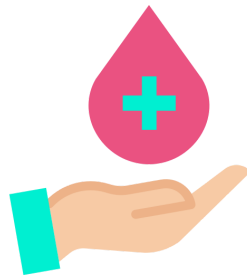


Final Project Report

Implementation of a Blood Donation Cloud-Based Platform



Somya Sharma - 827006361

Sarra Bounouh - 327005574

Janvi Palan - 427000511

Robert Quan - 622005281

05/05/2019

CSCE 678

TABLE OF CONTENTS

TABLE OF CONTENTS	1
1. INTRODUCTION	3
2. PROJECT STACK	4
3. RELATED WORK	5
4. TECHNICAL IMPLEMENTATION	7
4.1 Phase 1: Exploration Phase - Architecture fully AWS	7
4.2 Phase 2: Implementation Phase - MEAN Stack	9
4.2.1 Tech Stack	9
<i>Application Development Life Cycle</i>	9
<i>Database - MongoDB</i>	10
4.2.2 Data model	14
4.2.3 Application Backend - Logic	16
<i>Aggregate pipeline</i>	16
<i>Handling Geolocation</i>	18
<i>Address Geocoding</i>	19
4.3 Security	21
4.3.1 CORS Policy	21
4.3.2 Hashing and Salting	22
4.3.3 Implementing JWT authentication	23
4.3.4. Implementing reCAPTCHA	26
4.4 Containerizing the App	27
4.5 Hosting the App	29
5. EVALUATION	31
5.1 Data loading performance	31
5.2 Security evaluation	32
5.3 MongoDB and CAP Theorem	33
6. DISCUSSION	34
6.1 Data security	34
6.2 Issues and challenges	35
7. CONCLUSION	36
8. REFERENCES	37

APPENDIX	39
Node Dependencies	39
Angular Dependencies	39
Installations	40
Github URL	40

1. INTRODUCTION

According to a 2014 study [1][6], over 50% security breaches occur in the medical industry, and with up to 90% healthcare organizations having exposed their data. Therefore, new methods, architectures, or computing paradigms may be needed to address security and privacy problems in medical data sharing area.

In this paper, we present a web application that focuses on creating a matching platform for blood donors and blood receivers. Our motivation is to create an easy and secure way for recipients to find donors that are located near them based on latitude and longitude location and the specific type of blood that they can donate and receive.

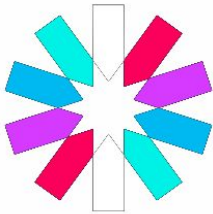
The technologies we are using for this project are listed in the next section, Project Stack. This stack concentrates on using an Angular based Web App that is hosted on Amazon Web Services Elastic Container Service. To ensure data privacy, we aim for a usage of our app exclusively by hospitals and health organizations, and we utilize authentication and role-based data access features, among other functionalities to enhance our system's security.

The web app is based on the MEAN stack (MongoDB, Express, Angular, NodeJS), a popular web app framework, that we developed and deployed to create our website. By hosting our data in an external NoSQL database, MongoDB, we were able to have an app that could do external calls to the server, retrieve and update data using complex queries, and without the need to host it inside our server. We choose MongoDB thanks to its SSL encryption and security features that support our goal of providing a simple and secure medical platform.

The deployment process was done with docker-compose, a multi app docker container manager and sent to AWS Elastic Container Service to host the container.

Through this project, our main contributions were to integrate several layers of security features (e.g.: MongoDB SSL encryption, JSON Web Token (JWT) authentication, recaptcha, etc.) while using a cloud-based hosting, and also use docker in this process to enable our application to work efficiently in different environments and to ensure fluid continuous integration cycles (DevOps).

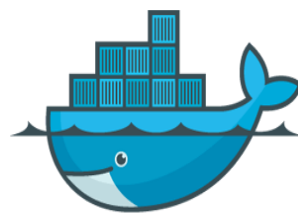
2. PROJECT STACK



JWT



Amazon EC2



docker



POSTMAN

3. RELATED WORK

Medical big data is difficult to share in a secure manner that protects perfectly user privacy (Wang et al, 2018)[2]. Wang et al. studies and analyzes the problems of medical big data sharing and proposes a sharing platform for medical data based on permissioned blockchains, which uses dual-BCs architecture (Account Blockchain and Transaction Blockchain). This system is compared to several other medical data sharing platforms based on blockchain technology, including the classic blockchain system architecture and smart contracts architecture.

In the Exposome Project, Shetty et al (2017)[3] identified the challenges that surround medical data processing and retrieval. The goal was to provide the key optimizations and features of retrieval and selection systems that support efficiently storage of disparate files and unstructured data. Some of these features involve using the MongoDB NoSQL database. The reasons for this choice include its fast capabilities in read / write operations, parallel data access and role-based access control. Furthermore, [4] suggested that MongoDB (or MongoDB Atlas) automatically encrypts all the data and creates SSL encryptions when communicating with the database.

In addition, the management of any medical data must comply with the HIPAA regulations[5]. This means that users will be explicitly told how their data will be managed and what organizations will have access to this data. The position paper from Luo et al (2019)[6] provides more information about some of the regulations that govern the medical data world. [6] presents a brief survey on the state-of-the-art of medical data sharing and management systems. In light of security and privacy compliance regulations such as Health Insurance Portability and Accountability Act (HIPAA)[7] and Health Information Technology for Economic and Clinical Health (HITECH) in United States, or General Data Protection Regulation (GDPR)[8] in Europe, it is a constant challenge to store and share users' data in a way that preserve privacy and security. Failure to do so, may generate severe penalties whenever confidential data is leaked. Thus, protecting medical databases from data breaches is an extremely difficult task that is constrained by rigorous regulations as well as a "tolerance zero" rule for privacy violation. The main obstacles in data sharing according to this position paper are: interoperability, security and privacy. To tackle these pain points, three major approaches are presented: cloud-based approaches, Blockchain-based approaches, Software-Defined Networking-Based Healthcare approaches.

First, in Cloud-based approaches, user trust and data encryption are the main problems. For instance, managing the encryption key can be a burden if pushed under the user's control. Li et al. [9] proposed to use attribute based encryption (ABE) for secure sharing of personal health records stored in semi-trusted cloud servers. Nonetheless, despite patients' full control of their medical information, this configuration pushes to the users the complex task of generating and distributing keys to authorized users.

Second, in the context of blockchain-based approaches, Zyskind et al. [10] suggests to utilize blockchain so as to provide secure and privacy-preserving data sharing among mobile users and service providers. In this model, there are two types of transactions: transaction *Tdata* (for data storage and retrieval), and transaction *Taccess* (for access control). MedRec [11] proposed a decentralized EMR management system based on blockchain technology. Their functional prototype implemented three types of Ethereum smart contracts. The latter enables associating patients' medical information stored in databases of variegated healthcare providers, as well as allowing third-party users to access the data after successful authentication. This research in medical blockchain assumes that medical data is stored in plain-text, in local databases and the blockchain network / smart contracts are responsible for applying data management business rules. There are two main problems with this architecture: (1) plain-text storage increase risks of data breaches, (2) incompatibility with regulations such as GDPR in Europe, which has strengthened the rights of patients to erase their personal information. This demonstrates that storing all medical information in an *immutable* chain is not the best option for security and compliance.

Third, Software-defined networking (SDN) involves decoupling data and control planes. Its goal is to manage the network in a centralized, accelerated and agile fashion, which made it gain more popularity in recent times among network based data management systems. L. Hu et al.[12] proposed a smart health monitoring method with software-defined networking, where a centralized smart controller are designed to manage all physical devices and provide interfaces to data collection, transmission and processing.

After interrogating this prior work, and in the context of the course concepts we learnt so far, we decided to adopt a cloud-based approach, where we use MongoDB as our NoSQL database, for its encryption features and role-based data access. In addition, we employ JSON Web Token (JWT) to control confidential data access by authenticating users, using temporary tokens.

4. TECHNICAL IMPLEMENTATION

4.1 Phase 1: Exploration Phase - Architecture fully AWS

Our initial application development process involved leveraging Amazon's AWS technologies to host our application as well as our development life cycle on AWS servers using the following technologies:

1. S3, an object storage service;
2. AWS Fargate, a container managing service for clusters;
3. Amazon API Gateway, an API management service;
4. AWS Cloud9, a cloud IDE to execute code;
5. AWS CodeCommit, CodePipeline, CodeBuild, a fully-managed source control service.

We were fortunate enough to find an AWS guide that provided the learning path to use these technologies and implement them with ease. The project consisted of creating a modern day web app called Mythical Mysfits. The static pages (HTML, CSS, JS, Media content, etc.) were hosted on Amazon S3 (Simple Storage Service) which is a highly durable, highly available, and inexpensive object storage service. We also used AWS Cloud9 to run every AWS CLI command needed.

In the second part of the modules, we created a new microservice hosted on AWS Fargate that lets our web app integrate with an application backend. Fargate is a deployment option in Amazon Elastic Container Registry that allowed us to deploy containers without managing clusters or servers. We also implemented a Network Load Balancer to diverge the incoming traffic to our website.

For this project, we downloaded a Flask app that contained all of the static and dynamic web pages. We created a docker image using a Dockerfile and then pushed that to AWS's ECR (Elastic Cloud Registry) that contains all of the images of our project. The implementation of the Docker system will be explained in the technical implementation section. We then used S3 to create a bucket for the CI/CD (Continuous Integration / Continuous Delivery) lifecycles and implemented the CodeCommit, CodePipeline and CodeBuild services for CI. The global architecture we executed on AWS is shown in the below figure.

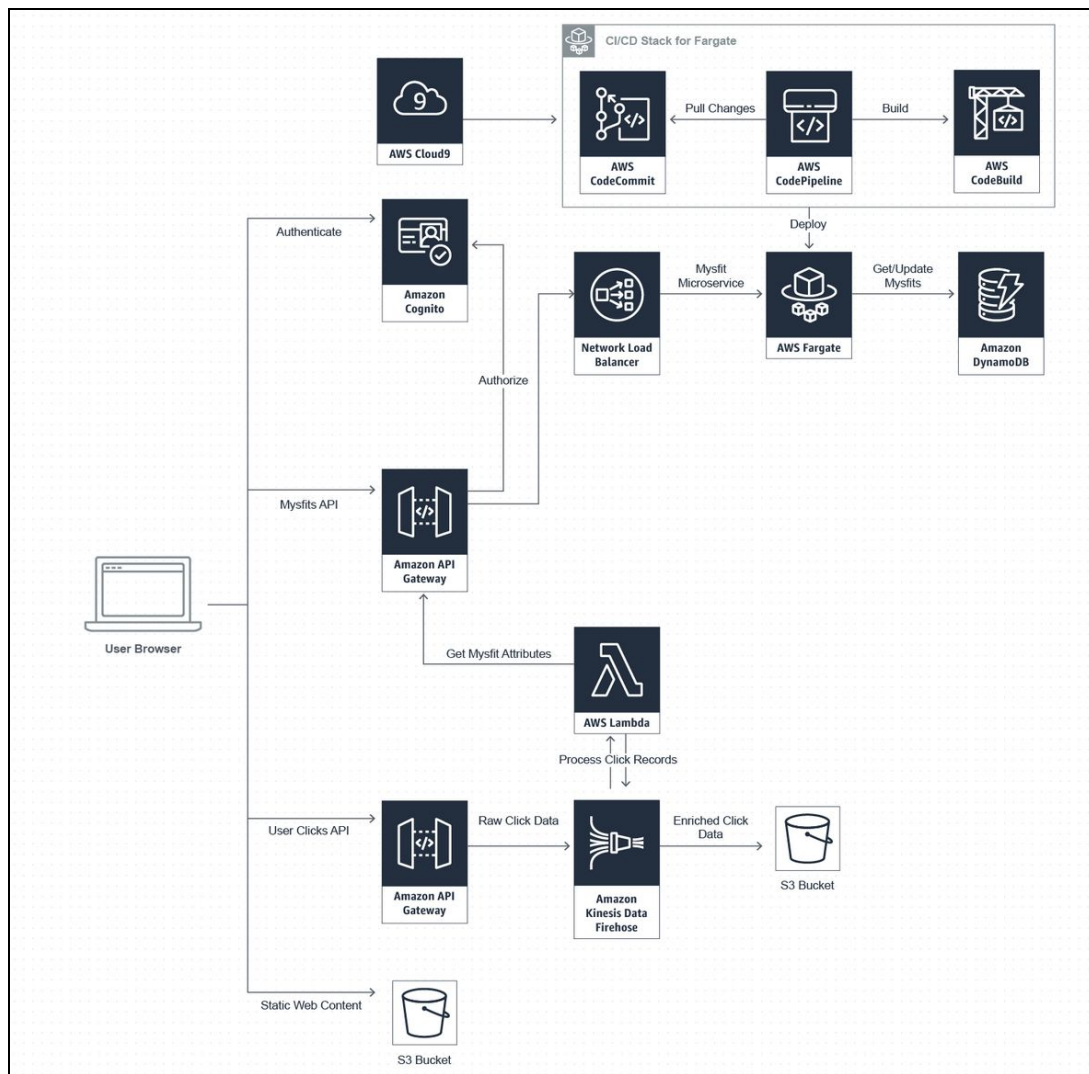


Figure 1: AWS tutorial full web service layout

It was at this phase that we realized that we were implementing most of these services on regular sized AWS servers. In the span of two days, the billing cycle for the account rose to \$6.00 and we were concerned about the possible bill at the end of the project. We took unanimous decision to approach this project in a different way, developing a similar project with less of a cost associated with the services and with a slightly different technology stack that is more suitable to our purpose.

4.2 Phase 2: Implementation Phase - MEAN Stack

4.2.1 Tech Stack

To build our web application, we use the MEAN stack. MEAN is a set of Open Source components that work together to provide an end-to-end framework for building dynamic web applications, from the code running in the browser to the database. The stack is made up of:

- **MongoDB** : Document database built on the NoSQL paradigm – used by the back-end application to store its data as JSON (JavaScript Object Notation) documents;
- **Express** (sometimes referred to as Express.js): Back-end web application framework running on top of Node.js;
- **Angular**: Front-end web app framework which runs our JavaScript code in the user's browser, allowing the application UI to be dynamic;
- **Node.js** : JavaScript runtime environment – where we implement our application back-end in JavaScript.

The most beneficial thing about using the MEAN stack was that the entire application development was in one language, Javascript, which makes it easier to work across the different levels of development.

Application Development Life Cycle

For the development of this app, we have split it into different steps. These steps include:

1. Developing a connection from NodeJS to MongoDB using Express and Mongoose.
Connection String used :
mongodb+srv://sshjuser:<password>>@cluster0-s1qny.mongodb.net/BloodbankDB?retryWrites=true
2. Implementing API in NodeJS, creating GET, POST, PUT and DELETE APIs
3. Developing the front-end in Angular 7 and routing requests to NodeJS controller
4. Implement the back-end logic to match donors on the basis of compatible blood types
5. Implement geocoding to identify the coordinates of a user, and use distance as a factor in matching donors to recipients
6. Adding a Javascript Web Token to give user authentication and security

measurements for the donors and receivers

7. Hashing the password with salt-secret to enhance password security
8. Adding reCAPTCHA to prevent web-scraping for donor information via bots
9. Packaging the web app in a Docker instance and upload it to be hosted on the AWS Elastic Container Service.

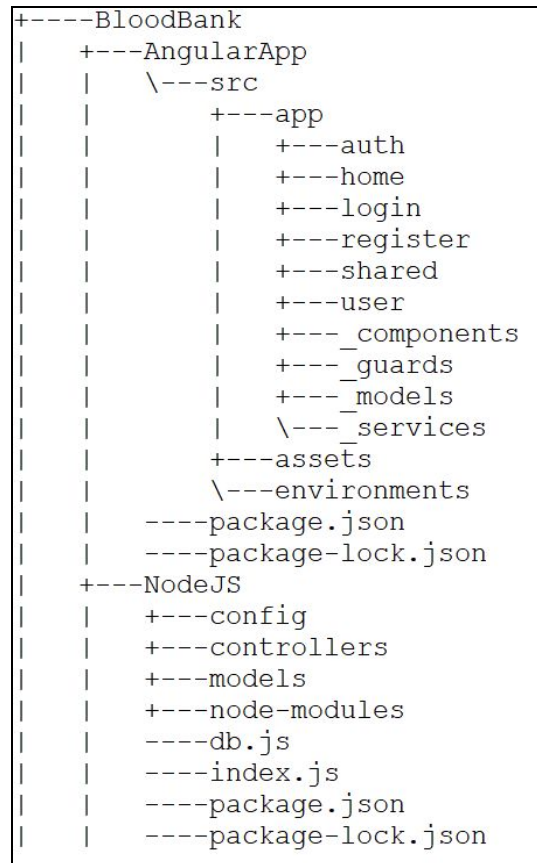


Figure 2: MEAN Stack Application Structure

Database - MongoDB

MongoDB, a cross-platform document-oriented database program, that uses JSON-like documents with defined schemata. The advantages of using a remote hosted MongoDB database for our application are:

High Availability: Since MongoDB provides high-availability on replica sets by default, it helps our application withstand network issues, as well as be a robust provider of information to the blood acceptors whenever there might be an urgency.

1. Querying and Aggregation: Another important benefit of MongoDB for our application is for the ease of querying and complex aggregation functionality that it provides. This is especially useful in our backend logic that calculates the best donor on the basis of matching/compatible blood groups as well as distance (calculated using MongoDB's GeoJSON and Google Map's Geocoding API).
2. The GET and POST functions are optimized to work from an external database. This in turn creates very low latency responses for our project and quick results.

We created a cloud MongoDB instance using <https://cloud.mongodb.com>. The MongoDB cluster was deployed on to AWS (Region = AWS / N Virginia (us-east-1)). Three replicas of the instance are available:

- Primary : ***cluster0-shard-00-01-s1qny.mongodb.net:27017***
- Secondary : ***cluster0-shard-00-00-s1qny.mongodb.net:27017***
- Secondary : ***cluster0-shard-00-02-s1qny.mongodb.net:27017***

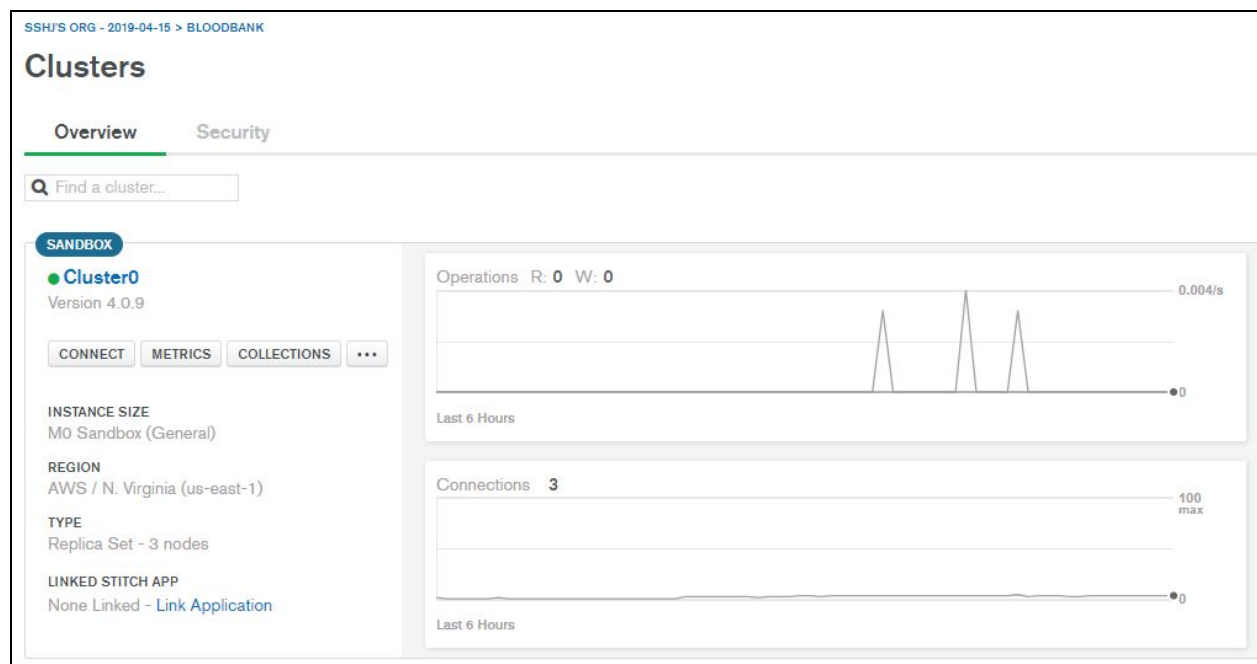


Figure 3: MongoDB Cloud Cluster

Cluster status and metrics can be easily seen using Mongo cloud.

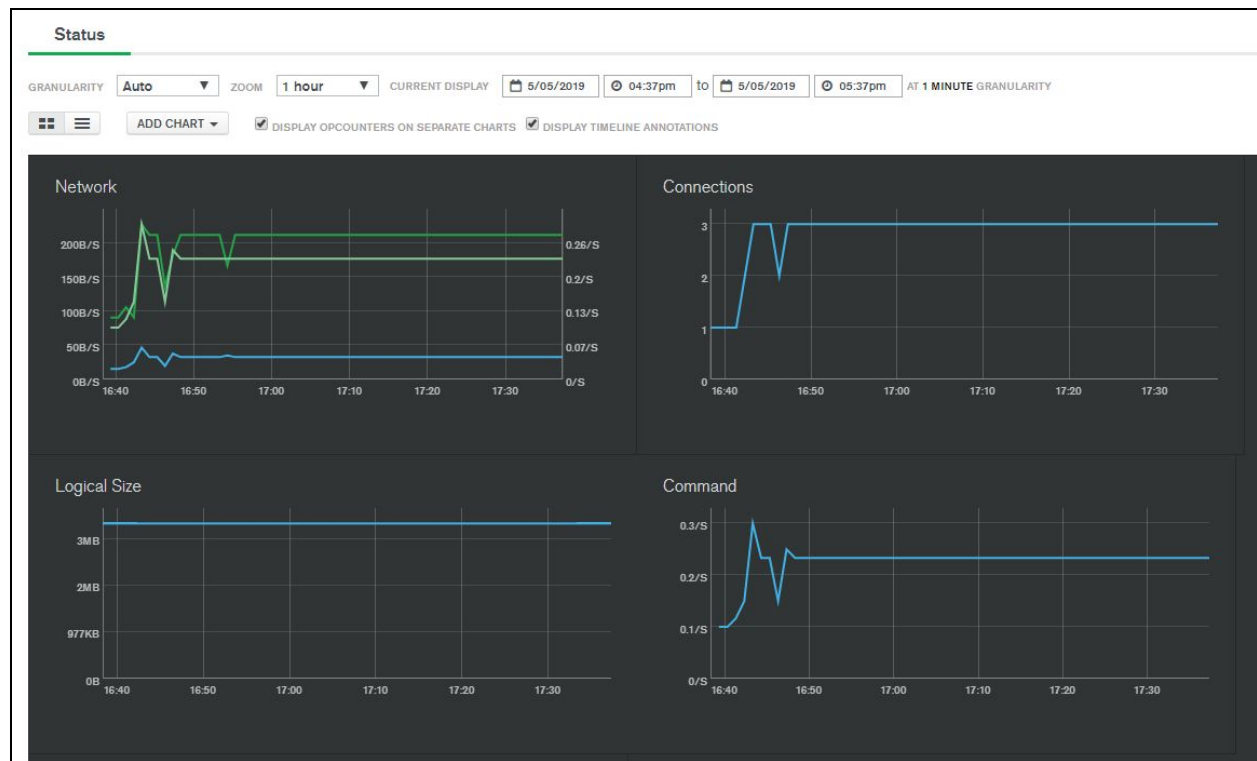


Figure 4: MongoDB Primary Replica Status

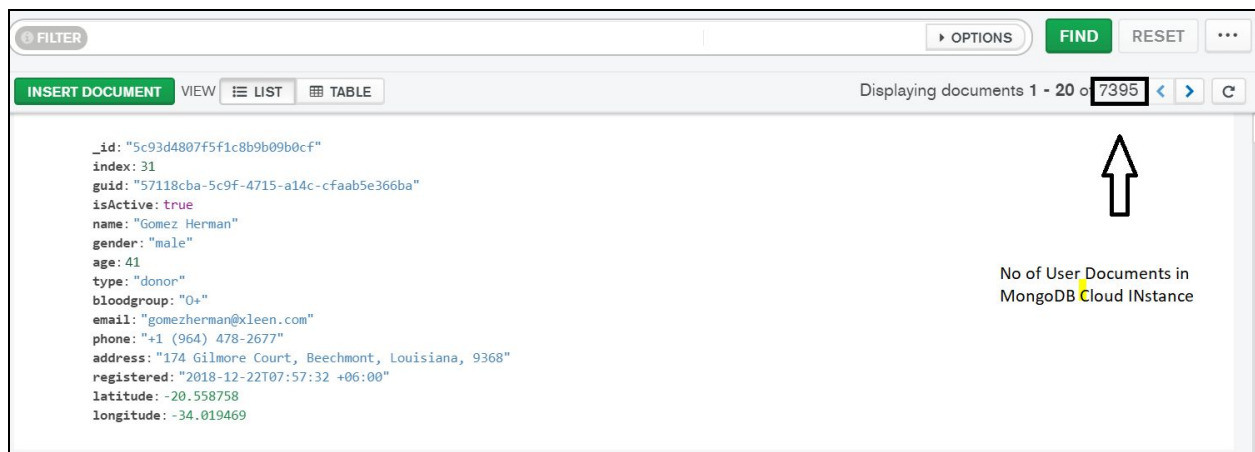
Once the mongo cloud instance was created, a collection named “Users” was created in the database. The initial data load for Bloodbank was created online using json generator available on <https://www.json-generator.com/>.

```
[
  '{{repeat(15000, 7)}}',
  {
    _id: '{{objectId()}}',
    guid: '{{guid()}}',
    isActive: '{{bool()}}',
    firstName: '{{firstName()}}',
    lastName: '{{surname()}}',
    password: '{{firstName()}}{{integer(18, 50)}}',
    email: '{{email()}}',
    gender: '{{gender()}}',
    age: '{{integer(18, 50)}}',
    type: '{{random("donor","recipient")}}',
    bloodgroup: '{{random("AB+", "AB-", "A+", "A-", "B+", "B-", "O-", "O+")}}',
    phone: '+1 {{phone()}}',
    address: '{{integer(100, 999)}} {{street()}} {{city()}} {{state()}} {{integer(100,
```

```
10000}}',
  latitude: '{{floating(-90.000001, 90)}}',
  longitude: '{{floating(-180.000001, 180)}}'
}
]
```

Table 1: JSON data generator code

The above json generator provides the initial dataset required for Mongo cloud instance. The JSON file is inserted to the Mongo Cloud instance using MongoDB Compass. Around 7395 users documents are inserted to MongoDB using Mongo DB Compass.

**Figure 5: User documents in MongoDB Cloud Instance**

Once the data is inserted, the dashboard for admin visualization is enabled which helps in monitoring the users registered.

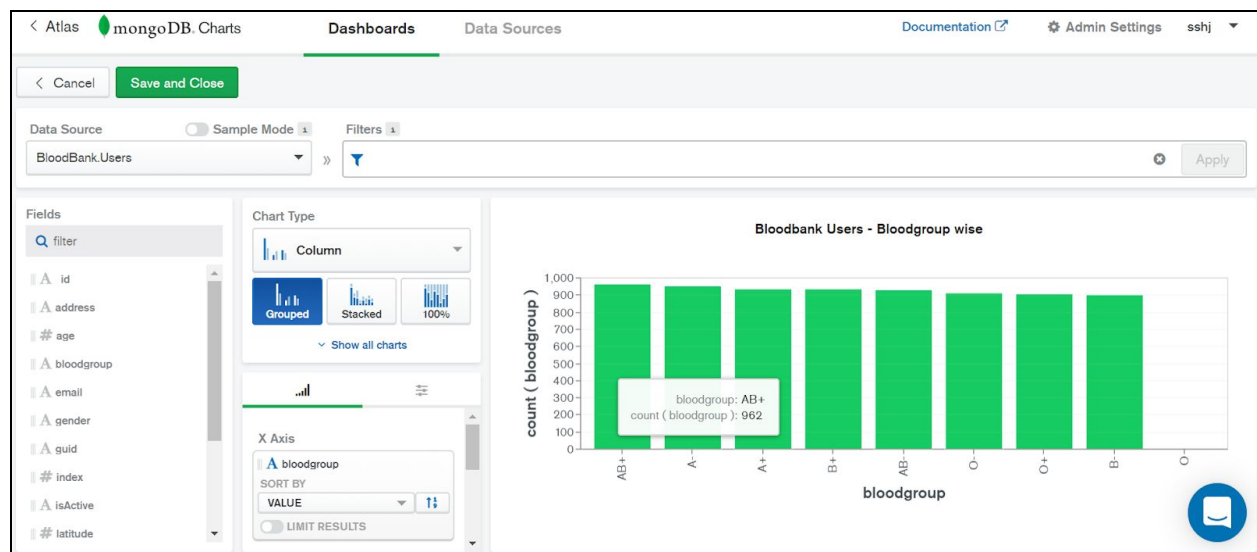


Figure 6: MongoDB Dashboard for Bloodgroup vs Count

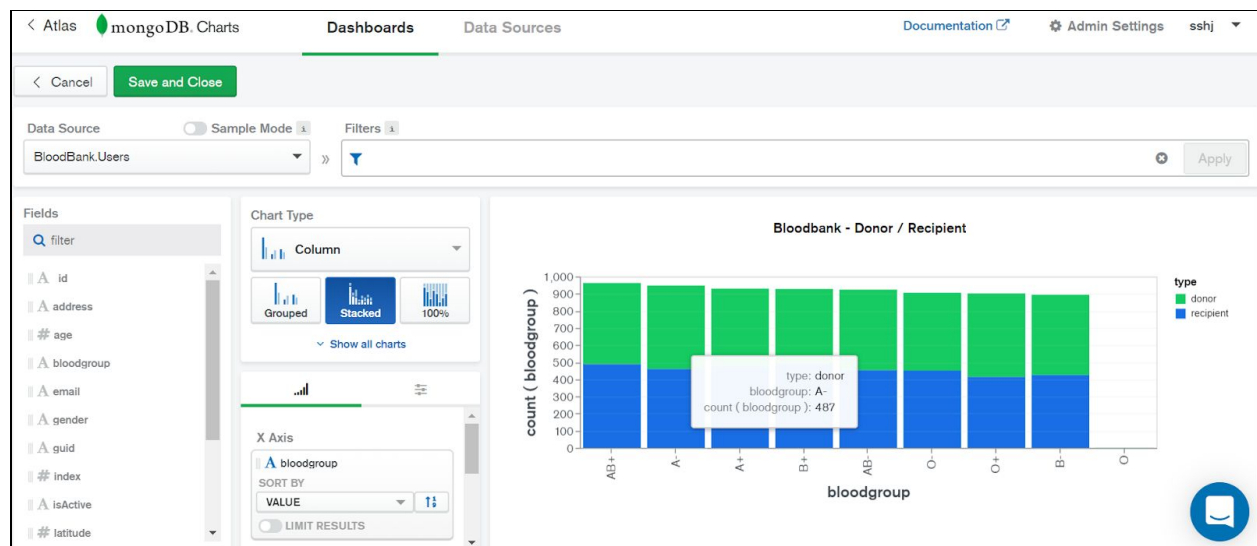


Figure 7: MongoDB Dashboard for Donor/Recipient vs Count

4.2.2 Data model

Data model for Blood bank typically is a json model combination of userSchema which contains all the necessary details of a user (donor/recipient) and geoSchema which includes the latitude and longitude using the address field from user data. The schema is as below:

```

{
  _id: {type: String},
  guid: { type: String },
  isActive: { type: Boolean },
  firstName: { type: String },
  lastName: { type: String },
  password: {type: String},
  gender: { type: String },
  age: { type: Number },
  type: { type: String },
  bloodgroup: { type: String },
  email: { type: String },
  phone: { type: String },
  address: { type: String },
  geometry: {
    type: {type: String},
    coordinates: {type : Array}
  }
}

```

Table 2: Schema

Similarly, we defined the UserSchema and the GeoSchema in the backend to align with the data model defined in MongoDB:

```

const GeoSchema = new Schema({
  type: {
    type: String,
    default: "Point"
  },
  coordinates: {
    type: [Number],
    index: "2dsphere"
  }
});

```

```

const UserSchema = new Schema({
  guid: { type: String },
  isActive: { type: Boolean },
  name: { type: String },
  gender: { type: String },
  age: { type: Number },
  type: { type: String },
  bloodgroup: { type: String },
  email: { type: String },
  phone: { type: String },
  address: { type: String },
  registered: { type: String },
  latitude: { type: Number },
  longitude: { type: Number },
  geometry: GeoSchema
});

var User = mongoose.model('User', UserSchema);

```

Figure 8: UserSchema and GeoSchema

4.2.3 Application Backend - Logic

Using mongoose, we implemented specific queries to:

1. Enable recipients to retrieve donors with matching bloodgroups \Rightarrow queries 1
2. Enable recipients to retrieve donors nearest to them (max distance set to 100,000 meters) \Rightarrow queries 2

More specifically, we used MongoDB *aggregate* pipeline to implement queries 1 as well as MongoDB GeoJSON to construct queries 2.

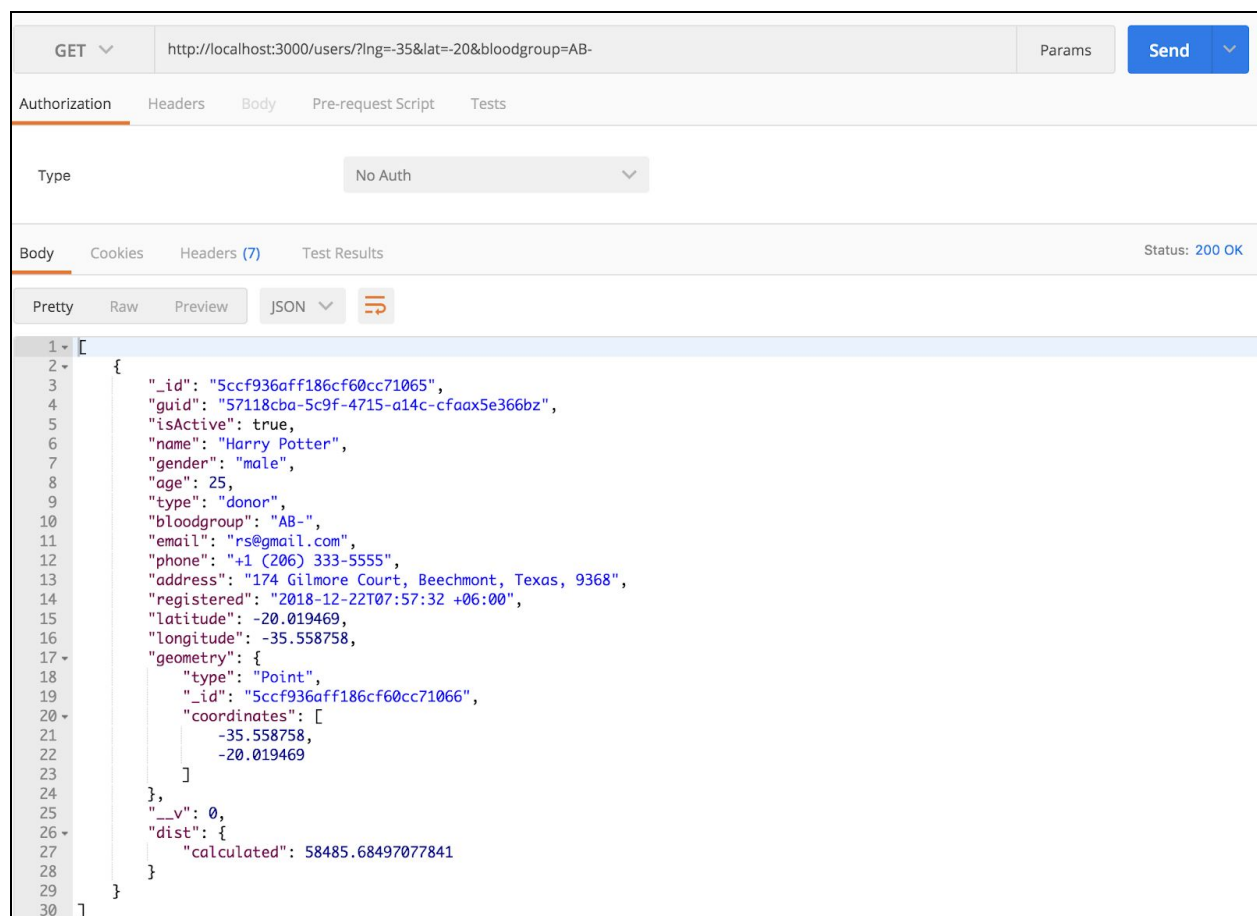


Figure 9: Sample response from the GET function of our application - Result shows nearest neighbors with compatible bloodgroup

Aggregate pipeline

As mentioned in MongoDB aggregation pipeline documentation: “MongoDB’s aggregation framework is modeled on the concept of data processing pipelines. Documents enter a

multi-stage pipeline that transforms the documents into an aggregated result. The most basic pipeline stages provide filters that operate like queries and document transformations that modify the form of the output document. Other pipeline operations provide tools for grouping and sorting documents by specific field or fields as well as tools for aggregating the contents of arrays, including arrays of documents. In addition, pipeline stages can use operators for tasks such as calculating the average or concatenating a string. The aggregation pipeline can operate on a sharded collection."

Our choice to use this pipeline was motivated by its proven efficient data aggregation using native operations within MongoDB. According to other prior work in this field, we found that this is the preferred method for data aggregation in MongoDB. Furthermore, the aggregation pipeline can use indexes to improve its performance during some of its stages, on top of other internal optimization phases.

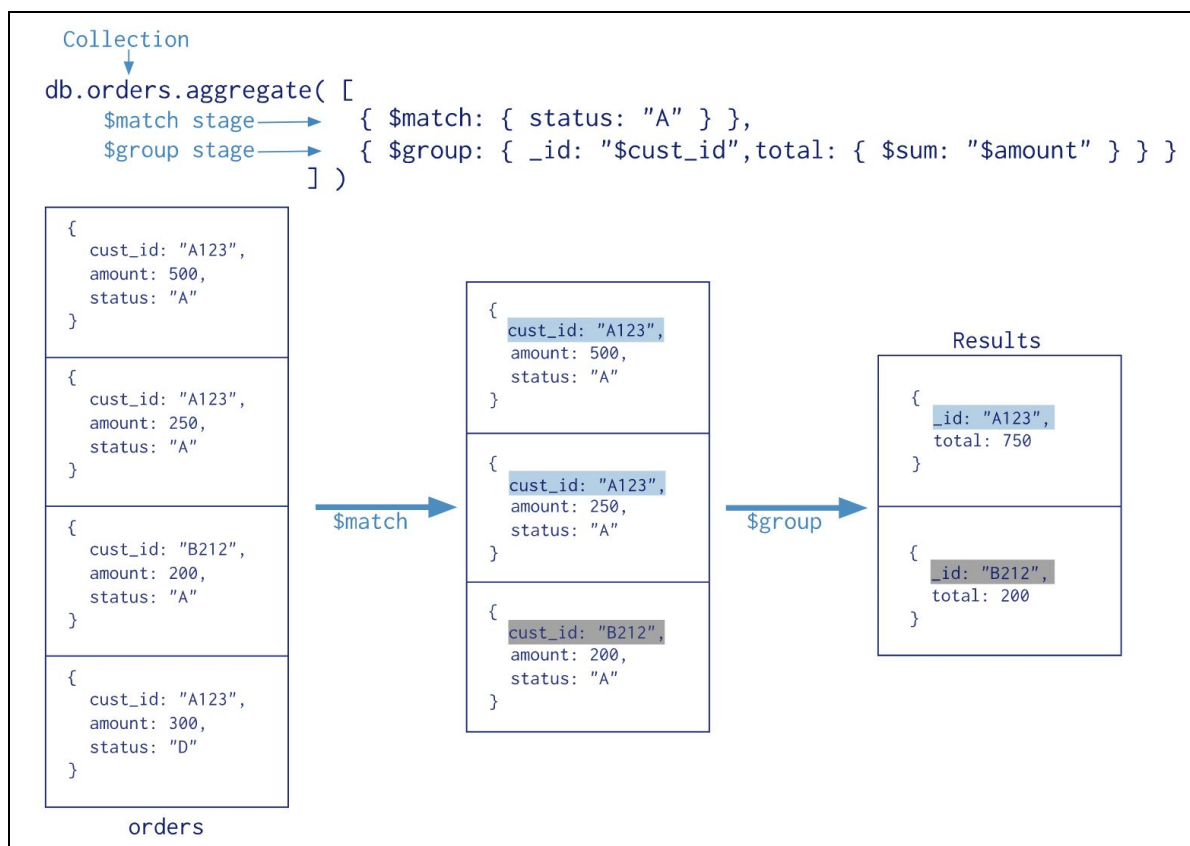


Figure 10: Example of usage of MongoDB aggregation pipeline

Handling Geolocation

MongoDB supports query operations on geospatial data in a special way as MongoDB supports some specific geospatial features. Geospatial data can be stored as GeoJSON objects in MongoDB in order to calculate geometry over an Earth-like sphere. The documentation states: To specify GeoJSON data, use an embedded document with:

- a field named `type` that specifies the GeoJSON object type and
- a field named `coordinates` that specifies the object's coordinates.
- If specifying latitude and longitude coordinates, list the longitude first and then latitude:
 - Valid longitude values are between -180 and 180, both inclusive.
 - Valid latitude values are between -90 and 90 (both inclusive).

MongoDB geospatial queries on GeoJSON objects calculate on a sphere; MongoDB uses the WGS84 reference system for geospatial queries on GeoJSON objects.

The example below, shows how to define a GeoJSON Point. We used a similar object to store the coordinates (longitude, latitude) of each donor / recipient in the database.

```
location: {  
  type: "Point",  
  coordinates: [-73.856077, 40.848447]  
}
```

Figure 11: Example of GeoJSON point, as used in our project

In the geometry schema we defined (see data model sub-section and figure below), we used a 2dsphere index to support queries that calculate geometries on an earth-like sphere. We then used the attribute geometry in the aforementioned aggregation pipeline using the `$geoNear` stage.

```
const GeoSchema = new Schema({
  type: {
    type: String,
    default: "Point"
  },
  coordinates: {
    type: [Number],
    index: "2dsphere"
  }
});
```

Figure 12: GeoSchema as defined in the data model

Longitude, latitude were obtained by converting the address information using Google Geocode API as explain in the next sub-section.

Address Geocoding

We use the Google Geocoding API to get the latitude and longitude coordinates based on the address entered by the user in the 'Register' page on our application. The Geocoding API is a service that provides geocoding and reverse geocoding of addresses.

Geocoding is the process of converting addresses (universal street address) into geographic coordinates (like latitude and longitude), which can be used to place markers on a map, or position the map, or in our case, calculate the distance between two locations to find the closest match.

Required parameters in a geocoding request:

- Address — The street address that you want to geocode, in the format used by the national postal service of the country concerned.
- Components — A components filter with elements separated by a pipe (|). The components filter is also accepted as an optional parameter if an address is provided. Each element in the components filter consists of a component:value pair, and fully restricts the results from the geocoder.
- Key — The application's API key. This key identifies your application for purposes of quota management.

The following is a sample of a geocode API request:

```
https://maps.googleapis.com/maps/api/geocode/outputFormat?parameters
```

For security, we use the https in our call in our application, to prevent sniffing or tampering with the address request.

The response returned is a JSON object (or a list of JSONs, based on the number of matches found), as shown in Figure 11.

```
{
  "results" : [
    {
      "address_components" : [
        {
          "long_name" : "1600",
          "short_name" : "1600",
          "types" : [ "street_number" ]
        },
        {
          "long_name" : "Amphitheatre Pkwy",
          "short_name" : "Amphitheatre Pkwy",
          "types" : [ "route" ]
        },
        {
          "long_name" : "Mountain View",
          "short_name" : "Mountain View",
          "types" : [ "locality", "political" ]
        },
        {
          "long_name" : "Santa Clara County",
          "short_name" : "Santa Clara County",
          "types" : [ "administrative_area_level_2", "political" ]
        },
        {
          "long_name" : "California",
          "short_name" : "CA",
          "types" : [ "administrative_area_level_1", "political" ]
        },
        {
          "long_name" : "United States",
          "short_name" : "US",
          "types" : [ "country", "political" ]
        },
        {
          "long_name" : "94043",
          "short_name" : "94043",
          "types" : [ "postal_code" ]
        }
      ],
      "formatted_address" : "1600 Amphitheatre Parkway, Mountain View, CA 94043, USA",
      "geometry" : {
        "location" : {
          "lat" : 37.4224764,
          "lng" : -122.0842499
        },
        "location_type" : "ROOFTOP",
        "viewport" : {
          "northeast" : {
            "lat" : 37.4238253802915,
            "lng" : -122.0829009197085
          },
          "southwest" : {
            "lat" : 37.4211274197085,
            "lng" : -122.0855988802915
          }
        }
      },
      "place_id" : "ChIJ2eUgeAK6j4ARbn5u-WAGqWA",
      "types" : [ "street_address" ]
    }
  ],
  "status" : "OK"
}
```

Figure 13: JSON response from Geocoding API

In our application, we take the user's address, and on submission of the form, called the

'geocode' function that calls the geocoding API. The response is recorded, and the top result from the response is stored. The latitude and longitude are extracted from there, and the geometry field in the user objects in the mongoDB is populated.

```

{...}
  config: Object { timeout: 0, xsrfCookieName: "XSRF-TOKEN", xsrfHeaderName: "X-XSRF-TOKEN", ... }
  data: {...}
  results: (2) [...]
    0: {...}
      address_components: Array(7) [ {...}, {...}, {...}, ... ]
      formatted_address: "302 Ball St, College Station, TX 77840, USA"
      geometry: {...}
        location: Object { lat: 30.6263146, lng: -96.34321299999999 }
        location_type: "ROOFTOP"
        viewport: Object { northeast: {...}, southwest: {...} }
        <prototype>: Object { ... }
      place_id: "ChIJMwpeUpwDRoYRnV0ik-Hicq4"
      plus_code: Object { compound_code: "JMG4+GP College Station, Texas, United States", global_code: "8625JMG4+GP" }
      types: Array [ "street_address" ]
      <prototype>: Object { ... }
    1: Object { address_components: (7) [...], formatted_address: "302 Ball St, Tom Bean, TX 75489, USA", place_id: "EiQzMDIgOmFsbCBTdCwgVG9tIEJlYW4sIFRYIDc1NDg5LCBVU0EiGxIZChQKEglpGIPbXhIMhhGOKb0ZxUAAdAhCuAg", ... }
      length: 2
      <prototype>: Array []
    status: "OK"

```

Figure 14: Sample response from our application for address entered

4.3 Security

Authentication is an important concern for any web application. It is essential to identify user rights. Encryption, authentication, session tokens are few of the security paradigms implemented in this project.

4.3.1 CORS Policy

CORS stands for Cross-origin resource sharing. Enabling CORS requires to install cors using: `npm install --save cors`

Access-Control-Allow-Origin setting allows any of the origin to connect.

Access-Control-Allow-Origin: *

However, setting the CORS origin to Angular client enables accessing the NodeJS API only via Angular application. This ensures API security.

```
app.use(cors({
  origin: 'http://localhost:4200'
}));
```

Figure 15: CORS Logic

4.3.2 Hashing and Salting

Cryptographic hash functions take a string as input and returns another hashed string which represents the same information in an encoded form. On the other hand, salt values are random data that is included with input for the hash function. Adding salt to hashing process, increases password security without much increase in complexity. It also helps in lowering the vulnerabilities due to password storage on cloud. Hashed passwords are deterministic in nature, same password always hashes to same string. A salt is actually a fixed-length random value which is cryptographically strong. Bcrypt stands for Blowfish cipher and crypt for the hashing function.





				
Password	p4s5w3rdz	p4s5w3rdz	p4s5w3rdz	p4s5w3rdz
Salt	-	-	et52ed	ye5sf8
Hash	f4c31aa	f4c31aa	1vn49sa	z32i6t0

Figure 16: Hashing with salt

While user registration the string password entered is converted to hash using bcrypt before storing the database.

```

userSchema.pre('save', function (next) {
  bcrypt.genSalt(10, (err, salt) => {
    bcrypt.hash(this.password, salt, (err, hash) => {
      this.password = hash;
      this.saltSecret = salt;
      next();
    });
  });
});

```

Figure 17: Hashing with Salt logic



```

_id: ObjectId("5cce761ee82cf6c8a8a39422")
isActive: true
firstName: "Somya"
lastName: "Sharma"
password: "$2a$10$il7e23ZYK8AGEn2r5.63ZevSYdLIFiuS21HAWPfeCmT9aXZFYIE30"
gender: "female"
age: 24
type: "recipient"
bloodgroup: "B+"
email: "somya@tamu.edu"
phone: "7374001475"
address: "3645 Wellborn Road"
saltSecret: "$2a$10$il7e23ZYK8AGEn2r5.63Ze"
__v: 0

```

Hashed Password

Salt Secret

Figure 18: Hashed password and salt secret inserted in MongoDB

When login function is invoked, a post request is made to the 'authenticate' Node JS API by passing the email and the password. The string password is encoded using bcrypt and compared to the password stored in the Mongo DB. The callback function returns a token for specified time.

4.3.3 Implementing JWT authentication

JWT authentication is becoming very popular day-by-day. JWT is a type of token based authentication which supports stateless APIs. JSON web tokens are used in the client-server authentication enabling information share. Below screenshots, depict the

jwt authentication calling the NodeJ4s API directly. When a user tries to access the home page without logging in, the access would be forbidden.

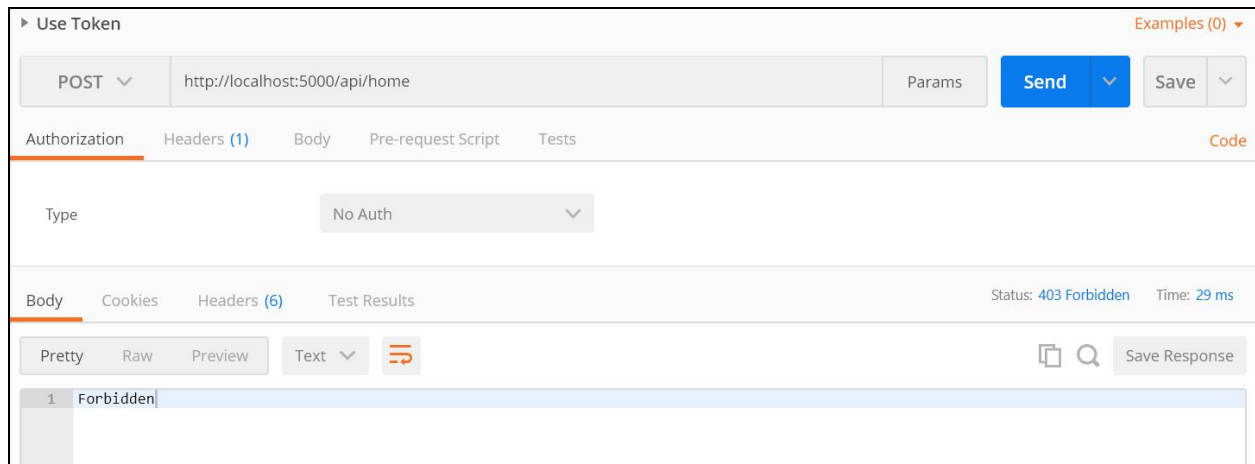


Figure 19: Access forbidden before JWT

Only when a user logs in, a token is generated for a limited amount of time. This time is set to 5 mins (which is also the session timeout).

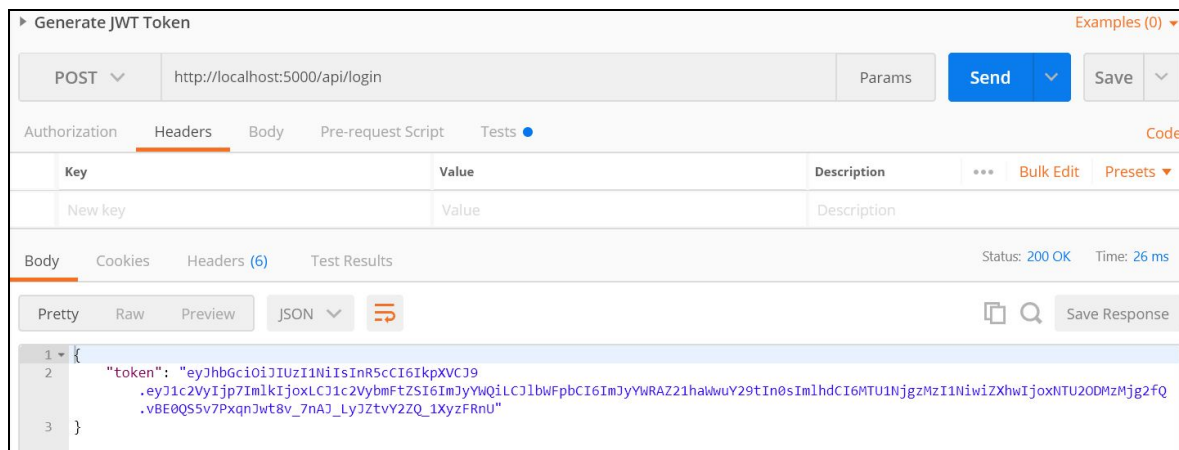


Figure 20: Generating JWT Token

When the home page is accessed using JWT token it enables to access the user data.

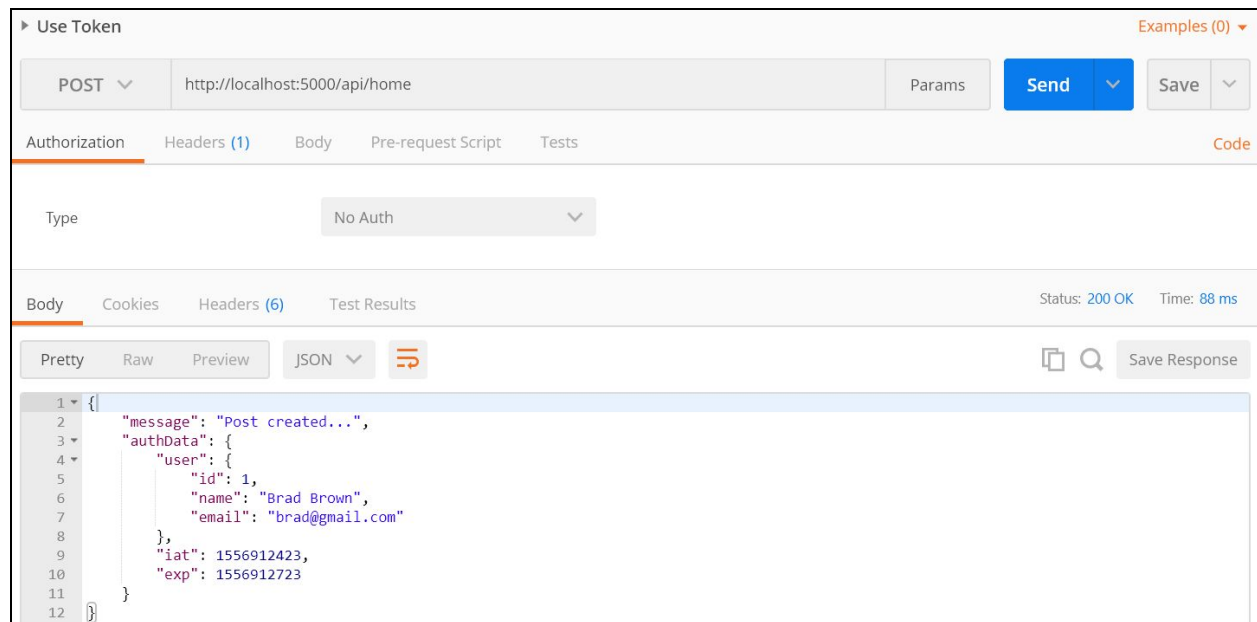


Figure 21: Access after JWT Token

Package “jsonwebtoken” is utilized to implement JWT authentication.

```
userSchema.methods.generateJwt = function () {  
  return jwt.sign({ _id: this._id},  
    process.env.JWT_SECRET,  
    {  
      expiresIn: process.env.JWT_EXP  
    })  
};
```

Figure 22: JWT generation logic

The config.json file is configured as below in Figure 20.


```
{
  "development": {
    "PORT": 3000,
    "MONGODB_URI": "mongodb
+srv://sshjuser:<<password>>@cluster0-s1qny.mongodb.net/BloodbankDB?
retryWrites=true",
    "JWT_SECRET": "SECRET#123",
    "JWT_EXP": "10m"
  }
}
```

Figure 23: JWT configuration parameters

4.3.4. Implementing reCAPTCHA

Recaptcha provides protection from spam and abuse. Using Google Recaptcha v2 site key and secret key are generated.

Use this site key in the HTML code your site serves to users. [See client side integration](#)

 COPY SITE KEY

Use this secret key for communication between your site and reCAPTCHA. [See server side integration](#)


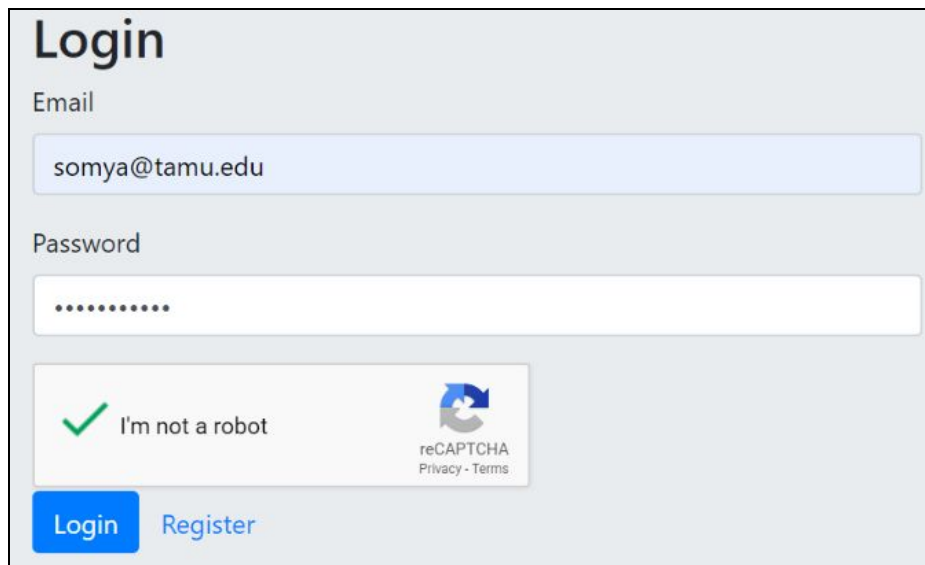
 COPY SECRET KEY

Figure 24: Google reCAPTCHA generating Site Key and Secret key

```
<re-captcha (resolved)="resolved($event)"
siteKey="6LcYtqEUAAAAAHyRSxGe_YQlZJ3YMSbKsDeahXlw"></re-captcha>
```

Figure 25: reCAPTCHA UI logic



Login

Email

somya@tamu.edu

Password

.....

☒ I'm not a robot

reCAPTCHA
Privacy - Terms

Login Register

Figure 26: reCAPTCHA

4.4 Containerizing the App

Docker is a program that performs operating-system-level virtualization. Docker is a very lightweight virtualization protocol which creates images out of any application configuration, environment configuration and operating systems functions and then runs these images in isolated containers on the OS of choice. These containers can communicate with each other through specified communication ports and allow each application to run in its own environment. In our project, we can create containers for each separate application in the MEAN stack. By creating separate applications, we can use Docker-Compose, a multi container synchronization program, to orchestrate the initialization and run time of the containers.

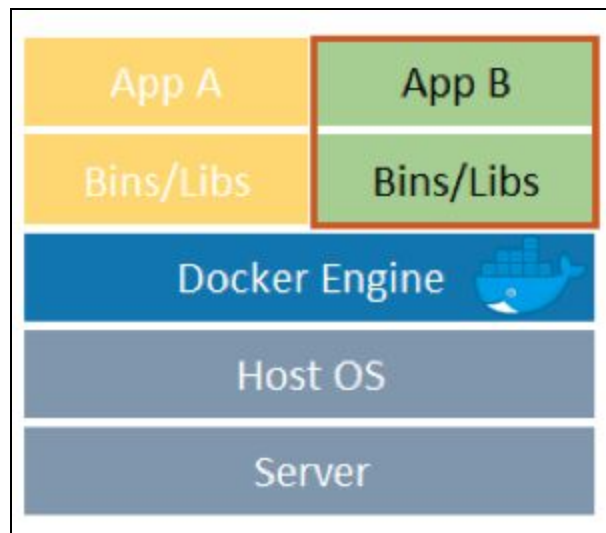


Figure 27: Docker containers and Virtualization stack

By creating two separate Dockerfiles, each in their respective app, we created a container for the Angular app and then another one for the Express & Node.js backend application. Using a docker-compose file, the two containers were connected and both instantiated when the containers were run.

The Dockerfile contains the steps needed to containerize the app. For example, this is the dockerfile for the Angular application:

```
1 FROM node:latest
2 RUN mkdir -p /app
3 WORKDIR /app
4
5 COPY package*.json /app/
6
7 RUN npm install -g
8
9 COPY . /app/
10
11 EXPOSE 4200
12
13 CMD ["npm", "start"]
14
```

Figure 28: Example Dockerfile setup

This file downloads the node distro from the dockerhub repository, installs the required dependencies from the package.json file that our app requires, copies over our app, open and exposes a port for communication and then runs the “npm start” command.

The second dockerfile looks very similar for the NodeJS and Express backend, but exposes a different port for communication.

By using docker-compose, we can run a simple command that connects both of the containers and creates a communication between the two.

4.5 Hosting the App

Amazon has a great and easy process for hosting a Docker-Compose collection. By using the AWS Elastic Cloud Service to create an ubuntu instance, we were able to set a public DNS route and then use the AWS CLI to push the docker-compose multi stack containers to the instance for public hosting.

The configuration that was set for ECS to host the app were: Ubuntu 64bit 18.04 using a t2.micro instance which is set up to run on 1 core and 613 MB of memory. This tier was still in the free range as long as it was under 750 hours of use. The second configuration set was an Elastic Load Balancer IP Address that was limited to 750 hours of use or 15 GBs of data processing. For the instance, a separate VPC, Virtual Private cloud, and a Subnet were set up to create a public DNS service for outside contact to the app. Specific options were set to allow the VPC to allow outside HTTP access using a default port number. The region for this instance was set in us-east-2 which is located in Ohio, USA. This Infrastructure as a Service has been designed to be very robust in case of a horizontal expansion later in the future.

The screenshot shows a web application interface for a Blood Bank. On the left, there is a form to add new users with the following fields:

- Name:** Enter full name
- Email:** Enter Email Id
- Phone:** Enter Phone number
- Age:** Enter Age
- Address:** Enter Address
- Bloodgroup:** (Dropdown menu)
- Gender:**
 - ☐ Male
 - ☐ Female
- Type:**
 - ☐ Donor
 - ☐ Recipient

At the bottom of the form are 'Submit' and 'Reset' buttons. On the right, there is a table of existing users:

Name	Email	Phone	Age	Gender	Type	Active
Jane Doe	sarra.bounouh@gmail.com	+1 (206) 333-5555	48	female	donor	true
Henry Quan	hr@gmail.com	+1 (206) 333-5555	25	male	donor	true
New user	new@gmail.com	890	12	male	recipient	true
Somya Rajkumar Sharma	somya@tamu.edu	7374001475	23	female	recipient	true
John Doe	jd@gmail.com	1234567890	47	male	donor	
Marion Cotillard	mc@gmail.com	1234567890	32	female	donor	
sinatra	sinatra@gmail.com	1234567890	56	male	recipient	true
michael bubble	mb@gmail.com	1234567890	36	male	recipient	true
Henry Quan	hr@gmail.com	+1 (206) 333-5555	25	male	donor	true
Sean Miller		1234567890	34	male	donor	true
Sarra Bounouh	sarra.bounouh@gmail.com	2063275375		male	recipient	true
Janvi Palan	janvipalan@tamu.edu	2405798	25	female	donor	true
Janvi Palan	janvipalan@tamu.edu	2405798590	21	female	recipient	true
Janvi C Palan	janvipalan@tamu.edu	2405798590			recipient	true
Rahil Parikh	janvipalan@tamu.edu	2405798590	25	male	donor	true

Figure 29: Blood Bank hosted on AWS ECS

5. EVALUATION

5.1 Data loading performance

We performed some data loading tests in MongoDB to measure its performance. The summary of the results are shown below:

# of documents loaded to MongoDB	Time taken in ms
1,000	1470 ms
10,000	3670 ms
100,000	4709 ms

Table 3: Data loading performance

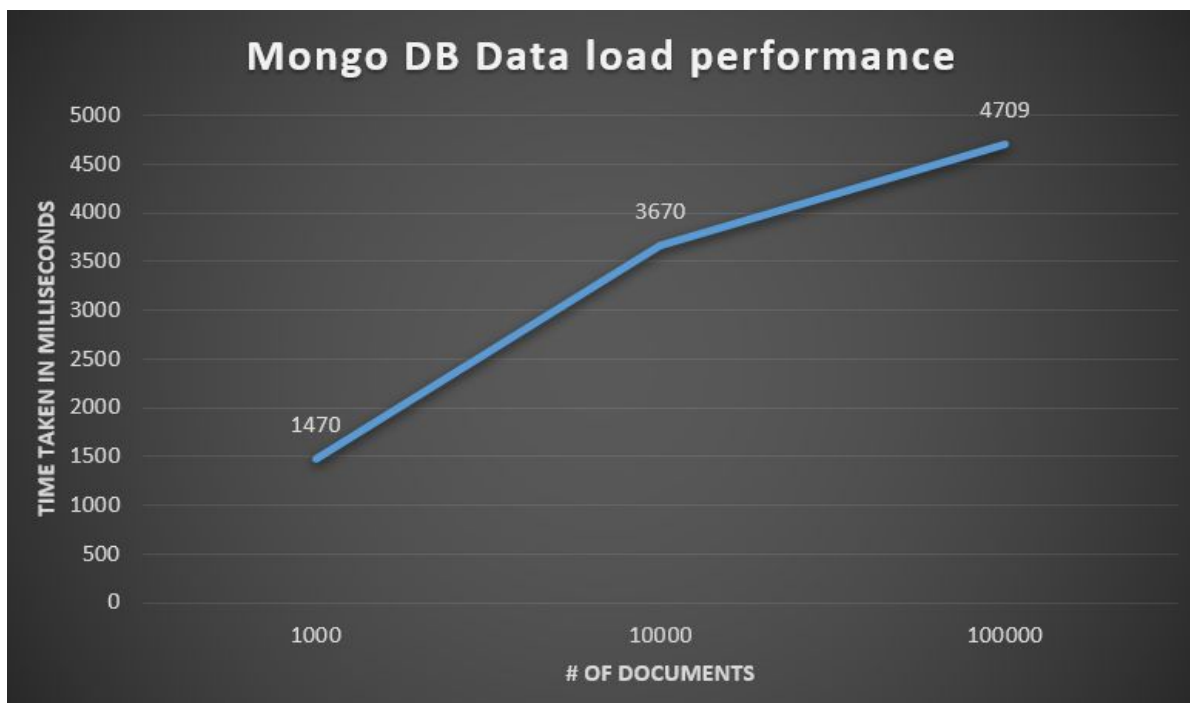


Figure 30: MongoDB data load performance graph

5.2 Security evaluation

We present the password security evaluation below. We consider the scenario where three people John Smith, John Doe and John Brown use the same password “john123”. Though the password is same for the three users the password hash computed along with salt secret is different. Hence, this makes it really difficult to guess or brute force the password. Once these users are registered, we can filter the records based on firstname as “John” in MongoDB Compass we get the below response:



The screenshot shows the MongoDB Compass interface with the 'users' collection filtered by 'firstName: "John"'. It displays three records, each with a unique ObjectID, firstName, lastName, password, and saltSecret.

	_id ObjectId	firstName String	lastName String	password String	saltSecret String
1	5cce7767e82cf6c8a8a39423	"John"	"Brown"	"\$2a\$10\$Jl9ouuIi.iY1tcx2EoeStu5"	"\$2a\$10\$Jl9ouuIi.iY1tcx2EoeStu"
2	5ccf3d41f35c56ca0c2fb810	"John"	"Smith"	"\$2a\$10\$TjtVEL5dZQDjmJ0uzPkVBej"	"\$2a\$10\$TjtVEL5dZQDjmJ0uzPkVBe"
3	5ccf3e4cf35c56ca0c2fb811	"John"	"Doe"	"\$2a\$10\$T5B7wo1YVp2Z4jiF/KrhTuN"	"\$2a\$10\$T5B7wo1YVp2Z4jiF/KrhTu"

Figure 31: MongoDB records

In the above figure, we can see the hashed password and salt secret. Below table shows the clear mapping of how the same password is hashed to different values with the use of salting.

	User 1 - John Smith	User 2 - John Doe	User 3 - John Brown
Password	john123	john123	john123
Password Hash	\$2a\$10\$TjtVEL5dZQDjmJ0uzPkVBejTeiVzEDmMvpdafiQ2QQp.kui8M8rMe	\$2a\$10\$T5B7wo1YVp2Z4jiF/KrhTuNEL2KO4lVpoLpdPO7Etn10B7a69OEua	\$2a\$10\$Jl9ouuIi.iY1tcx2EoeStu54dvvdGbTndgjdYIIAgTFUfsurbiee
Salt Secret	\$2a\$10\$TjtVEL5dZQDjmJ0uzPkVBe	\$2a\$10\$T5B7wo1YVp2Z4jiF/KrhTu	\$2a\$10\$Jl9ouuIi.iY1tcx2EoeStu

Table 4: Security Evaluation - Hashing with Salting

5.3 MongoDB and CAP Theorem

From a benchmarking of NoSQL databases, we found out that MongoDB maintains consistency and partition tolerance, meanwhile, other services, such as DynamoDB, ensures availability and partition tolerance. In the context of our application, we do not want to give “false hopes” to the recipients, therefore we think that consistency is more important in this context. This reinforces our decision of using MongoDB in the context of a healthcare app.

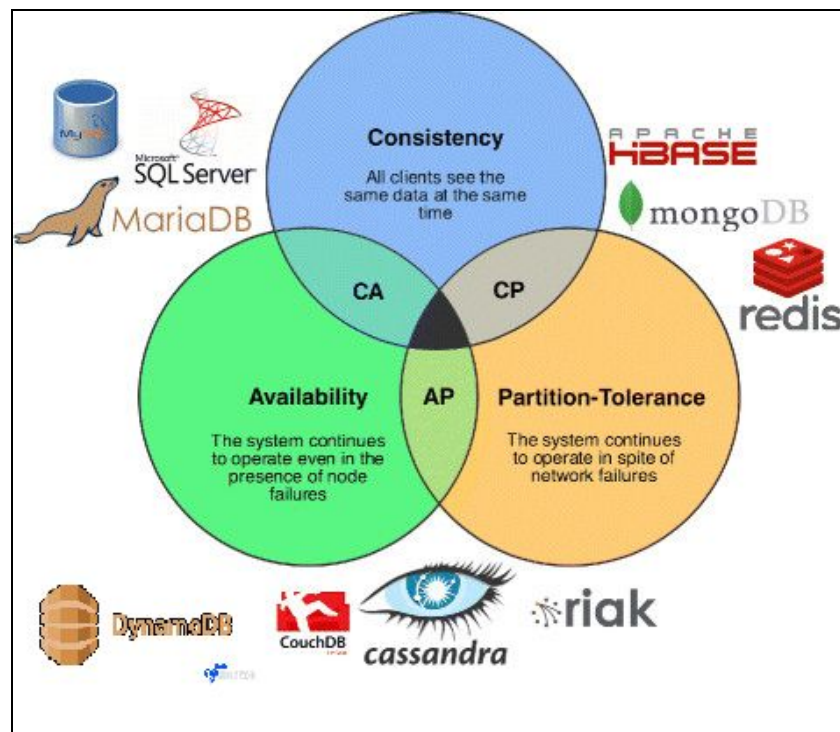


Figure 32: CAP Theorem - Storage Options

6. DISCUSSION

6.1 Data security

As mentioned in the related work section, medical data storage and sharing can pose a huge challenge to researchers and healthcare providers. Medical data is usually extremely confidential and highly regulated. Therefore, securing medical records is a priority in information systems and architectures that encompasses them. There has been some research done to invest in specialized software to prevent unauthorized access to this data. In this project, we selected MongoDB as the preferred NoSQL database for its inherent security and privacy capabilities, i.e. no need to integrate additional pieces of software that add more complexity to the system architecture. MongoDB supports complex structures, and also operates over a wide variety of access patterns and data types [13]. Moreover, several complex queries can be implemented on MongoDB using *mongoose* framework.

Furthermore, prior work demonstrated that MongoDB proved faster in reading a records as compared to Apache Cassandra (16 seconds versus 43 seconds) [14]. Data loading time also proved to be less as compared to Apache Cassandra (45 seconds versus 59 seconds) [14]. We also showed in our evaluation, that loading 100,000 of user records, could be done in 4709 ms. Moreover, the graph plotted [Figure 28] for data loading clearly shows the relation between data load and time taken.

Additionally, the measures taken for password protection for User data on cloud using hashing with salting using Bcrypt. Bcrypt has significant advantage over simple salted SHA-256. Although salting increases the latency during the hashing computation, it really makes it difficult for a brute-forcing attack. "*`bcrypt` was designed for password hashing hence it is a slow algorithm. This is good for password hashing as it reduces the number of passwords by second an attacker could hash when crafting a dictionary attack.*" [16]

MongoDB also supports:

- **Authorization:** This mechanism decides which database resources and operations are accessed by the users. MongoDB provides nine different access control roles that can be implemented step by step, according to the database

policy required by the researcher or the healthcare provider.

- **Data modelling:** Efficient and time-critical retrievals are also dependent on the way data is stored in the database. Hence choosing the correct data model is essential. Our application establishes a simplified data model of Users and Geolocalization information. Using a combination of NoSQL features along with relational mapping also ensures that already existing systems can also be migrated to the developed system.

6.2 Issues and challenges

There were a few issues we faced by developing the apps on different development environments. We used a mixture of Mac OS and Linux. Together with slight variations in the versions of Angular and Node, the development and deployment process took longer than expected, once the issue faced was in the docker container creation. The code repository included a few libraries that were not compatible with the node image pulled from the DockerHub repository. This led to a manual task of finding different dependencies that caused issues in the dockerization process. This issue propagated into the docker-compose portion of the project.

Hosting the docker-compose file provided a few more challenges, but ultimately we decided to use ECS to create an instance that could host the containers. In this form of virtualization, pushing docker images to be hosted on the AWS repository was effortless and by setting up a public DNS, we could access the site relatively quickly. The website is no longer hosted due to financial concerns.

Another challenge that we faced during the project is integrating all the components. This project put us in a real-world situation of the industry, where several software engineers work on a large project. Even while using version control (Git), we had to resolve some merging conflicts manually in order to integrate some branches. We learnt that in the future we have to commit small changes rather than substantial ones so as to decrease the effort of merging.

7. CONCLUSION

Our MEAN stack application has been developed with security and medical data sensitivity in mind. Special care has been taken to include authentication security through the JWT protocol and re-captcha to prevent abusive traffic on the server. By using an external MongoDB server that includes SSL encryption for the data and authentication security, we can protect HIPAA approved medical data.

We saw in the related work section that there are several approaches to tackle the challenging problem of medical data storing and sharing:

- Cloud-based approaches,
- Blockchain-based approaches,
- Software-Defined Networking-Based Healthcare approaches.

In the context of this project we chose to use the first strategy, not only because we were curious about some of the Cloud technologies we employed, but also because prior work demonstrated that blockchain is not the optimal solution if we want to be compliant with healthcare regulations and reduce data breaches, and Software-defined networking (SDN) is more advanced than the purpose of our project.

In the future, we could augment our system with more security and encryption features to make the application compliant with healthcare regulation. In addition, recent research in the field of medical data systems has been focusing on seamlessly integrating attribute-based encryption, privacy level classification, blockchain technology, and software-defined networking (SDN) to achieve secure and privacy-preserving sharing of clinical information. This constitutes other possible evolutions to bring to our system. In particular, we could attempt to use the SDN approach on top of our application, as a cloud strategy to tackle network management and efficient configuration that would improve network performance and monitoring.

8. REFERENCES

- [1] 2018. Biggest Healthcare Data Breaches 2018 So Far
<https://www.healthcareitnews.com/projects/biggesthealthcare-data-breaches-2018-so-far>.
- [2] Rong Wang, Wei-Tek Tsai, Juan He, Can Liu, Qi Li, Enyan Deng, A Medical Data Sharing Platform Based On Permissioned Blockchains, December 2018, Proceeding ICBTA 2018 Proceedings of the 2018 International Conference on Blockchain Technology and Application, pp. 12-16
- [3] Roshan Ramprasad Shetty, Akalanka Mailewa Dissanayaka, Susan Mengel, Lisa Gittner, Ravi Vadapalli, Hafiz Khan, Secure NoSQL Based Medical Data Processing and Retrieval: The Exposome Project, 2017, Proceeding UCC '17 Companion Companion Proceedings of the 10th International Conference on Utility and Cloud Computing, pp. 99-105
- [4] Ryan P Taylor et al. Consolidation of cloud computing in Atlas 2017 J. Phys Conf. Ser. 898 052008
- [5] 2004. Mercuri, R. T. The HIPAA-potamus in health care data security. Communications of the ACM, Vol 47, No.7, pp25-28
- [6] Yan Luo, Hao Jin, Peilong Li, A Blockchain Future for Secure Clinical Data Sharing: A Position Paper, March 2019, SDN-NFVSec '19 Proceedings of the ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization, pp. 23-27
- [7] 2017. Summary of the HIPAA Security Rule. <https://www.hhs.gov/hipaa/for-professionals/security/laws-regulations/>
- [8] 2016. General Data Protection Regulation. <https://eugdpr.org/the-regulation/>
- [9] Ming Li, Shucheng Yu, Yao Zheng, Kui Ren, and Wenjing Lou. 2013. Scalable and secure sharing of personal health records in cloud computing using attribute-based encryption. IEEE transactions on parallel and distributed systems 24, 1 (2013), 131–143.
- [10] Guy Zyskind, Oz Nathan, et al. 2015. Decentralizing privacy: Using blockchain to protect personal data. In Security and Privacy Workshops (SPW), 2015 IEEE. IEEE, 180–184.

- [11] Asaph Azaria, Ariel Ekblaw, Thiago Vieira, and Andrew Lippman. 2016. Medrec: Using blockchain for medical data access and permission management. In Open and Big Data (OBD), International Conference on. IEEE, 25–30.
- [12] Long Hu, Meikang Qiu, Jeungeun Song, M Shamim Hossain, and Ahmed Ghoneim. 2015. Software defined healthcare networks. IEEE Wireless Communications 22, 6 (2015), 67–75.
- [13] Veronika Abramova and Jorge Bernardino. 2013. NoSQL databases: MongoDB vs cassandra. In Proceedings of the International C* Conference on Computer Science and Software Engineering (C3S2E '13), Ana Maria Almeida, Jorge Bernardino, and Sudhir Mudur (Eds.). ACM, New York, NY, USA, 14–22. DOI=<http://dx.doi.org/10.1145/2494444.2494447>
- [14] Goli-Malekabadi Z, Sargolzaei-Javan M, Akbari M K. An effective model for store and retrieve big health data in cloud computing. J. Computer Methods and Programs in Biomedicine, 2016, 132: 75–82. <https://doi.org/10.1016/j.cmpb.2016.04.016>
- [15] Dan Arias, Adding salt to hashing: A better way to store passwords, May 2018. <https://auth0.com/blog/adding-salt-to-hashing-a-better-way-to-store-passwords/>
- [16] Dan Arias, Hashing in Action: Understanding Bcrypt, May 2018. <https://auth0.com/blog/hashing-in-action-understanding-bcrypt/>

APPENDIX

Node Dependencies

Node version : 10.15.3

Package	Version
Express	^4.16.4
Express-jwt	^5.3.1
Jsonwebtoken	^8.5.1
Mongoose	^5.5.4
Cors	^2.8.5
Bcryptjs	^2.4.3
Body-parser	^1.19.0
Angular-google-recaptcha	^1.0.3

Table 5: Node dependencies

Angular Dependencies

Angular CLI: 7.3.8

Angular: 7.2.14

Package	Version
@angular/animations	^7.2.14
@angular/cdk	^7.3.7

@angular/common	~7.2.0
@angular/compiler	~7.2.0
@angular/core	~7.2.0
@angular/forms	~7.2.0
@angular/http	^7.2.14
@angular/router	~7.2.0
ng-recaptcha	^4.2.1
rxjs	^6.5.1
rxjs-compat	^6.5.1

Table 6: Angular dependencies

Installations

- npm install express --save
- npm install -g @angular/cli --save
- npm install mongoose --save
- npm install body-parser --save
- npm install -g nodemon --save
- npm install rxjs-compat --save
- npm install @angular-devkit/build-angular --save
- npm install rxjs@6 rxjs-compat@6 --save
- npm install cors --save
- npm install @angular/cdk --save
- npm install @angular/material --save
- npm install express-jwt --save
- npm install jsonwebtoken --save
- npm install bcryptjs --save
- npm install angular-google-recaptcha --save

Github URL

<https://github.com/somyars/BloodBank>