important to be pre-implemented.

This REST principle allows for applets to be communicated through the API used within the application.

1. **Uniform Interface**

By applying the principle of generality to the components interface, we can simplify the overall system architecture and improve the visibility of interactions.

**Constraints for applying the principle of Uniform Interface:**

* Identification of resources
* **Manipulation of resources through representations 🡪** The resources should have uniform representations in the server response. API consumers should use these representations to modify the resources state in the server.
* **Self-descriptive messages**
* **Hypermedia as the engine of application state🡪 The client should have only the initial URI of the application. The client application should dynamically drive all other resources and interactions with the use of hyperlinks.**

1. **Client-Server**

separation of concerns, By separating the user interface concerns (client) from the data storage concerns (server), we improve the **portability** of the user interface across multiple platforms and improve **scalability** by simplifying the server components.

1. **Stateless**

The server cannot take advantage of any previously stored context information on the server. For this reason, the client application must entirely keep the session state

1. **Cacheable**

The cacheable constraint requires that a response should implicitly or explicitly label itself as cacheable or non-cacheable. And If the response is cacheable, the client application gets the right to reuse the response data later for equivalent requests and a specified period.

## How it works

Figure 4 - REST architecture

A request is sent from client to server in the form of web URL as HTTP GET or POST or PUT or DELETE request. After that, a response comes back from server in the form of a resource which can be anything like HTML, XML, Image or JSON. But now JSON is the most popular format being used in Web Services.

## When to use REST?

* + we need a time-bound request/reply interface
  + Convenient support for transactions
  + our API is available to the public
  + the project is small (REST is much simpler to set up and deploy)

# 2nd: Event-Driven architecture

As most of the system design architects used REST architecture as their service communication layer, but further more and more Projects used Event-Driven Architecture due to its pros which we will discuss in this section.

## Event-Driven detailed architectural design.

Figure 5 - event-driven architecture

## Main Components:

Event-Driven *Software* Architecture, describes various logical components and their roles in events generation, transmission, processing, and consumption.

An event-driven architecture mainly consists of four components:

**1**. Event🡪The change in the state of an object that occurs when users take a specific action.

**2**. Event Handler🡪A software routine, which handles the occurrence of an event.

**3**. Event Loop🡪Handles the flow of interaction between an event and the event handler.

**4**. Event Flow Layers🡪The event flow layer is built on three logical layers: Event Producer, Event Consumer, and Event Channel (Event Bus).

Producer🡪Responsible for detecting and generating events.

Consumer🡪 Consumes the events produced by the event producer.

Event Channel🡪Transfers events from the event generator to the event consumer.

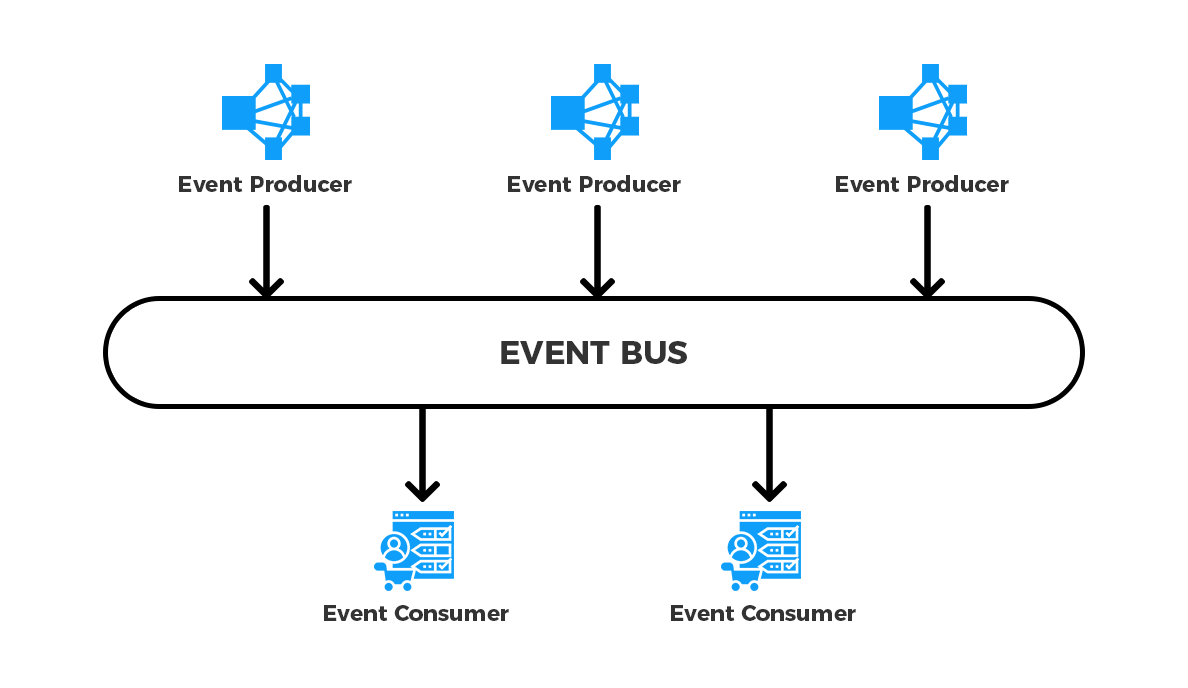


Figure 6 - event channel illustration

## Characteristics

Event-driven distributed systems have two essential characteristics which differentiate them from other system architecture types:

1. The existence of several software/hardware components that run simultaneously on different inter-networked nodes.
2. The use of events as the main vehicle to organize component intercommunication.

## Advantages:

* Adding new events and processes is very easy in the event-driven architecture.
* Event-driven architecture also gives the transactional guarantee, getting notified of every successful transaction that occurs.
* The architecture is easily replaceable.
* We can roll-back changes or move to any event in the event-driven architecture. Which is helpful in case any issue occurs.
* Highly responsive. Instead of waiting for issues to occur, you can easily detect them in advance therefore ensuring that the app keeps working.

## How It works?

We have to define what is an event first, an event is simply *put,* a significant change in state, which is triggered when a user or service takes an action. Other services consume such events in order to complete any tasks that arise as a result of that event.

**Unlike with REST**, services that create requests do not need to know the details of the services consuming the requests.

## Difference from REST architectures

An event-driven architecture offers several advantages over REST, Including:

* **Asynchronous**🡪 Event-based architectures are non-blocking and asynchronous. This permits resources to go on to the next task without worrying about what happened before or what will happen next once their unit of work is completed. They also allow events to be queued or buffered, preventing consumers from applying pressure or blocking to producers.
* **Loose** **coupling**🡪 *services operate independently,* without knowledge of other services, including their implementation details and transport protocol.
* **Easy Scaling**🡪 Due to the decoupling of services in an event-driven design, and the fact that services typically execute only one activity, tracking bottlenecks and scaling that service becomes simple.
* **Recovery Support**🡪 With the queue of Event-Driven Architecture, recovery of lost work by replaying events from the past. This can be Important to prevent data loss when a consumer needs to recover.

## Difference from Request-Response architectures

|  |  |  |
| --- | --- | --- |
|  | Event-Driven | Request-Driven |
| When action is taken? | According to context. | When request is being processed. |
| Why action been taken? | Triggered by situation being detected. | As a response to a specific request. |
| What happens when an event/request occurs? | Event can be ignored, increment the state, trigger an internal derived event or trigger a situation. | Response is always produced. |

## Issues To always Consider

Despite the event-driven architecture offers many benefits, it is very complex to implement and needs a lot of essential concepts to keep in mind when implementing this architecture.

1. **Increasing Complexity**

Since, Event-Driven Systems are *loosely coupled and Highly Distributed,* it’s difficult to tell which event is linked to which microservice and what the relationship between them is and we may never know when a small change can result in an unexpected chain of reactions.

1. **Not much control**

It’s not clearly defined which event should be consumed and which shouldn’t. there are some actions that we may not want to publish, such as a failures or bugs. The issue is how can a microservice know that transaction that we don’t want to publish, This adds an extra layer of confusion.

1. **Certain Events can’t be Undone**

While we can quickly undo modifications in Event-Driven Systems, some events are beyond our control. It's just not possible due to the event's reliance on a third-party system. we can't, for example, undo an already sent email.

## Event-driven Architecture Implementation.

### **1st: Client Code.**

We are using text editor with local host <http://localhost:3001>.

As we are using QUILL as a text editor that has the most powerful text editor that anyone can uses with fully features that any multi user text editors will need. Also it has an important concept called delta which are operations that you can perform to get from one step to the next step as inserting characters in the page that you are writing in it.

So every time someone made a change, the **DELTA operations** will tell you what are the changes that occurs and where are them. So no one can overwrite someone’s else writings as it save their deltas instead of saving the entire document, so this is a very powerful features to be taken into consideration.

Moreover, we are going to use **socket IO** super standard library for WebSocket integrations, which will make it easy to do real time connections between client and server.



Figure 7: Toolbar main function with options

So here is the call toolbar options which is basically an array that consists of different arrays with different sections that will all appear in our local host text editor , example: heading from 1-6 and normal , different fonts, lists, bullet lists, ordered ones, different colors fonts and background ones ,…. etc



Figure 8: Text Editor Function with Useeffect() to connect with socket

So here we are making the main function textEditor and we are implementing quill inside of it and are using **useeffect()** as we want to ever render one time.

So firstly, **useeffect1()** , we are calling the callback io and gives it the url the we are connecting to, by setting the port number to 3001 which is the server URL and will rename it as a socket. On the other side of the code, we are going to put the Client URL that we are going to discuss later.

And to finalize the socket, we are going to write the discount line at the end to make sure that the connection is disconnected after it was connected at the beginning.

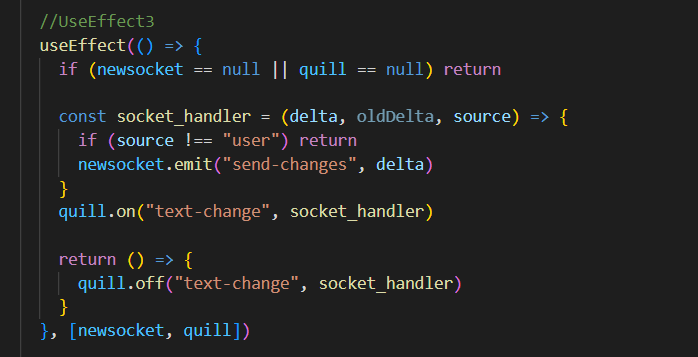


Figure 9: Useeffect() 3 to detect changes when quill changes

Now , the second **useEffect()** will be used to be detecting changes whenever quill changes.

As it said, quill on text change runs this function as it passes us the delta, old delta and the source. The source is going to determine whether the user made these changes or whether the actual kill library made these changes so it’s important to test to user that he made the changes first , else its’s going to return and do nothing. As we only wanted to track the changes that the user made. Then we are going to send these changes by using sockets to the server to be saved and changed on our database by emitting a message from the client to the server and passing it the delta. This delta is very important regardless that it has to sent the only parts changed not the whole document. Then ate the end, we are going to remove the event listener that is no longer needed.

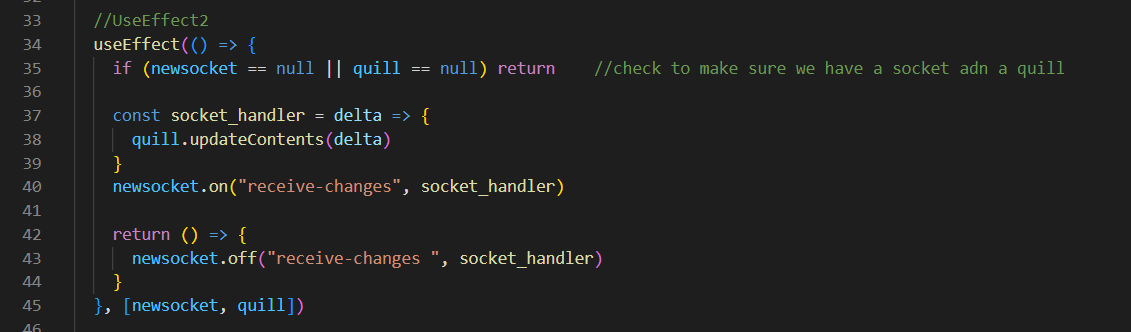


Figure 10: using useEffect() to receive changes

Furthermore, the function of using the ***useEffect()*** is used another time #2 to do the operation of receive changes. Where we use the socket operation ***on()*** to set up the event of *“receive-changes”* to our Server while the other parameter is a ***socket\_handler*** to take in the delta that we get from our receive changes, That’s why we used the ***updateContents()*** feature from the quill tool to pass the delta which contain our last changes which is basically going to run these specific changes.

While the socket ***off()*** operation which takes the same parameters of *“receive-changes” and the Handler* will do exactly the same thing of setting up an event listener but this one is just updating our document to have the changes that are being passed from our other clients using the same document for editing.

By taking into considerations that clients using are two separate different computers working on the same document and our Server is doing all the Communication between the two objects (clients) to make sure the document is functioning exactly as we want.

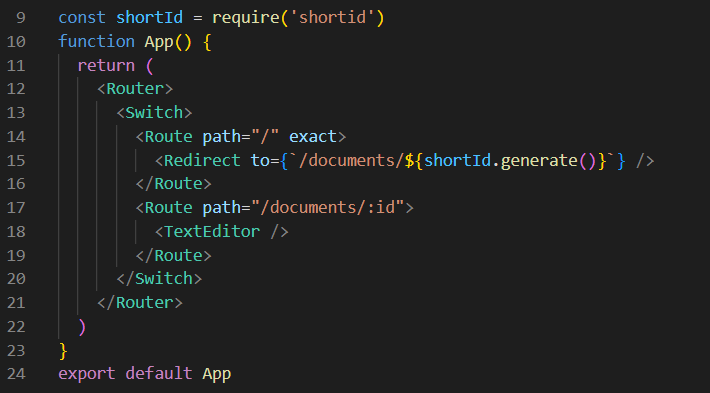


Figure 11: using React Routes

after setting up all the events for the server to listen to with using the ***useEffect()*** instances and the quill key features functions, the clients here are modifying the same instance of document, all on local host:3000. So we need to build a fix so we can then have multiple different documents by using ***React Router.***

After the imports and configurations, we use normal JavaScript switch cases with Wrapping Everything inside a Router Tag as shown in *figure 8*:

1. One Route for Redirecting to a different brand new document by generating a unique id (***shortId***)
2. Second route for Rendering the text edito

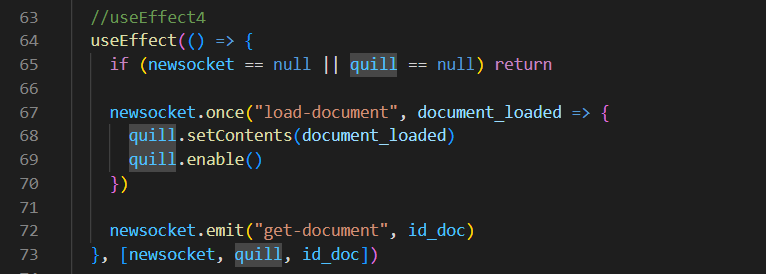


Figure 12: using useEffect() to load document

This ***useEffect()*** is used after ensuring we have an access to the id we specified in the route from the url of the document.

What happens is everytime the socket quill or the document Id changes we run the code in this ***useEffect()*** event, if the socket or the quill is not null, we want to tell the server what the document is actually a part of. So we used the socket function ***emit()*** and pass it *“get-document” & the document id.* This function send up to the server the document id so we can actually attach ourselves to the room for that document and if we have a document saved it’s going to send us that document back to us.

All we have to do is just make sure we listen to that event. So we use the socket function ***once()*** since we only want to listen to this event once, and this function will automatically clean up the event after it gets listened to once. This function takes *“load-ocument” and the document itself* as parameters.

So when we call get document, it is going to pass up to our server our document, the server is going to do some background operations to load our document and finally it is going to send it back down to our client with this load document event.

Then we are going to use some quill functions ***setContents()*** so we can load up our text editor and have the contents inside of it and ***enable().*** And this is used because we want to disable our text editor until our document is loaded.

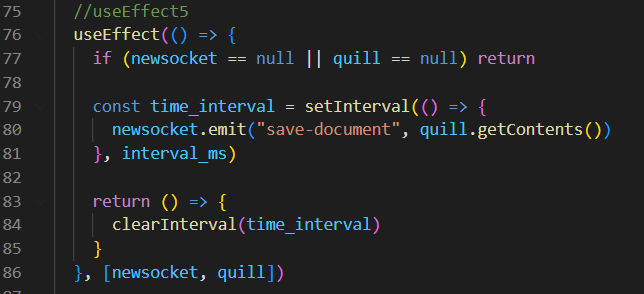


Figure 13: using useEffect() to make sure saved data is updated data passed

This useEffect() is basically setting a timer every couple of seconds we are going to save our document, this is done by the function ***setInterval()*** and here we pass it how milliseconds we want to wait, and we also ***emit()*** that save document function and in order to get the actual contents of our document, we use the ***quill.getContents()*** function to get all the information we need to save to the database.

Finally, we make sure we clear the interval so we don’t have it running anymore.

### **2nd: Server’s Code:**

1. **Document model File**

This file contains the declaration of database schema of the document we are using ***“DocumentSchema”,*** with some attributes ***id, data, dateCreated and users*** which is of type of users’ schema.Data is an object containing all the operations provided by QUILL features (insert, delete, etc…).

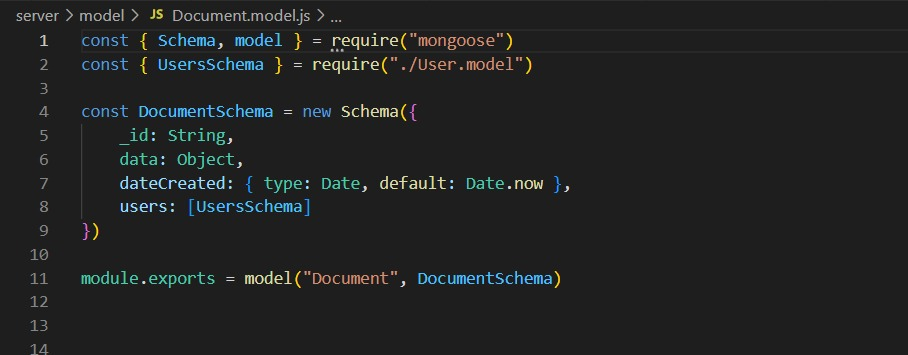


Figure 14: Document Model File

1. **User model file**

It contains a ***userSchema object*** consisting of the names of the clients currently using the document.

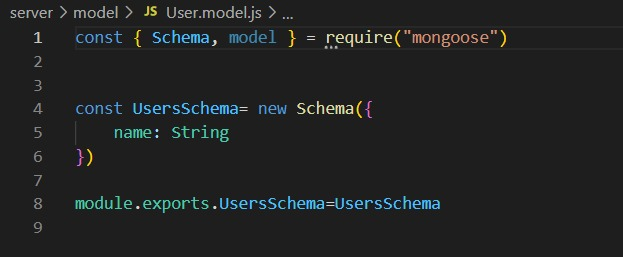


Figure 15: user model file

1. **DB.js file**

It simply contains a function ***mongoose.connect()*** passing it the URL of the db created to connect to the database replica set we created - *discussed in details in database replication section- .*

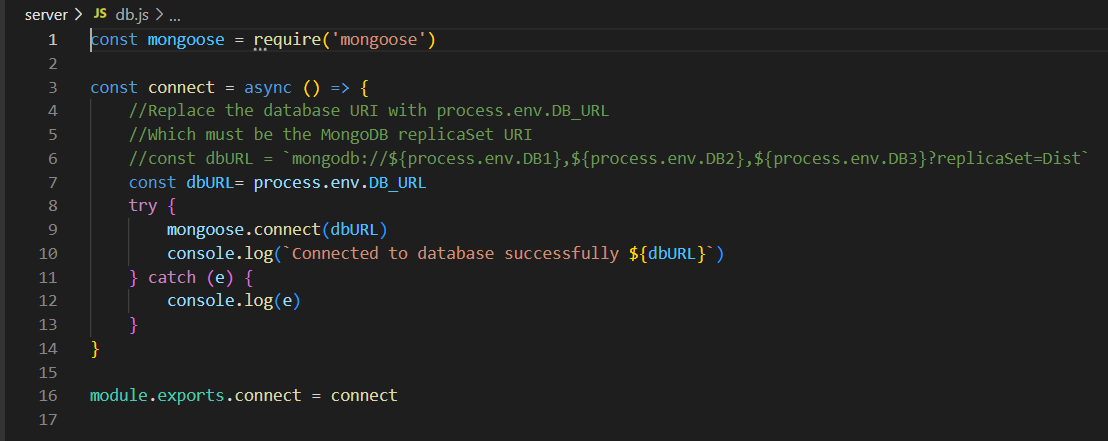


Figure 16: DB file connecting to the db replica set created

1. **Server.Js File**

Starting by importing some libraries on the server side by importing the env file configurations that we are going to use later in the code by **require(“dotend”).config()**  and the document model that we are previously explained by **require (“./model/document.model”)** , the redis connector by **require(“socket.io”)** and the redis adaptor by **require(“@socket.io/redis-adapter”)**

After that we are connecting to the mongo database by using the command **dbconnect()**

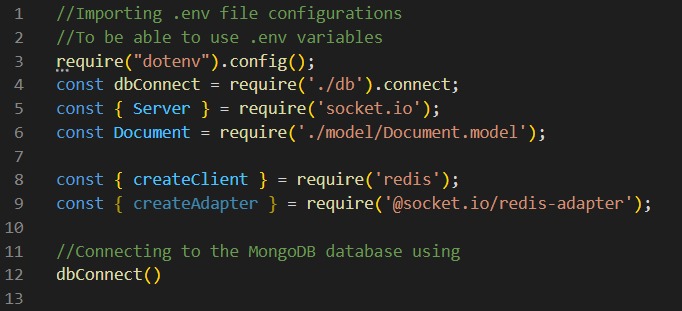
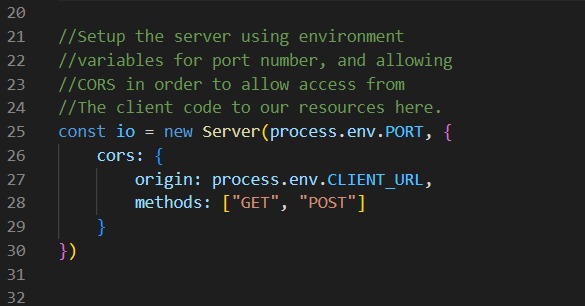


Figure 17: Configuration of the Server.js File

Here we are setting up the server using environment that we were importing in the previous screenshot, by taking its **cors** which are the origin and the methods that the server can use which are post and get.

Figure 18: setup a new environment for the server

Here we are making a new variable that will be used several times which is **defaultValue** and initialzing it with empty string. As it is the very first value in the document that will appear to the user in the first place.

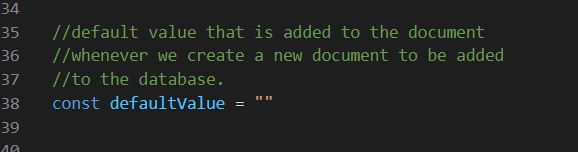


Figure 19: Initialzing the defaultValue of the Document

Here we are initialzing some arrays. The first one is **Allclient[]** which will contain all the clients that are establishing the web socket connection on the same server. So that the server could keep track of the number of clients connecting and disconnecting from it.

The second array is **redisPub** w **redisSub** are the arrays containing all the publisher and subscriber to the channel using the same web socket connection.

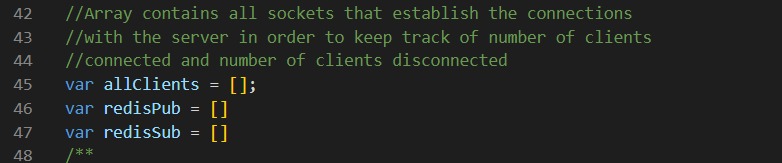


Figure 20: initializing some Arrays

This is the main **function createClient()** of creating the client by initializing the publisher and subscriber redis clients in order to subscribe to the same single channel so that all the websosckets can connected to each other using it. Also, when some changes happen, they all got to know the changes that happen or the request that arrives by single socket.

All of them are connected to the same URL first , then if this process works well , then the console will output a message that the publisher is ready to be used and connected successfully , else, will throw an error message.

Then after the successful connection, the clients will be pushed into the array allClients by **redisPub.push(pubClient)** that we previously initialized so that the servers will have access to all the clients that are connected to it by the client socket ID.

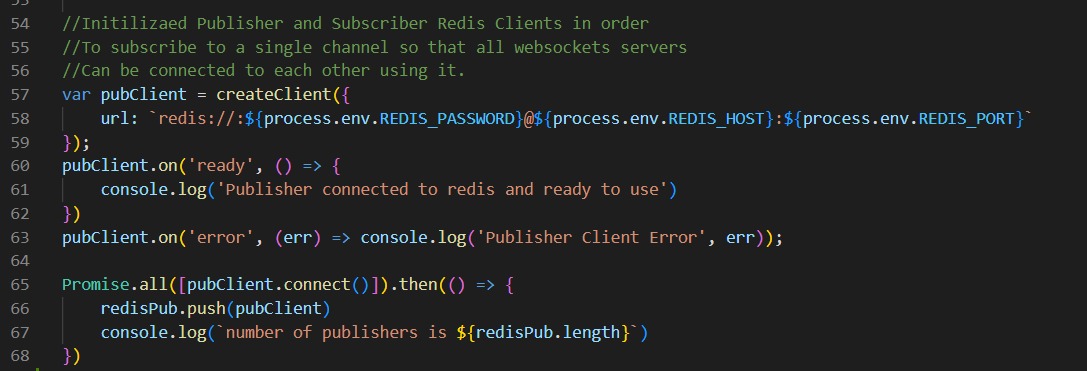


Figure 21: the main CreateClient() function

this one is creating an event listener for the connection event upon request. This will happen and run whenever one client establish the connection with the server and the server is ready to connect with it upon the channel.

So Firstly, the connection is establish and a message is printed on the console by **console.log(“subscriber is connected to redis and ready to be used”)** for the server that the client which acts as a subscriber to the redis channel that its is connected successfully and ready to be use

Then when the subscriber are connecting successfully, it returned to the client that he is connected on which server with the username of the server that the client is connecting on it right now by **console.log(“subscriber connected to $(username))**

Then after the successful connection, the clients will be pushed into the array allclients by **redisPub.push(pubClient)** that we previously initializing so that the servers will have access to all the clients that are connected to it by the client socket ID.

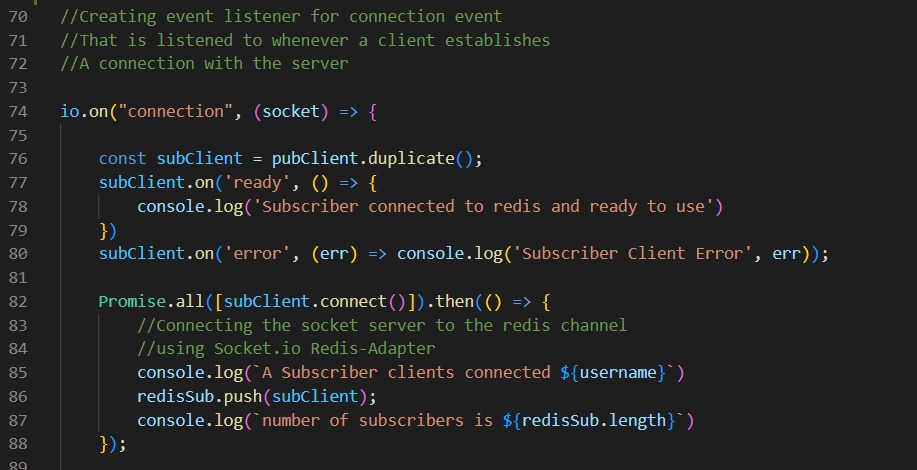


Figure 22: create the connection Socket

Here is to whenever the client connected to the server, we fetch the server username from the **socket.handshake.querey** library so that we can log it with the username on it.

Then the socket ID will be pushed to the **allclients** Array by **allClients.push(socket)** that we previously initialized so that the server will get to access the sockets that are connected and subscribed to the same redis channel.

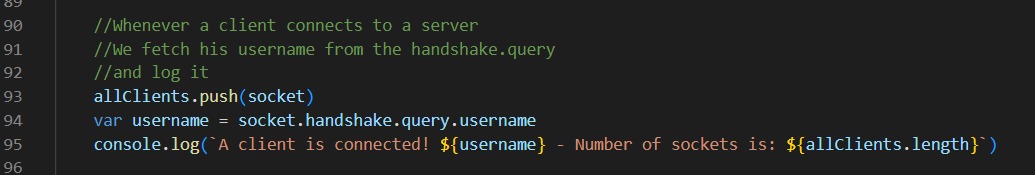


Figure 23: the push function in the server side

Here is to the **lookupDocument()** event that take the socket ID as a parameter.

It firstly check if the parameter is Null to return nothing, else to check the document with this ID by **await Document.findbyid(id)** in the database, if found then it will return the whole document to the client as a return message. Else if the ID is not found, then it will create a new document with this new ID and will put **the initialization value = defaultValue** which we described earlier in the code which is an empty string , to open an empty document for the user.



# Publish Subscribe Architecture

## Redis Publish Subscribe

Publish/subscribe messaging, or pub/sub messaging, is a form of asynchronous service-to-service communication used in serverless and microservices architectures. In a pub/sub model, any message published to a topic is immediately received by all of the subscribers to the topic. Pub/sub messaging can be used to enable [event-driven architectures](https://aws.amazon.com/event-driven-architecture/), or to decouple applications to build execution, dependability and scalability.

In modern cloud architecture, applications are decoupled into more modest, free structure obstructs that are simpler to create, send and keep up with. Publish/Subscribe messaging provides instant event driven messages for those distributed applications.

The Publish Subscribe model permits messages to nonconcurrently be communicated to various pieces of a framework asynchronously. A sibling to a **message queue**, a message topic provides a lightweight mechanism to broadcast asynchronous event notifications, and endpoints that allow software components to connect to the topic in order to send and receive those messages. To communicate a message, a part called a distributer essentially pushes a message to the subject. Unlike **message queues**, which batch messages until they are recovered, message topics transfer messages with minor queuing, and push them out immediately to all subscribers. All components that subscribe to the channel will receive all messages that are broadcast to all components, unless a message filtering policy is settled by the subscriber.

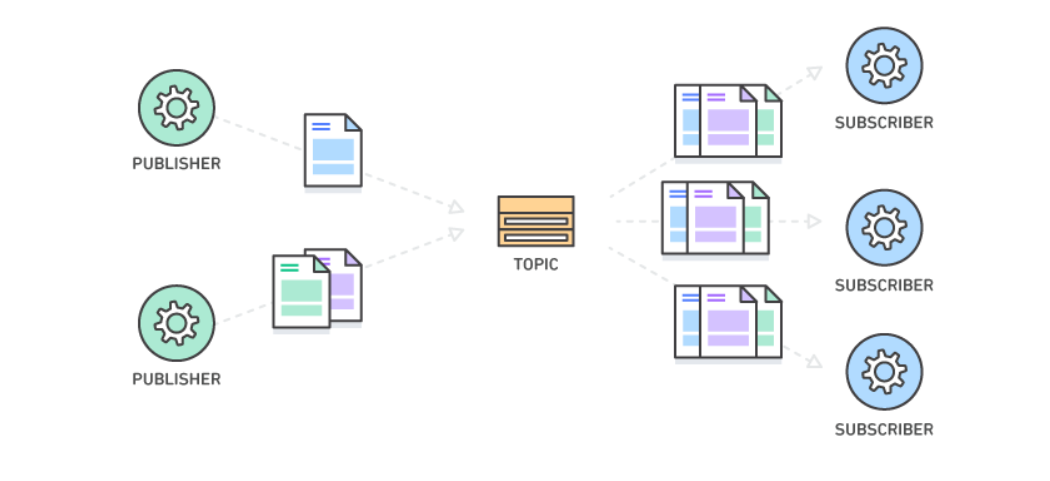


Figure 24: Publish Subscribe Overview

## Publish Subscribe Redis

Beside data storage, Redis can be used as a Publisher/Subscriber Architecture platform . In this architecture, publishers can send messages to any number of subscribers on a channel. These messages are fire-and-forget, in that if a message is published and no subscribers exists, the message is deleted and cannot be recovered.

Once one component subscribed to this channel by certain ID, the client is put into subscriber mode and no commands can be issued by the client. In that way, the client has become on read-only mode. Publishing has no such limitation.

Messages sent by different clients to these channels will be moved by Redis to every one of the bought in clients..

A client subscribed to one or more channels should not issue commands, although it can subscribe and unsubscribe to and from other channels. The answers to subscribtion and unsubscribtion are sent as messages, so the client can just peruse a reasonable stream of messages where the principal component demonstrates the kind of message. The commands that are permitted with regards to a bought in those commands could be subscribe, unsubscribe, sunsubscribe , psubscribe , PING , reset, quit .

Since every one of the messages got contain the first membership causing the message delivery (the direct on account of message type, and the first example on account of message type) client libraries might tie the first membership to callbacks (that can be anonymous functions, blocks, function pointers), utilizing a hash table.

## Format of pushed messages

The first element is the kind of message:

* subscribe: implies that we effectively subscribe into the channel allowed as the second component in the answer. The third part addresses the channels' numbers we are at present subscribed into.
* unsubscribe: implies that we effectively withdrew from the channel given as second component in the answer. The third part addresses the channels' numbers we are at present subscribe into. At the point when the last argument is zero, we are not generally subscribed into any channel, and the client can give any sort of Redis order as we are outside the Pub/Sub state.
* message: it is a message gotten as consequence of a PUBLISH order gave by another client. The subsequent component is the name of the starting channel, and the third part is the real message payload

## how the publish/subscribe Architecture works

Software configuration designs depend on building reusable courses of action of modules and their interconnections. These modules are regularly classes or items addressed in an UML configuration diagram. However, when you look at modern architectural patterns, the modules are larger, self-executing processes spread across distributed systems.

To see the value in the benefits of the Pub/Sub design, you should begin from the fundamental example whereupon a data framework is constructed and follow its development towards a distributed system.

Typically, an information system is an assembly of a generalized set of software modules that follow this simple sequential pattern.

At this scale, the system faces the problem of routing messages from input modules to their individual result modules. To tackle this issue, the info and result modules will require an addressing component. The handling component will deal with the messages and course them to the right beneficiary with the aid of the address. Every one of the three modules work together in tackling the issue of routing.

At Internet scale, the framework will deal with great many simultaneous connections. The framework will get messages from and send messages to clients everywhere. It necessities to also be fit for taking care of high volume and worldwide geological spread of clients.

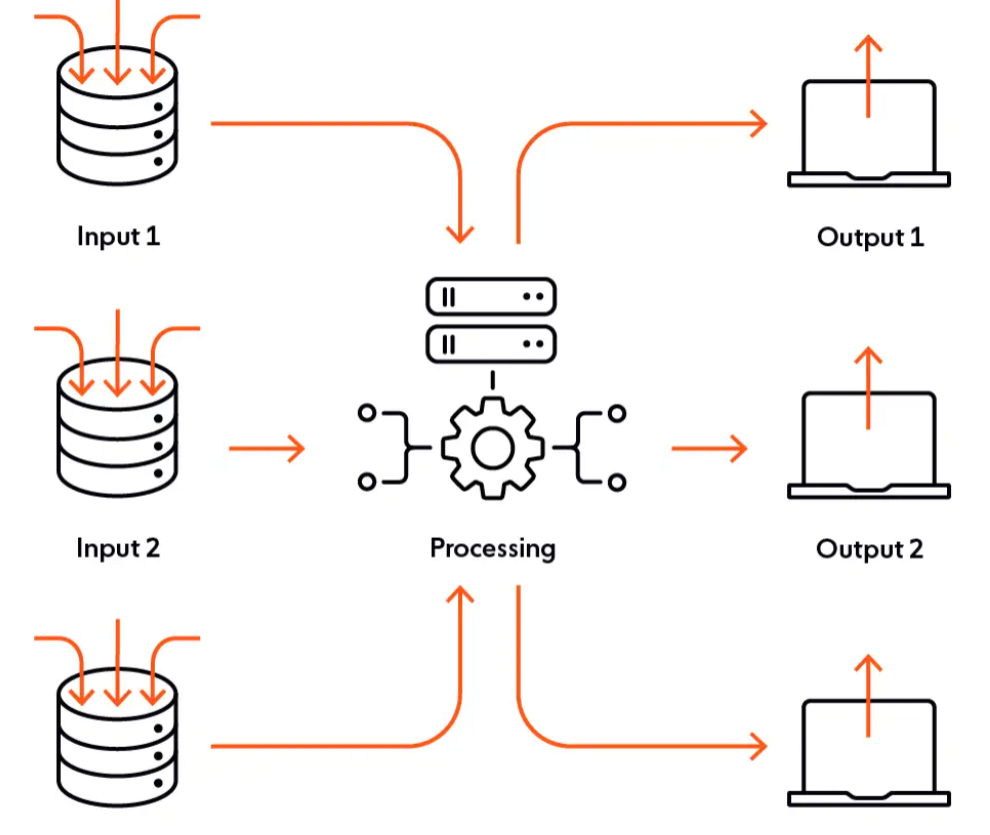


Figure 25: how Publish Subscribe Redis works

## Implementation of Redis publish Subscribe Architecture

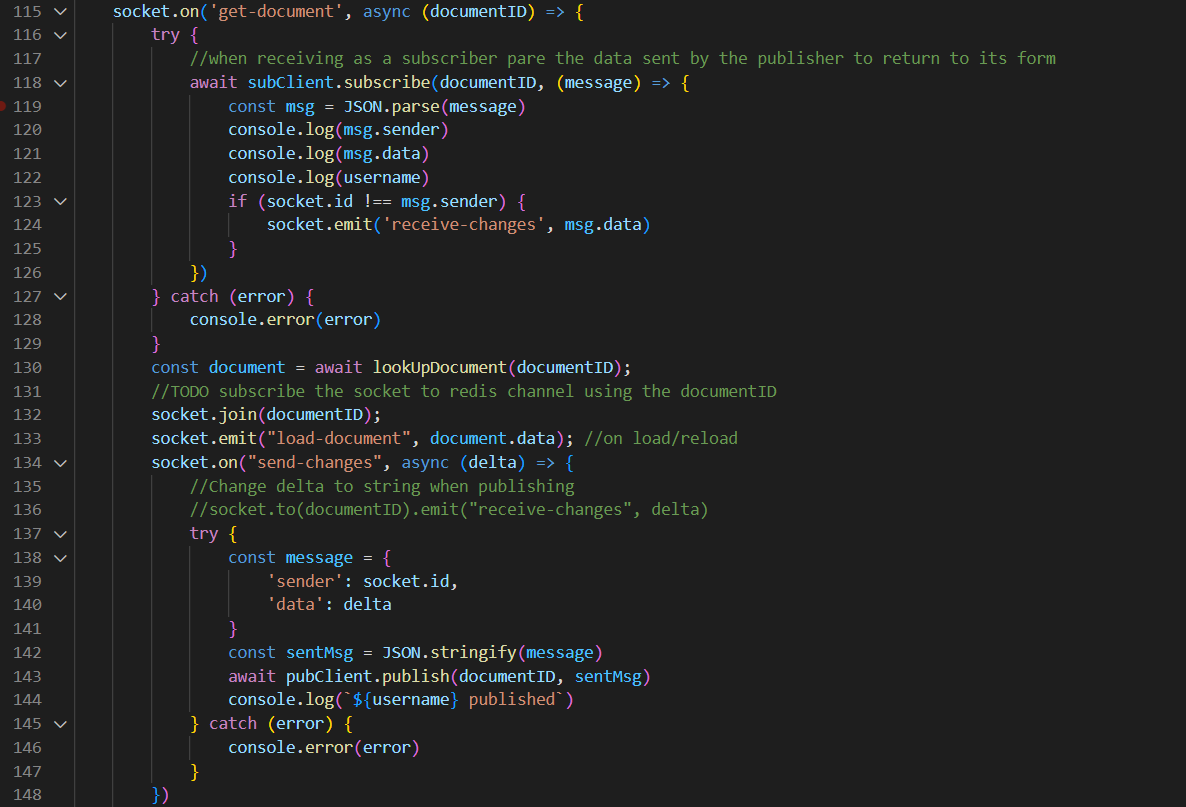


Figure 26 : Redis Publish Subscribe Architecture

First, all the servers started to subscribe to the redis channel in order for all edits to be sent to all of them. This subscription takes the document channel ID and the data or message to be sent to all of the servers.

Then there is a check in line 123-124 for the socket number with the one who sent the data with, as the data will be sent to all the sockets except the one who sent among this message with.

Noted that the servers are acting as a publisher and subscriber both to the same channel. So the sockets now are connected to the servers, so when one server send a request (by the socket ID and the data wanted to be send or edits in the document) , then these messages will be sent to all sockets that are connected to the servers except the one who send the message.

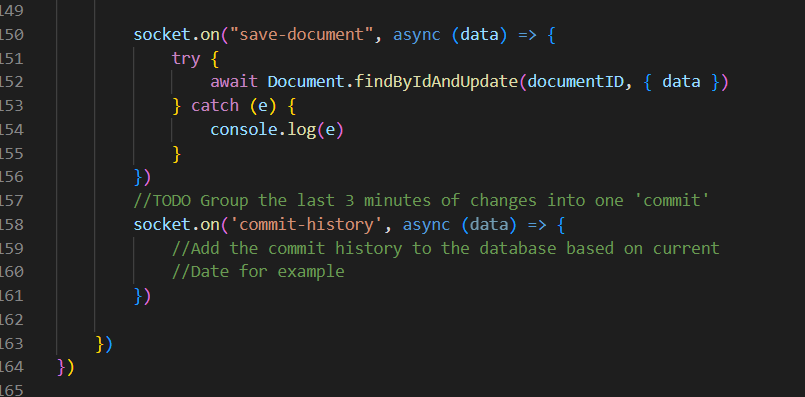


Figure 27: Redis Publish Subscribe commit to the database

Here, sockets are waiting for any changes to be received else, there are going to commit the history to the database based on what message received or changes happen.

## Load Balancer

Load balancing refers to the act of distributing network traffic across multiple services.

This makes sure that there's not too much load on a single server which could cause it to crash.

A load balancer acts as a ‘reverse-proxy’ to represent the application servers to the client through a virtual IP address (VIP). This technology is known as server load balancing (SLB). SLB is designed for pools of application servers within a single site or local area network (LAN).

Load balancers health check the application on the server to determine its availability. If the health check fails, the load balancer takes that instance of the application out of its pool of available servers. When the application comes back online, the health check validates its availability and the server is put back into the availability pool.

Because the load balancer is sitting in between the client and application server and managing the connection, it has the ability to perform other functions. The load balancer can perform content switching, provide content-based security like web application firewalls (WAF), and authentication enhancements like two factor authentication (2FA).

This is the primary function of the load balancer, server load balancing (SLB). The agent can provide additional functionality based on their role in the conversation. They can decide to allow and/or deny certain details (security). They may want to validate that the person they are talking to (authentication).

### Benefits of using Load Balancer

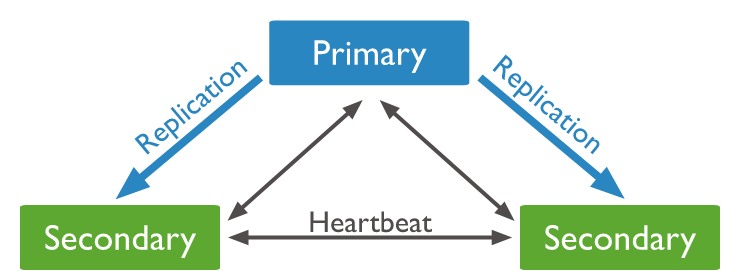
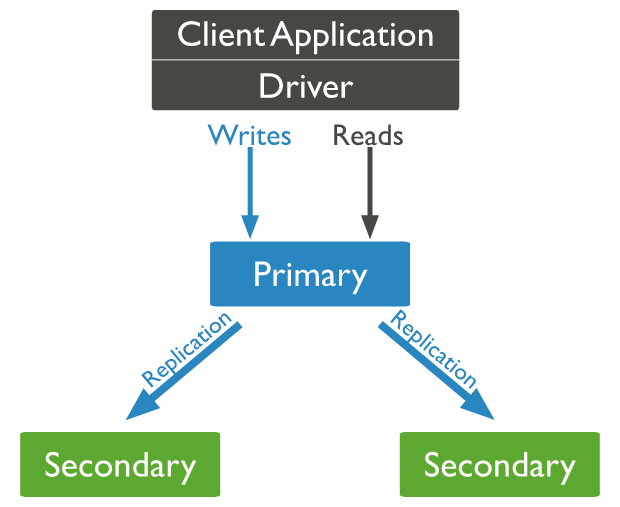
* it helps improve the responsiveness of your application.
* It also limits the chances of servers crashing as they're not being subjected to loads beyond what they're capable of withstanding.
* This is done to ensure maximum speed and capacity utilization

## Load balancer Implementation in Multi-Users Distributed Text Editor

# Database Replication

Due to the crucial component in the distributed systems applicationswhich is **Data Availability,** data base replication is the way to address this component.

**Database Replication in distributed systems applications** is the process of building multiple copies of data and store them in different locations for mainly the sake of **backup,** similarly to data mirroring, data replication can be applied to both individual computers and servers, it also refers to Data distribution from a source server to other servers while being updated and synced with the source so that users can access data relevant to their activities without interfering with the work of others.



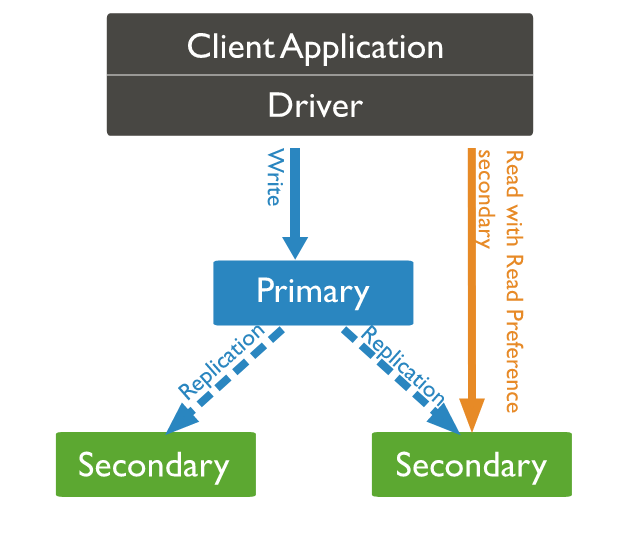
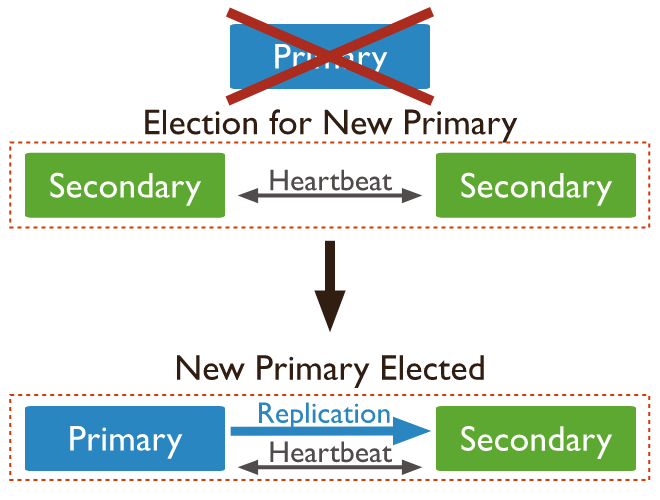


Figure 28 - database replication illustration

Data replicates can be stored:

* Within the same system
* Onsite and offsite hosts
* Cloud-based hosts

## Importance of Database replication in distributed systems:

1. **High availability** 🡪 as we have mentioned that the data availability is the crucial component for a successful distributed system application and it’s applied by implementing the concept of data replication as it is the most effective way to increase the data availability, data can be also replicated over multiple locations so that the user can be able to access it if some copies became unavailable or lost as a result of any site failures.
2. **Fault tolerance** 🡪 when any network/system/site fails occurs, the system succeeds to operate as when a replica fails, the service can be served by another replica.
3. **Read Scalability** 🡪 read queries can be serviced from copies of the same data that have been already replicated and this helps to boost the overall throughput of queries
4. **Reduce Latency** 🡪 that can be applied when keeping data **geographically** closer to the user, replication helps to reduce data query latency. Example is *CDNs (Content Delivery Networks)* whose applications (i.e. Netflix) succeed to retain a copy of duplicated data closer to the user.

## Common Types of Data Replication in distributed systems:

1. **Asynchronous vs Synchronous Replication**

|  |  |
| --- | --- |
| Asynchronous Replication | Synchronous Replication |
| The replica gets modified after the commit is done onto the database. | The replica gets modified immediately after some changes are made in the relation table. |

1. **Based on Server Model**

|  |  |  |
| --- | --- | --- |
| **Single Leader Architecture** | **Multi Leader Architecture** | No Leader Architecture |
| One server accepts client writes and replicas pull data from it. | Multiple servers can accept writes and serve as a model for replicas. | Every server can receive writes and work as a replica model. |
| It’s a synchronous technique but It’s quite rigid. | Leaders should be in close proximity to all of the copies to avoid delays. | Despite it provides maximum flexibility, it makes synchronization difficult |

## Advantages of data replication in distributed systems:

* + **Enhanced Performance:** Users can obtain data from the server nearest to them because the same data is stored in multiple locations, reducing network latency and increasing speed.
  + **Increased Availability:** Multiple users are allowed to manage and view data without them interfering with each other.
  + **Allows Multiple User Access:** for query-execution when multiple users accessing.
  + **Ensures business continuity:** when site/network failures in important business systems, crucial data is never lost but van be recovered and business in maintained to continue.

## Disadvantages of data replication in distributed systems:

* + **Large storage space:** especially when using full replication technique where many copies need to be synchronized and updated which may lead to high costs and reduced performance.
  + **Maintenance costs:** when running multiple servers together.
  + **Out of date or incorrect data replication:** when some sources may be out of sync due to any network failures, may lead to unnecessary data kept.

## Database Replication Implementation in Multi-users distributed Text Editor

We used ***MongoDB replica- sets*** for implementing the database replication technology in our Multi-Users Distributed Text Editor which is mainly characterized by being:

* Synchronous Replication
* Single leader Architecture based

And these types of database we have discussed in the previous section.

## Using MongoDB replica-sets

First, we implemented replication by creating 3 different databases on 3 different terminals on their own but all in the same replica set. Then, we defined one database of them as a PRIMARY database and the others are automatically defined as SECONDARY database.

We start by Running the command: *mongod -replSet Dist –dbpath mongos/db1 –port 27018*

To be able to configure our desired 3 database replicas in the same replica set.

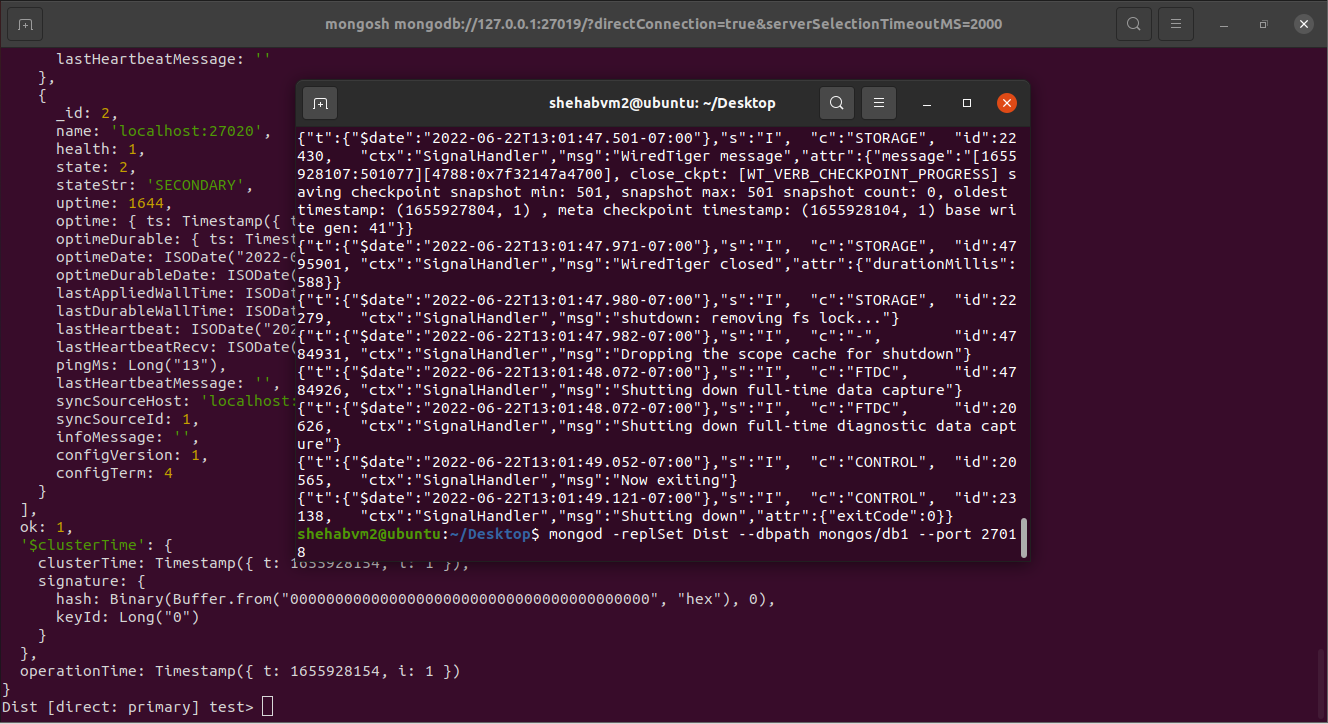


Figure 29: mongod command to start the MongoDB database associated with replicaSet

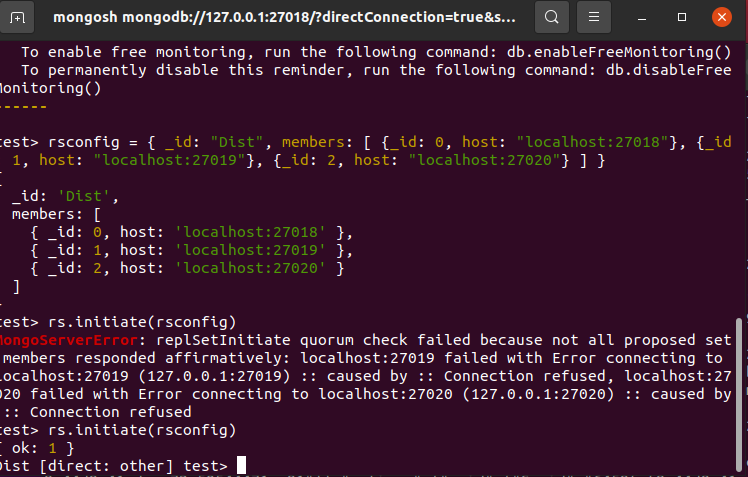
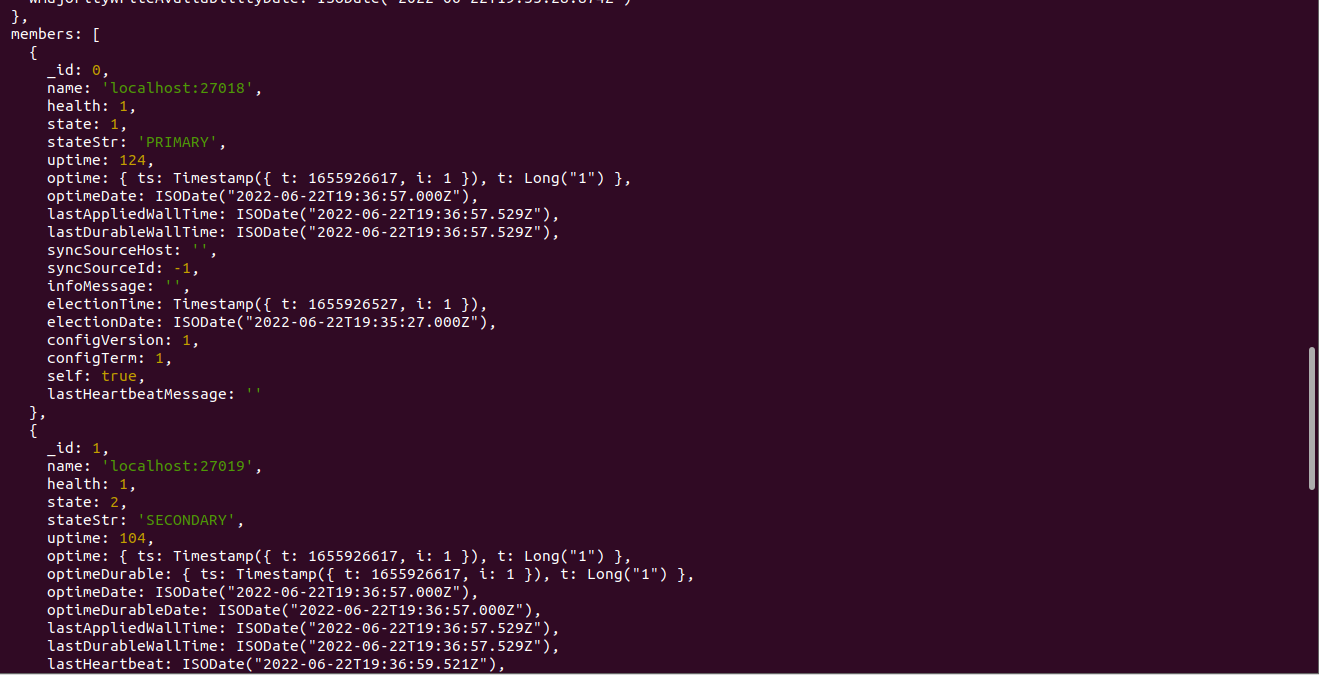
Then we start configuring our 3 database replicas by defining a variable *rsconfig*, where we define 3 databases with the same local host but different IDs and port numbers addresses of 27018, 27019, 27020. Later we initiate our replica set using the defined configuration variable *rsconfig* using *rs.initiate(rsconfig)*

Figure 30: configuration of the databases

Then we can check for the status of the replicaSet using the following command *rs.status()*

The members of the replica setwith the information for each database are then displayed, one database is specified as a PRIMARY database while the others are automatically set to secondary. each database has its own info of:

* ID
* Name “host + port number”
* State either primary, secondary, or unreachable

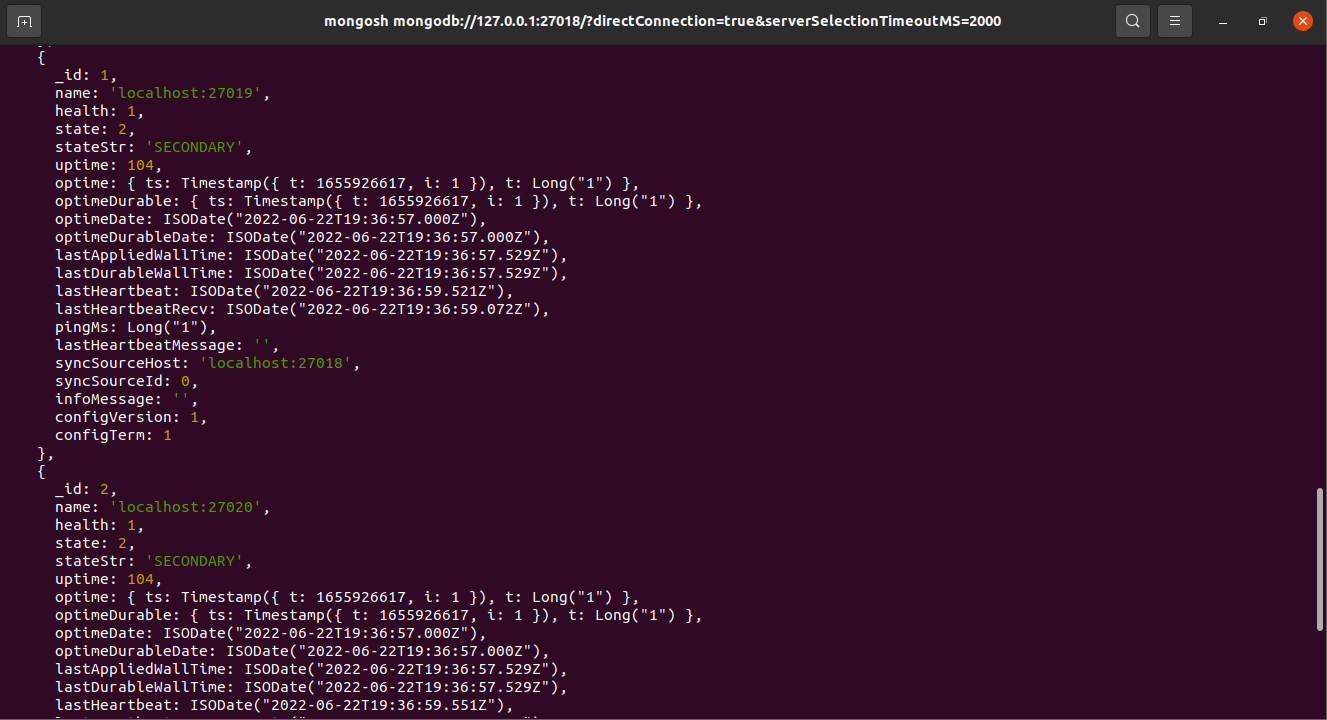
Figure 31: database 1 info

Figure 32: database 2 info

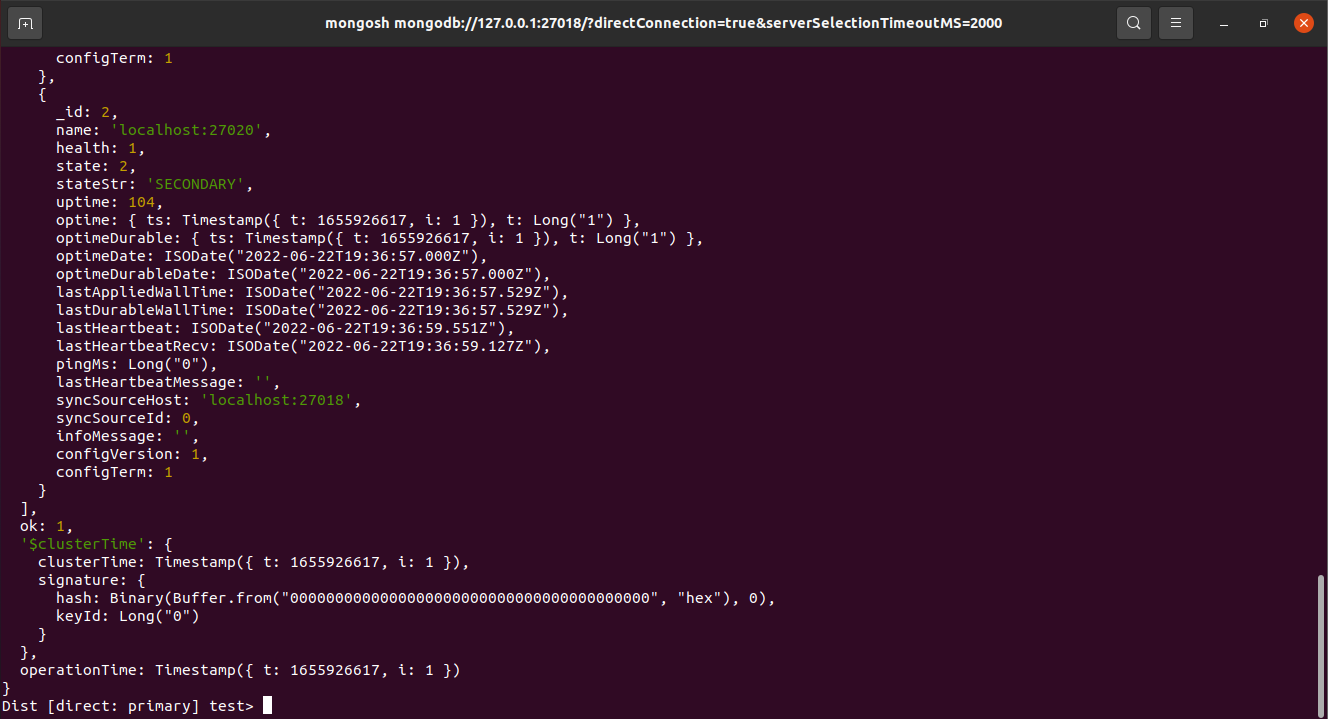


Figure 33: database 3 info

Then we tried to shut down Database 1 which has the port number 27018 and check which database will be the primary database. It was Database 2 which has the port number 27019 that became the new Primary database.



Figure 34: rs.Status() of DB1 cannot connect because of db failure.

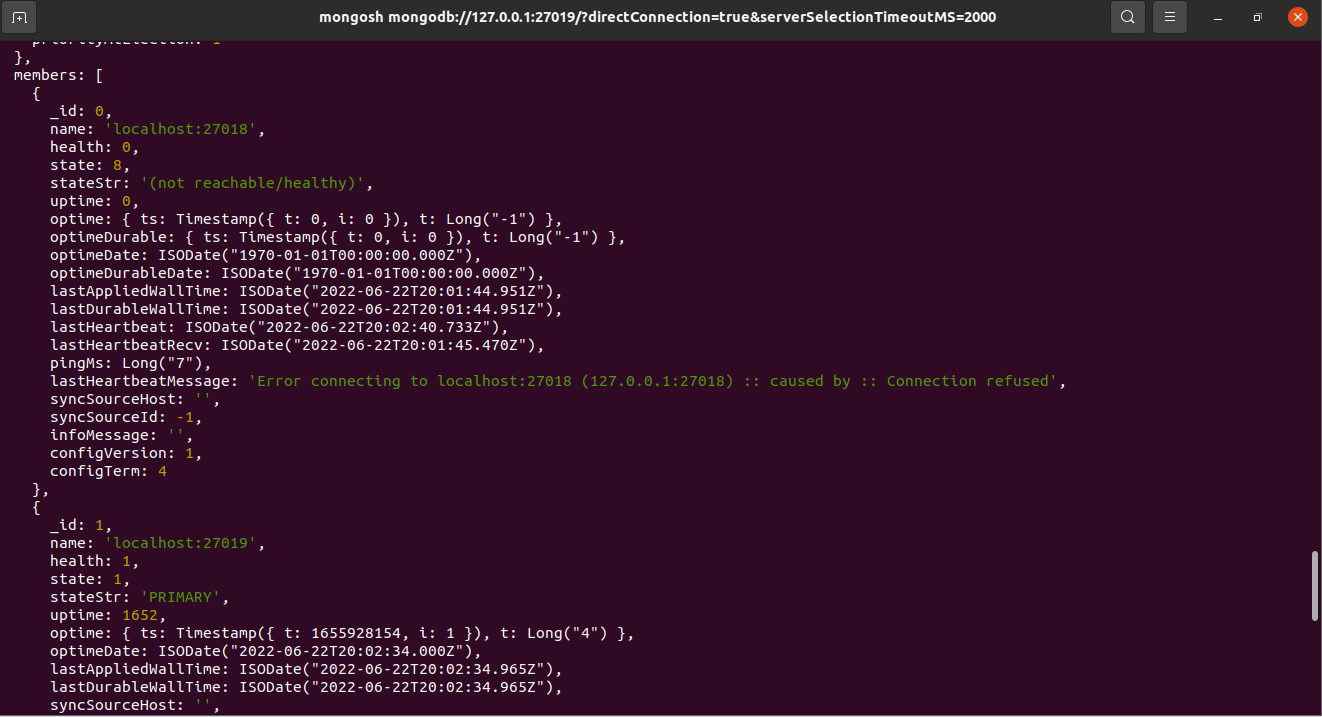


Figure 35: Database 1 became the Primary Database

As seen above, after shutting down database 0 with port number 27018. The database 1 with port number 27019 became the Primary database instead of being a Secondary database. While the third database 2 with port number 27020 stayed as a Secondary database. If we tried to bring back database 0 with port number 27018 and run it again. It will be a secondary database.

For the Primary database currently available, we are able to read and write to the collections, but for the secondary ones we are able only to read from it but an access is required to specify reading from the databases with status “Secondary”. This is done by running the command: *db.getMongo().setReadPref(‘secondary’).*

And to return the data written to the document in the primary to the users on the same shared document we use the command: *db.users.find()*

*Text

Description automatically generated*

Figure 36 - Inserting data to a secondary database

Figure 37: Inserting data on a Primary database successfully.

Text

Description automatically generatedFigure 38:enabling read access on the secondary database

Figure 39 - Fetching documents from a secondary database.

## Database Dockerization

Final Updates is made by Installing ***Docker***, then creating a network called ***DistN.***

sudo docker network create DistN

sudo docker run -d -p 30001:27017 --net DistN --name db1 mongo:latest --replSet Dist

The ***detach*** flag means the container will run in the background, separately to the shell process.

The container external port 30001 is bound back to the internal port of the container 27017 on the host. We will be able to connect to Mongo instance on localhost:30001. If you want to change the port number, modify the first part of the ***-p flag,*** such as 9000:27017 to use localhost:9000.

* Bound it to the network we created ***DistN***
* Giving the container a name ***db1***
* Creating a ***replicaSet Dist***

Run the second and third containers using the following commands.

But We'll have to change the external ports and database names.

sudo docker run -d -p 30002:27017 --net DistN --name db2 mongo:latest --replSet Dist

sudo docker run -d -p 30003:27017 --net DistN --name db3 mongo:latest --replSet Dist

In order to create the replicaSet we have to login to the first database in an interactive mode and launch the mongo shell.

sudo docker exec -it db1 mongo

Then create a configuration:

rsconfig={\_id:"Dist",

members:[

{\_id:0, host:"Enter your machine public IP:30001"},

{\_id:1, host:"Enter your machine public IP:30002"},

{\_id:2, host:"Enter your machine public IP:30003"}

]}

Run rs.initiate(rsconfig)

It must be working now!

Run rs.status() to make sure it is working well.

# Testing Scenarios and Results:

# Failure scenarios:

1. When Primary database fails.

🡪 The mongoDB replica sets implements automatically that the secondary database will take the place of the Primary one so no data loss occurs and the processes of the system remain working; shown in the following figures.

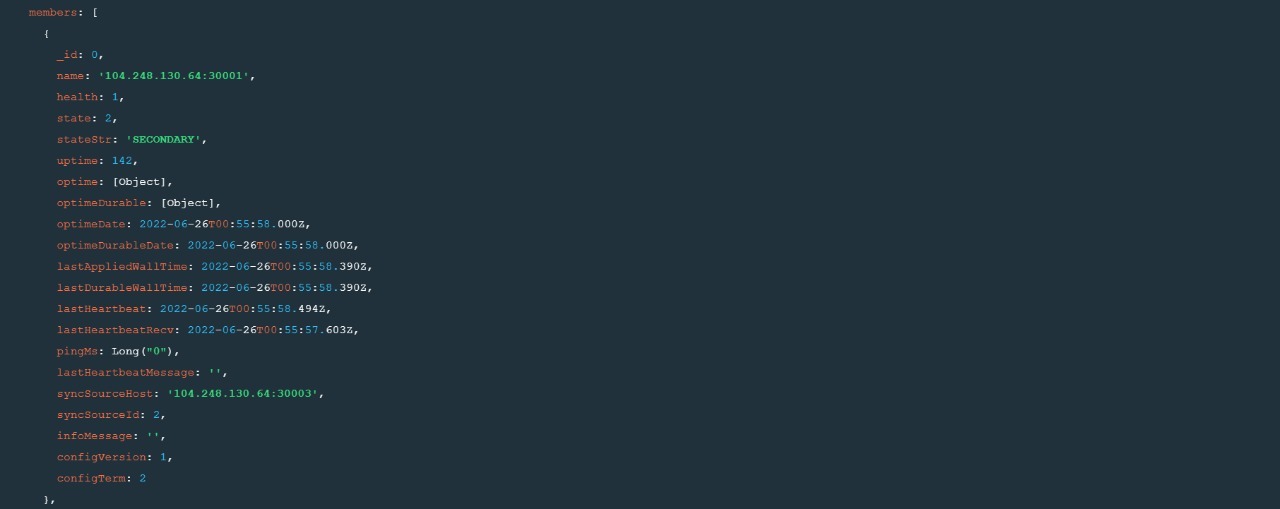


Figure 40: database 1



Figure 41: database 2



Figure 42: database 3

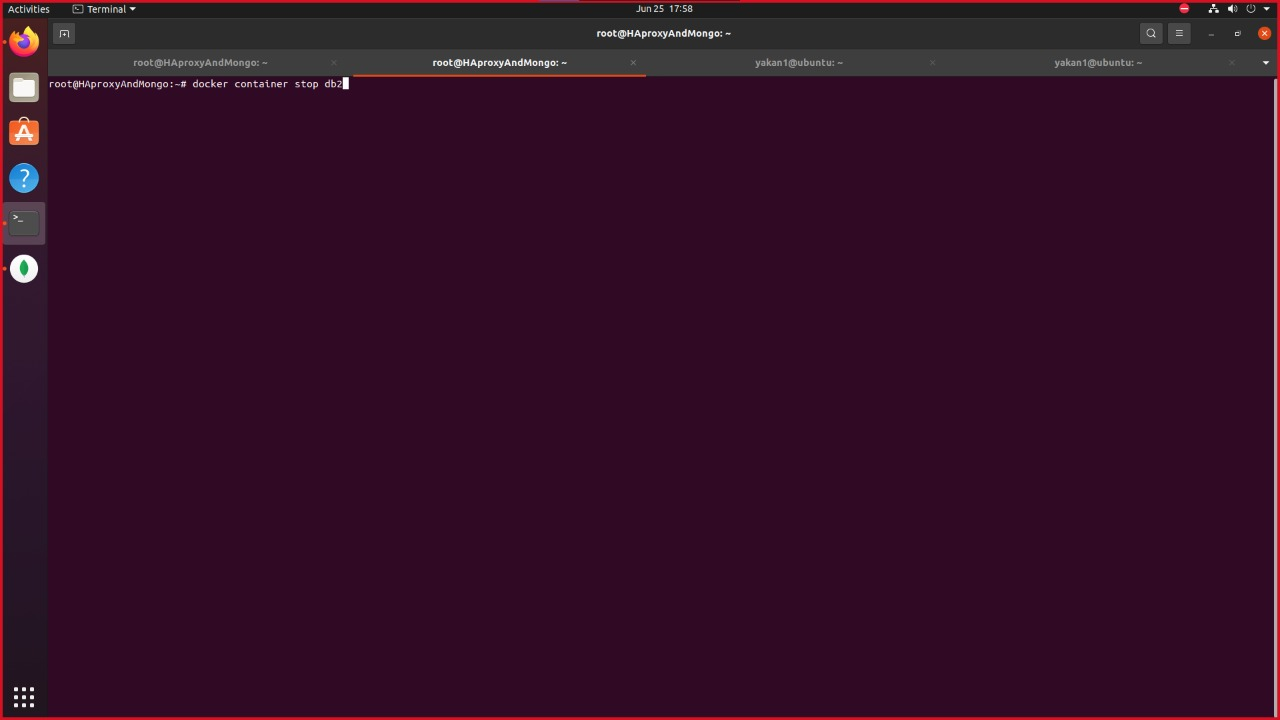


Figure 43: stopping current primary

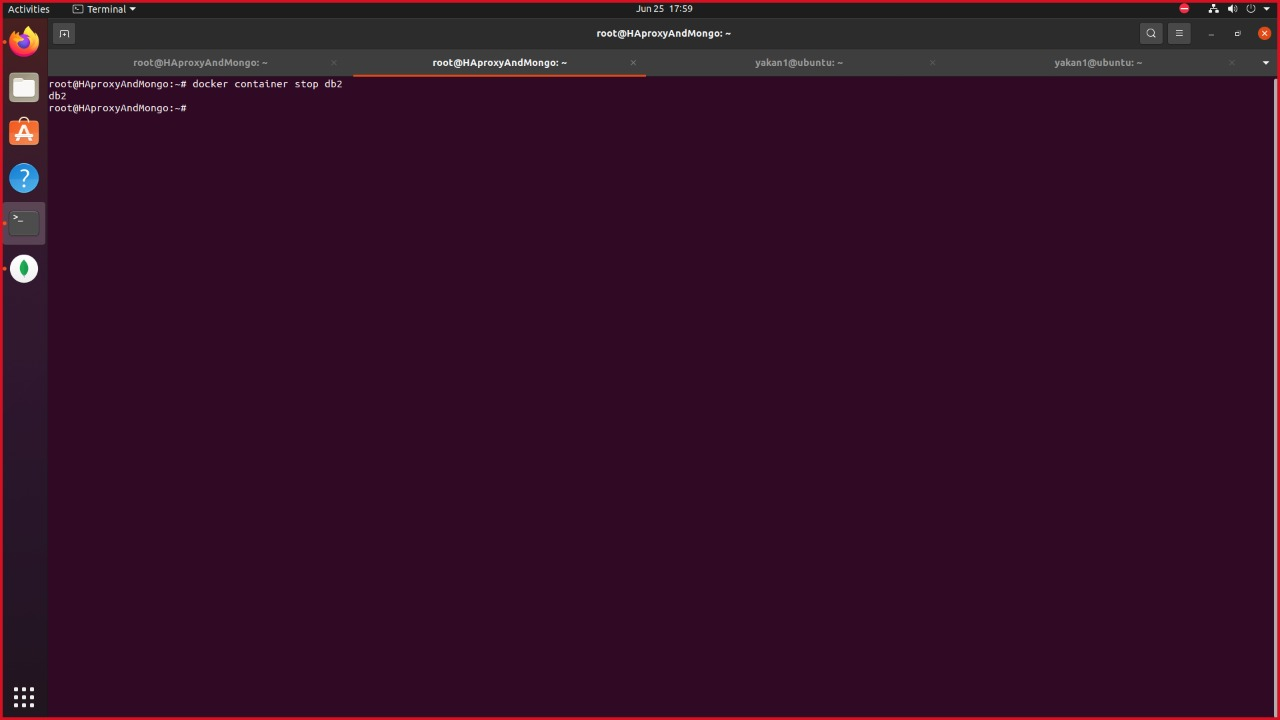


Figure 44: Primary database stopped

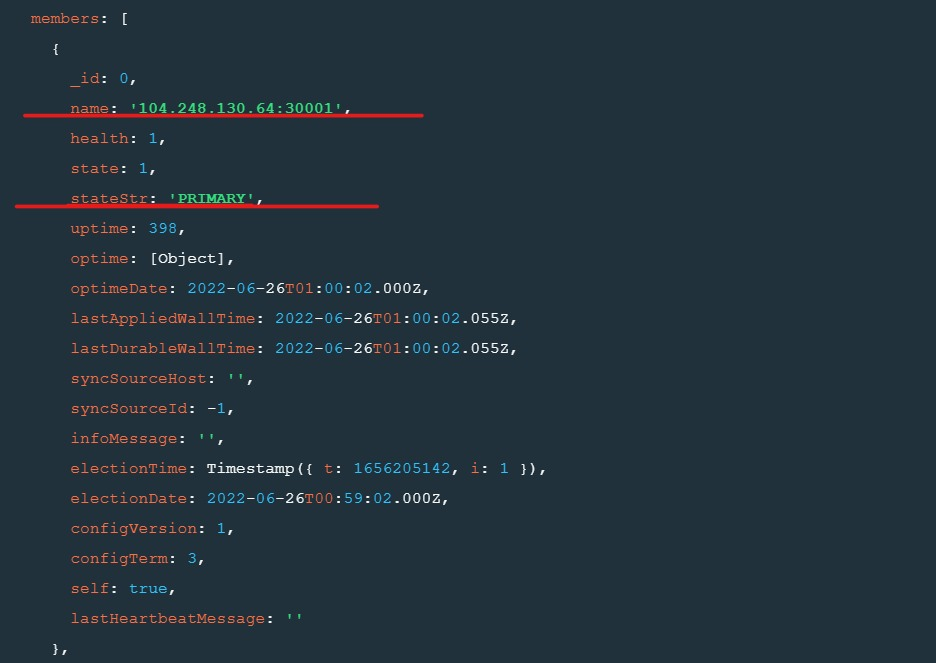


Figure 45: changed database after terminating the primary

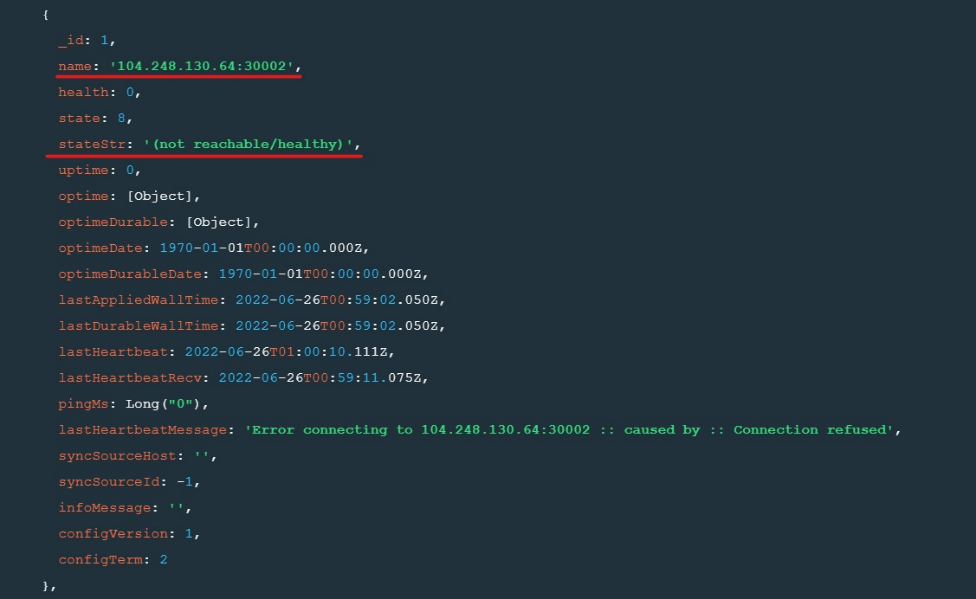


Figure 46: Terminated Server



Figure 47: third database



Figure 48: restarted previous primary container

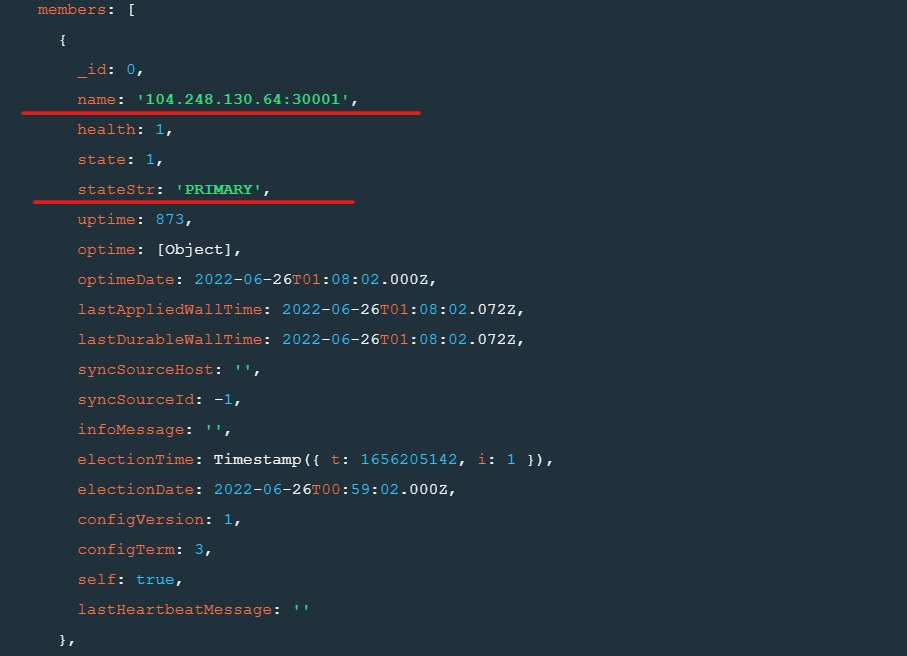


Figure 49: After Restarting



Figure 50: previous Primary database now restarted as Secondary



Figure 51: Secondary database Currently

# Conclusion:

# End User Guide:

# References

<https://hackernoon.com/best-practices-for-event-driven-microservice-architecture-e034p21lk>

<https://softobiz.com/understanding-the-event-driven-architecture/>

<https://www.astera.com/type/blog/rest-api-definition/>

<https://restfulapi.net/>

<https://hevodata.com/learn/data-replication-in-distributed-system/#intro>