Healthcare Record Keeping Database Systems

Nathan Lucero

California State University-Stanislaus, Turlock, CA

nlucero@csustan.edu

*Abstract*

Healthcare organizations are a premiere example of handling large collections of data. Typically, any hospital must keep records of employees (security guards, accountants, receptionist’s, medical staff (doctors and nurses), patients, and many more who play a role within the organization. These large collections of data that healthcare providers may ask range from a variety of possibilities. They may ask you to enter in your first and last name, date of birth, some contact information, previous medical information, and much more. Authorized healthcare personnel must be able to add, modify, and delete records of individuals as quickly and efficiently as possible for the sole purpose of data management. Therefore, for this paper I propose two effective applications for data management for a simulated healthcare system I call Solaris. One application will be implemented by using a Representational State Transfer Application Programming Interface, otherwise known as a REST API. The other application will take form of a CRUD Web Based Application. Both applications will reference the NoSQL database MongoDB which is the server which will store all the records of the inputted individuals. They also be built using JavaScript as the preferred language all while using Node.js. Node.js contains all the necessary packages and components such as Express, and Mongoose which are critical to connecting to the MongoDB database. The Graphical User Interface chosen for implementing the code was Microsoft’s 2019 Visual Studio. Lastly, both projects will be shown on a free-to-deploy Heroku server. For the rest of this paper, I will state each solutions background information, methods of implementation, their outputted results, as well as provide a recommendation as to which program serves to be a better choice for any healthcare system.

# Motivation

A clear effective method of referencing logged individuals stems from the development of a program within a computer to help automate the process of inserting big amounts of data and also quickly reference any previously logged data. However, there remain organizations that still manually log individuals by using pen and paper. This manual insertion of data can be quite expensive in time and physical space allocation. The wasted time trying to find specific patient records can lead to logistical concern when Doctors or Nurses are trying to quickly serve many clients within a single day. Unless manually organized, this traditional paper method can also transpire into major concern where certain healthcare organizations are forced to allocate these logged paper records into designated storage rooms.

The purpose for developing Solaris was not only to advocate for healthcare providers to shift away from using traditional paper but to also develop a quick and simple application to help expedite the process of logging in enormous amounts of sensitive record documentation. Generating a simple REST API and a flexible CRUD Web Application can help the medical staff cut down on time spent trying to find and modify client documentation. This reduction in time and storage can help any organization attend to more patients more quickly. Spending costs can be also reduced as designated storage rooms can be used for another patient, more equipment, more prescription medication, maybe even add another office for another doctor.

Background

## MongoDB

MongoDB is a NoSQL document database popular for its convenient and simple-to-use indexing of records. NoSQL databases are purposely built to have flexible schemas for building modern applications. NoSQL databases are widely recognized for their ease of development, functionality, and performance at scale. These types of databases are optimized specifically for applications that require large data volume, low latency, and flexible data models, which are achieved by relaxing some of the data consistency restrictions of other databases. Data is represented often as an object or JSON-like document because it is an efficient and intuitive data model for developers. [1]

In this document model, data is stored as a JSON format within documents also called collections. Documents let you structure data in a way that is efficient for computers to process and natural for humans to read. This is incredibly powerful for developers because they do not have to make their design accommodate the needs of the database. This also means they can adapt by adding data when they need to without worrying that this simple change is going to break everything. MongoDB also lets you move data to where you need it do you can keep data near users around the globe for fast and easy access. [2]

## Node.js, Express, and Mongoose

Node.js is a runtime environment for server-side and network applications. Node.js uses JavaScript and it is available for many different platforms, such as Linux, Microsoft Windows and Apple OS X. Node.js applications are built using many library modules and there is a very rich ecosystem of libraries available, some of which I used to build both applications. [3]

Express is a minimal and flexible Node.js web application package that provides a variety set of features to develop web and mobile applications. It facilitates the rapid development of Node based Web applications. The Express framework allows: to set up middleware’s to respond to HTTP requests, defines a routing table which is used to perform different actions based on HTTP a Method and URL, it allows to dynamically render HTML Pages based on passing arguments to templates. [4]

Mongoose is the Node.js dependency that defines schemas for created collections. Mongoose also has utilities for simplifying Node’s callback patterns that make it easier to work with than the standard driver alone. In general, Mongoose makes it even easier to use MongoDB with Node.js. Mongoose will be primarily used to connect the application to the MongoDB server. [5]

## REST API

A Representational State Transfer Application Programming Interface uses a client-server model, which is where a developed server is an HTTP server and the client sends HTTP verbs (GET, POST, PUT, DELETE), along with a URL and variable parameters that are URL-encoded. The URL describes the object to act upon and the server replies with a result code and valid JavaScript Object Notation (JSON). Since the server replies with JSON, components within JavaScript and MongoDB interacts well with JSON and the built application. [6]

## CRUD

The CRUD acronym is used to describe specific database operations. CRUD stands for CREATE, READ, UPDATE, and DELETE. These database operations map very nicely to the HTTP verbs, as follows: POST: A client wants to insert or create an object. GET: A client wants to read an object. PATCH: A client wants to update an object. DELETE: A client wants to delete an object. [7]

## Heroku

Heroku is the cloud platform that will allow this project to build, deliver, monitor and scale applications (apps). Here Heroku will access this projects GitHub repositories and localhost URL and deploy both programs to a server that can be accessed by anyone.

# Related Work

## Experian Health

Experian Health which is an established organization located in Franklin Tennessee whose responsibility is to manage their patient’s identity data. By logging all their patients records they use quantifiable predications to select which prescription or treatments that should be used for particular clients. Experian Health, has been the healthcare industry leader for integrating, automating, and developing the front and back end revenue cycle management process. Experian Health leverages data to improve and modernize healthcare services, to simply assist with patient care. To enable this innovation, the company relies on the NoSQL MongoDB database for its giant ideas and organization of huge collections of patient records. MongoDB was chosen because of its flexible data model, scalability, strong performance, and overall inexpensive cost. Experian Health built a Universal Identity Manager platform with MongoDB to fill a critical gap in how medical groups identify patients throughout their care journey. This patient identifier platform brings together data from separate database systems, and formats the data to create a single view of a patient and improve the safety, speed, quality, and cost of care for patients and caregivers alike. [8]

## Apervita

Apervita is a specialist cloud-based health infrastructure company located in Chicago, Illinois in which exists precisely to automate safe communication of records, contracts, and other privacy sensitive documents between stakeholders. Apervita has used the NoSQL database MongoDB for over seven years. The company integrated the document-based database primarily because of its simple to use facets that make this particular document model good fit for a healthcare industry’s data. Apervita in 2018, made the decision to deploy and migrate its entire platform from MongoDB Atlas on to Amazon’s AWS web server. This company’s mantra is as follows, A writer by the name of Ben Wolfson for MongoDB’s blog posts explains Apervita’s mantra saying, “Medicine is, at heart, a data problem. Diagnosis and treatments contain private sensitive documentation of a patient’s life. These must be protected from misuse. They must also be immediately accessible to the caregivers and decision-makers.” [9]

# Method 1

## REST API: Solaris

For the implementation process of my first healthcare system application, I had prototype different kinds of entry fields that I wanted to be filled in from different individuals. This environment contained a total of 50 different individuals who play the roles of doctors, nurses, receptionists, accountants, security guards, and patients. Each logged record will contain contents such as the individuals: first name, last name, their assigned role, a valid email address, a phone number, their date of birth, their gender (male, female, or non-binary), anyone’s status either alive or deceased ,a yearly based salary, specialization (only for doctors), a patients last appointment (only for patients), a patients primary doctor, a patients diagnosis, a patients treatment, and finally any prescription for any patients.

Most logged entries were assigned randomly using online name generators, phone number generators, and calendar date/time generators. Doctor and patient roles were methodically templated out with research. This research consisted of finding and assigning doctors their specialized field of study. Here I based their income as medium yearly salary from their specialization. Also, patients were assigned the appropriate doctor to treat, diagnose, and prescribe any medication to each client.

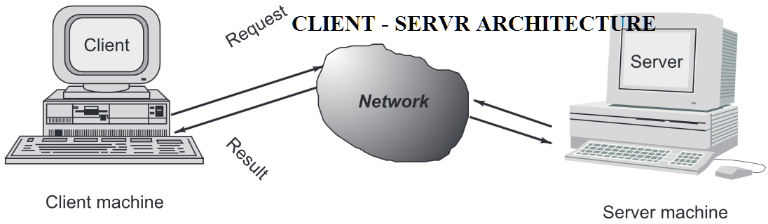
For the first application there was an immediate need for MongoDB’s Compass. Compass analyzes the documents and displays rich structures within defined collections through an intuitive graphical user interface. It allows users to quickly visualize and explore a schema structure to improve understandability within the data’s frequency, type, and range. Compass allowed inserting, deleting, and modifying each individual’s document convenient and also speeds up the development process.

Solaris is the name of the collection holding all the records of the individuals within MongoDB’s database. In order to store all the logged records into a cloud server there was a need for MongoDB’s Atlas. Before connecting Compass to Atlas there was a sign—up process to Atlas’s server. Using the preferred method of Google to login I was then talked to assign roles for people to work on Solaris’s database remotely from different devices. Once a role was generated and security permissions were established a link was given from Atlas. This link is copied to Compass and immediately both Atlas and Compass were connected.

## REST API: Calibrating the application

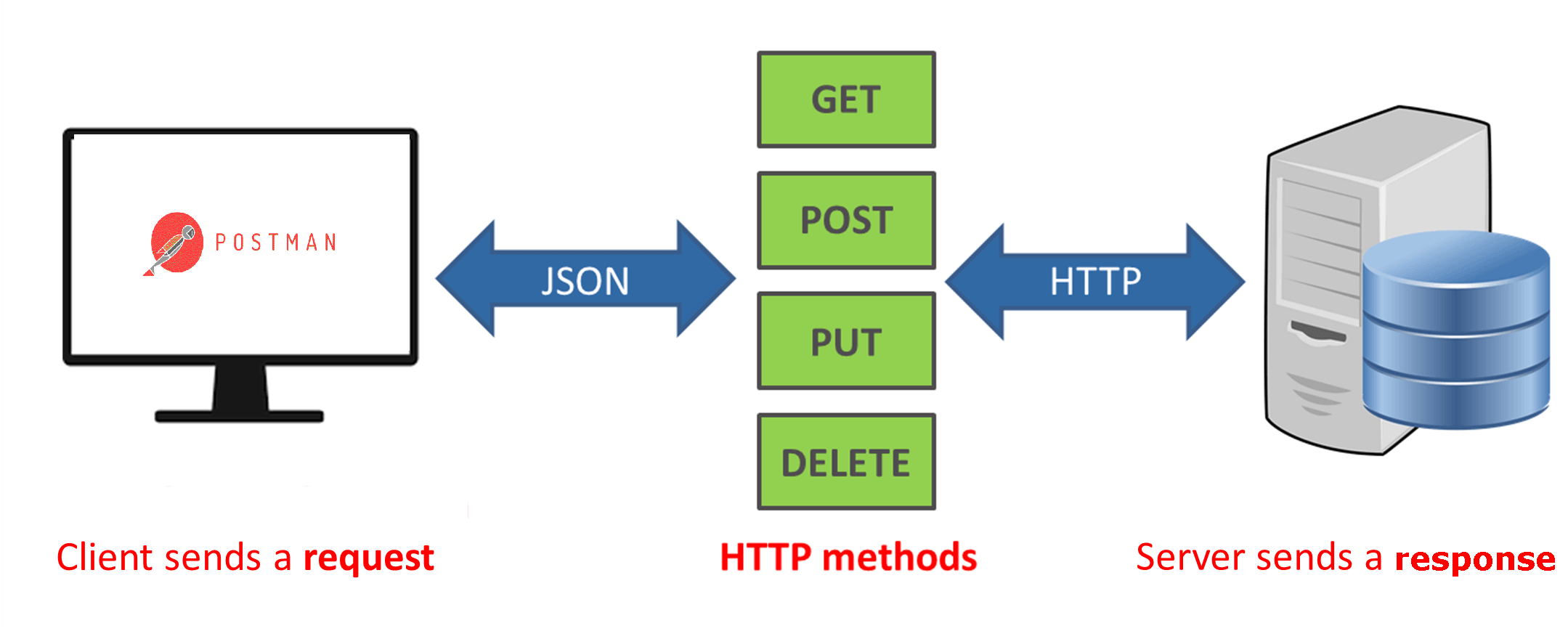
Now that I have a database logged with 50 different individuals and their respected fields entered it was time to develop a REST API to communicate with the MongoDB server. This REST API is responsible for creating, modifying, and reading the stored documents from the MongoDB server.

This REST API will take form of a digital client-server architecture as seen in *Figure 1*. Here we are dealing with of course both clients and servers. A client is a computer (Host). A server is a remote computer which provides information (data) or access to particular services. Essentially, the client will be requesting something and the server will be serving clients as long as the data is present within the database.



*Figure 1: Client-Server Architecture*

Before adding any code there was a need for a third-party application called Postman. Postman will be responsible for handling GET, POST, PATCH, and DELETE methods for a client and server architecture shown in *Figure 2*. The GET method will be responsible for retrieving logged JSON formatted documents from the MongoDB server, POST will allow the insertion of JSON formatted documents to the MongoDB server, PATCH works by referencing any records’ uniquely generated Object ID with a JSON format within Postman and thus modifying any logged data if necessary, and lastly DELETE works by referencing any persons uniquely generated Object ID and is able to remove them completely from the MongoDB server.

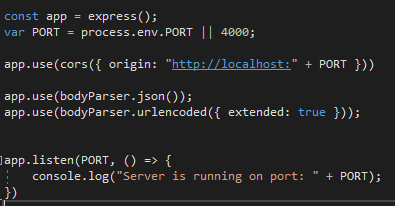


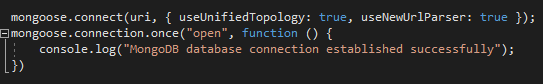
*Figure 2: CRUD Methods*

Now that Postman was successfully installed onto my local device, I was able to then download Microsoft’s 2019 Visual Studio (VS) which allowed me to read and write JavaScript programming code for my Node.js project. To calibrate this Node.js environment within the VS’s CLI I had to install the required dependencies such as: Express, Mongoose, cors, body-parser, and nodemon. Mongoose will be used to link the application to the MongoDB database. Body-parser will allow the application to pass JSON parameters, cors will allow the program to specify the origin of the application if we need to get data from another domain, and nodemon is the tool that will allow me to monitor this application and also automatically restart and start the designed server.

## REST API: Connecting to MongoDB Server

Once the environment has been calibrated, I was then able to code the working application. The first file that was handled was called “server.js”. What “server.js” does is allow the application to successfully connect to my unique MongoDB database on a localhost specified at port 4000. This code can be read from *Figure 3*. The defined “uri” for the line “mongoose.connect (uri, …” was of type const and contained my specific MongoDB server which was, “mongodb+srv://Admin\_NathanLucero:Twister1997@cluster0.gioxl.mongodb.net/Solaris?retryWrites=true&w=majority”. NathanLucero is Atlas’s defined user, Twister1997 is the set password, and Solaris is the collection containing all the records of the individuals on the MongoDB server. If the connection was successful the application will display on the terminal its defined console log message.



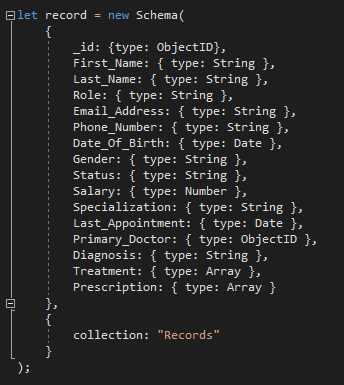


*Figure 3: Connecting to the MongoDB Server*

## REST API: Defining a Schema

In order to create this application’s schema, I created a file called, “model.js”. What this schema file does is define the various types of data within the MongoDB database. This definition of data will allow the REST API to reference this structure when performing a POST method and allow its server request to log the information according to its defined structure.

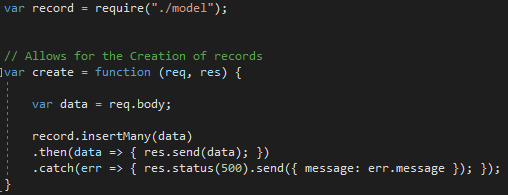
This can be seen on *Figure 4* where the image demonstrates how the structure of the Schema closely resembles the JSON format that MongoDB has. Notice how there is no specified individual. What this file does is act as a placeholder for how the logged data should be implemented. For example: First\_Name should be typed using acceptable characters on a keyboard. Whereas Date must contain a date formatted as YYYY-MM-DD and salary must contain only an integer number value. Type ObjectID is a special reference that explicitly calls for the assigned uniquely generated ObjectID within the MongoDB server. Entries of type array are allowed to consist of multiple entries separate by a comma. The collection is where the data will be stored. I have referenced all the data to be stored within the Records collection.



*Figure 4: REST API Defining a Schema*

## REST API: The Create Method and POST ROUTE

For this part we will be working once again with the “server.js” file. Here I created a variable called create which refers to a function handling request and response transmission. The declared variable data references the “body-parser” since the requested data will be in the body section of the displayed HTML page. The line, “var record = require("./model");” declares a variable record which references the file “model.js” schema within the application. The code, “record.insertMany(data)” supports the insertion of many records within one POST method. Below this is an error message for the user just in case there was a problem inserting a record. *Figure 5* displays the necessary code for creating a record and displays the localhost POST route where the records will be stored in a JSON format. It’s important to note that Postman must be used in order to perform a POST call.

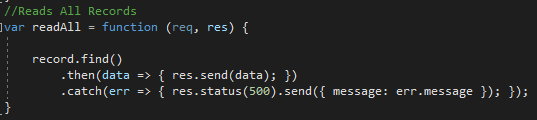




*Figure 5: The Create Method & The POST Route*

## REST API: Read Methods and GET Routes

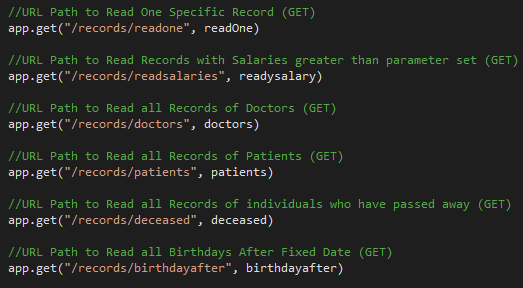
Here I am still remaining in the “server.js” file. Right below the create function I created another function called “readAll”. What this function will do is allow the application to retrieve all the stored that the MongoDB server has stored within its Records collection. More specifically, this will retrieve both any previously stored and newly stored documents by using the code “records.find()”. This line of code is MongoDB’s method of finding any specified filters that are set in between the parenthesis. If no filters are set and is left empty then the application will simply retrieve all the stored records within the database onto a specified localhost path defined by GET. The read method and GET path can be illustrated by *Figure 6*.





*Figure 6: A Method to Read all Records & The GET Route*

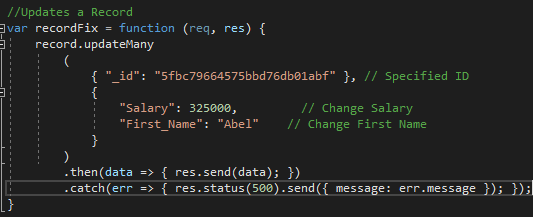
Now to be more creative I decided to add some more filters and developed more GET routes to retrieve my specified filters. The routes and variables I created all follow the same pattern as the “readAll” variable. The only difference is what is inside the “record.find()” parenthesis and the different variable names . Here I added: 1. record.find({ "\_id": "5fbc757e4575bbd76db01abc"}) to retrieve one specified record by their unique ObjectID, 2. record.find({ "Salary": {$gt: 300000}}) to display all the records whose salary value is greater than 300000, 3. record.find({ "Role": "Doctor" }) to display all the doctors within the database, 4. record.find({ "Role": "Patient" }) which reads all the patients within the database, 5. record.find({ "Status": "Deceased" }) to find which individuals that have passed away, 5. and lastly record.find({ "Date\_Of\_Birth": {$gt: new Date('1985-01-01')} }) to find any records whose date of birth is later than January 1st, 1985. *Figure 7* displays a list of all the created variables and their respected GET paths which will retrieve the specified records.



*Figure 7: The GET Route*

## REST API: An Update Method and PATCH Route

Just like the create method, the update method is constructed following a very similar format. The update variable declared was called “recordFix”. This time the function will perform a “record.updateMany()” method. I have made this line of code specify any matching ObjectID to work on their record. Once the record has been specified the data can be modified by applying any necessary changes. Here I selected the ObjectID “5fbc757e4575bbd76db01abc” which is my name within the database. Once I was specified, I changed my salary to 325000 and my first name to Abel. Once that is set and done. I then went over to Postman performed a PATCH call which surely enough updated the name and salary. *Figure 8* will illustrate the coding to perform a update method as well as show the route necessary for the MongoDB server to update and display the contents on a localhost.

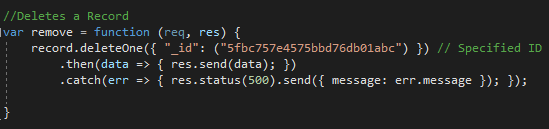




*Figure 8: The Update Method & PATCH Route*

## REST API: A Delete Method and DELETE Route

The final operation for this REST API is the delete method. For this delete method I declared a variable called “remove” and what this function will do is simply delete a record by specifying which ObjectID to be deleted within the MongoDB database. In order to accomplish this a line of code must be written called “record.deleteOne()”. This line of code will allow the program to perform the necessary operation to delete any record specified by their ObjectID. For the example I tested my own ObjectID and performed a DELETE call on Postman which successfully deleted my specified record. *Figure 9* will demonstrate this delete method and route.





*Figure 9: The Delete Method & DELETE Route*

# Method 2

## A CRUD Web Applicaton: SolarisDB

This application was developed to provide assistance in performing CRUD operations to become simpler to understand for an actual user. This application will help bypass the tedious updating of specified records in order to perform POST, PATCH, and DELETE methods. While these methods will still be used, the need for using Postman will be eliminated since most of the code will accommodate for these requests on the backend of development.

This CRUD web application also follows a REST API architecture. The only difference is the redirection of routes and the generation of a layout. The layout will be created by using HTML code to make the website look more attractable and easier to use with the implementation of buttons. For the purpose of this application I will briefly explain some implementation steps for another REST API again since that has already been covered in more detail in Method 1.

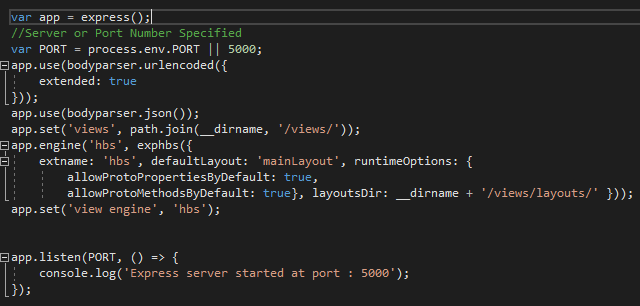
Now it is important to mention, that this collection of data is different from Method 1’s application. I did not create any individuals for this application. What this application does is allow users to simply click, insert, and submit records to a new collection of data which I called “SolarisDB” not to be confused with the collection “Solaris” which has 50 pre-made individuals already recorded.

What I will explain here is how to generate a nicely templated website that is capable of performing inserting of records with the help of a submit button, an option to modify records with the use of a pencil button, an option to delete any specified record by simply clicking on a trash icon, an option to view all the records stored as a organized list, and lastly an option to go back to the back where you are able to insert another record.

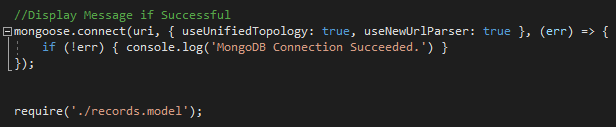
## A CRUD Web Applicaton: Server Setup

Just like the REST API application this new Node.js project will have dependencies contain similar dependencies such as: Express, Mongoose, express-handlebars, handlebars, and a body-parser. Handlebars is the newly installed dependency responsible for allowing bootstrap templates to be developed as “hbs” files. These “hbs” files contain the necessary HTML code for front-end and back-end development.

Just like in Method 1 a file called “server.js” is responsible for loading this application to a localhost server at port 5000. This can be seen in *Figure 9*. In addition, for organization purposes a separate file called “db.js” was integrated to allow this project to connect successfully to a newly declared MongoDB Atlas collection called “SolarisDB”. This connection to the MongoDB server can be seen in *Figure 10*. The declared const “uri” for this project equals the following url: “mongodb+srv://Admin\_NathanLucero:Twister1997@cluster0.gioxl.mongodb.net/SolarisDB?retryWrites=true&w=majority”. If the application successfully connects to both a localhost server as well as connect to the specified MongoDB database then a console log will output a message saying a connection has been successful, otherwise error logs will be created which will help discover where the code can be debugged.



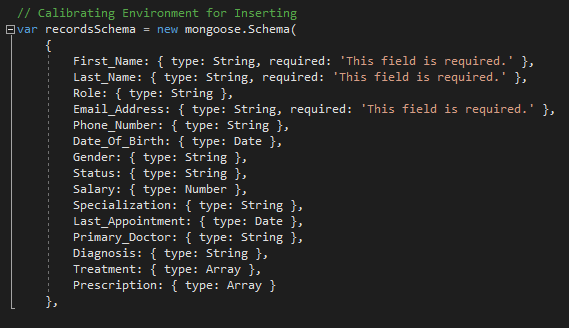
*Figure 9: server.js*



*Figure 10: db.js*

## A CRUD Web Applicaton: Schema calibration

Just like in *Figure 4* of Method 1*,* the defined schema will take place in a file called, “records.model.js”. This schema is essential for allowing the structure of input fields to accurately hold specified data types. A salary must have an integer number value otherwise errors could occur. The primary difference for this “records.model.js” file is that there will be a restriction that will force a user to not leave the entries “First\_Name, Last\_Name, and Email\_Address” empty. This is shown in *Figure 11*.



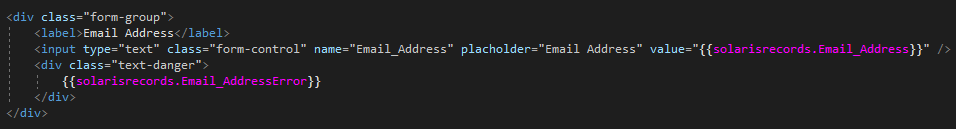
*Figure 11: records.model.js*

## A CRUD Web Applicaton: Using HTML for Website Design

For this part of the report I will go over the “hbs” files called, “mainLayout.hbs”, “addOrEdit.hbs”, and “list.hbs”. The file called “mainLayout.hbs” is responsible for generating a bootstrap template for the websites design. What this means is that by referencing two links (bootstrap links) this program will be able to use premade HTML templates containing all the boarders, colors, sizing, and more. Here also is a hard coded “<div>” tag that generates a background with a hexadecimal color code #b6dcda which resembles the color teal. Here also located at the top of the code creates a <title> tag which simply changes the URL’s tag above the URL. I changed the title to “Solaris Healthcare” to provide some professionalism.

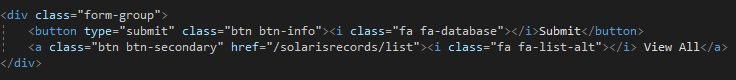
The “addOrEdit.hbs” file contains all the necessary HTML code which will provide the user a bunch of empty fields. These empty fields will serve as placeholder and wait for a user’s input. The URL link “localhost:5000/solarisrecords” will request a POST method that way all the inputted data can be logged to the MongoDB database.

To illustrate this, below is the image of an empty for a Email Address. The text within the <label> tag is designed for the user to see which piece of information the field is asking for, “type=text” allows the user being able to input their own values into a text box, name=”Email\_Address” is the designed schema variable which will take in only that type of value, value={{solarisrecords.Email\_Address}} refers to the assigned value that will be passed as a, POST value. Email Address is a special field which requires to fill out this empty field. Otherwise a danger box will appear if there is no valid input. Not all fields are required though which makes Email Address, Fisrt Name, and Last Name Special in this case. This example is shown in *Figure 12*.



*Figure 12: addOrEdit.js Email Address Example*

Now that our application has a pretty HTML layout it is appropriate to add a submit button which will perform the POST request to the server. If the insertion is successful the user will be redirected to another URL containing records of all the inserted data where they can find their submission. The stored collections of people can be seen at “localhost:5000/solarisrecords/list”. *Figure 13* show the submission button insertion as well as a view all button with HTML code.



*Figure 13: Implementing a Submit Button*

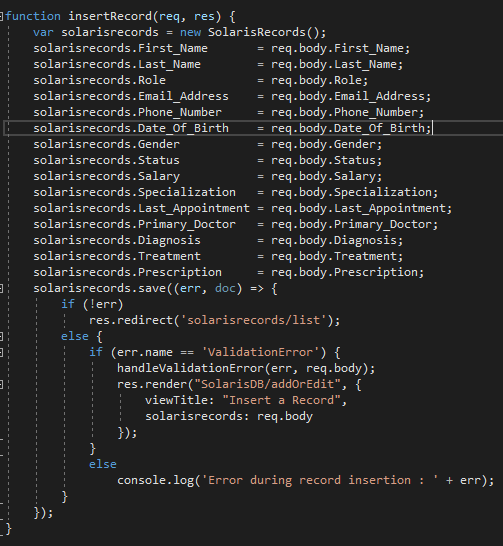
The final URL that needs to be generated can be found within the “list.hbs” file which creates the “localhost:5000/solarisrecords/list” link. What this file does is create the layout necessary for each inputted logged value to be displayed in an organized way. This link also provides a user an interactable pencil icon which allows any clicked entry to be updated and an interactable trash icon which will delete a user upon clicking the icon.

The primary use of this link and file is to retrieve all the inserted documented records stored within the MongoDB server. It is important to note that within the <td> tags, there is a big list of different fields. For example the code <td>{{this.First\_Name}}</td> means that the list will reference “this” value identified as “First\_name” in order to retrieve that record.

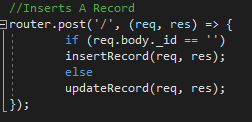
## A CRUD Web Applicaton: CRUD and Routes

In order to have any functionality for our HTML files and links there needs to be some Create, Read, Update, and Delete operations. These operations can be found within the file “recordsController.js”. These generated functions also are connected to their respected routes to perform the necessary POST, GET, PATCH, and DELETE method. For the sole purpose of removing redundancy I will only explain how these applications insert function works.

Just like Method 1’s implementation of a POST Route and Create function. Here also I did the same thing only thing different are the based in values. As seen in *Figure 14* we can see the actual working function for inserting a document. Remember, that “solarisrecords…” is the necessary declaration of each field in order to path the values to “list.hbs” and “addOrEdit.hbs”. If there contains any error inserting a record then the link localhost:5000/solarisrecords/list will display to the user that they have entered an invalid input. If the insertion was successful the values will be passed to the “addOrEdit.hbs” file where a POST method will occur. Once a POST method has occurred and the GET method will work and retrieve the inserted record(s). *Figure 15* is the necessary route needed to insert a record.



*Figure 13: The Create Function*



*Figure 13: The POST Route*

# result

Once I had both applications functioning on their respected local hosts and port numbers, I had to deploy them as a web service application. This deployment was necessary as running these applications as a localhost is very limiting. Deploying both apps to the web will allow both projects to be accessed by anyone who has my generated link.

I deployed both of my applications using a free web-based server application called Heroku. After multiple attempts at Google’s Cloud, Amazon’s AWS, and Azure lead to multiple failed deployments. Heroku on the other hand was super simple and quick to deploy. In order to deploy them I had to create a free Heroku account and from there I simply clicked “build app”. I then cread a Procfile that contained one line of code, “web: node server.js”. What this does is let Heroku start your application on to its own server rather than using a localhost. Once that was successfully done the next task was to download Heroku’s command line interface (CLI). Having Heroku’s CLI will provide the necessary commands which permit the developed Node.js applications to pair with the Heroku server.

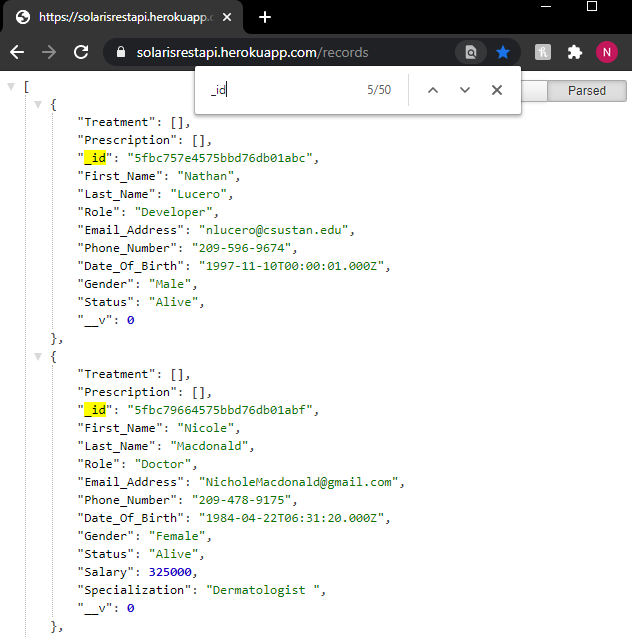
Once the Heroku CLI was successfully integrated into my PC, the next objective was to initialize a Node.js project as a GitHub repository. Finally, after this was accomplished the final task was to upload the repository to a designated branch within Heroku which then generated a working URL to access both applications on a web-based server.

Now that both servers are deployed on the web. Here are the results for both a simple REST API and A CRUD Web Application.

**REST API Results:**

<https://solarisrestapi.herokuapp.com/records>

[All Records]



<https://solarisrestapi.herokuapp.com/records/readsalaries> [Records with Salries Greater than 300000]



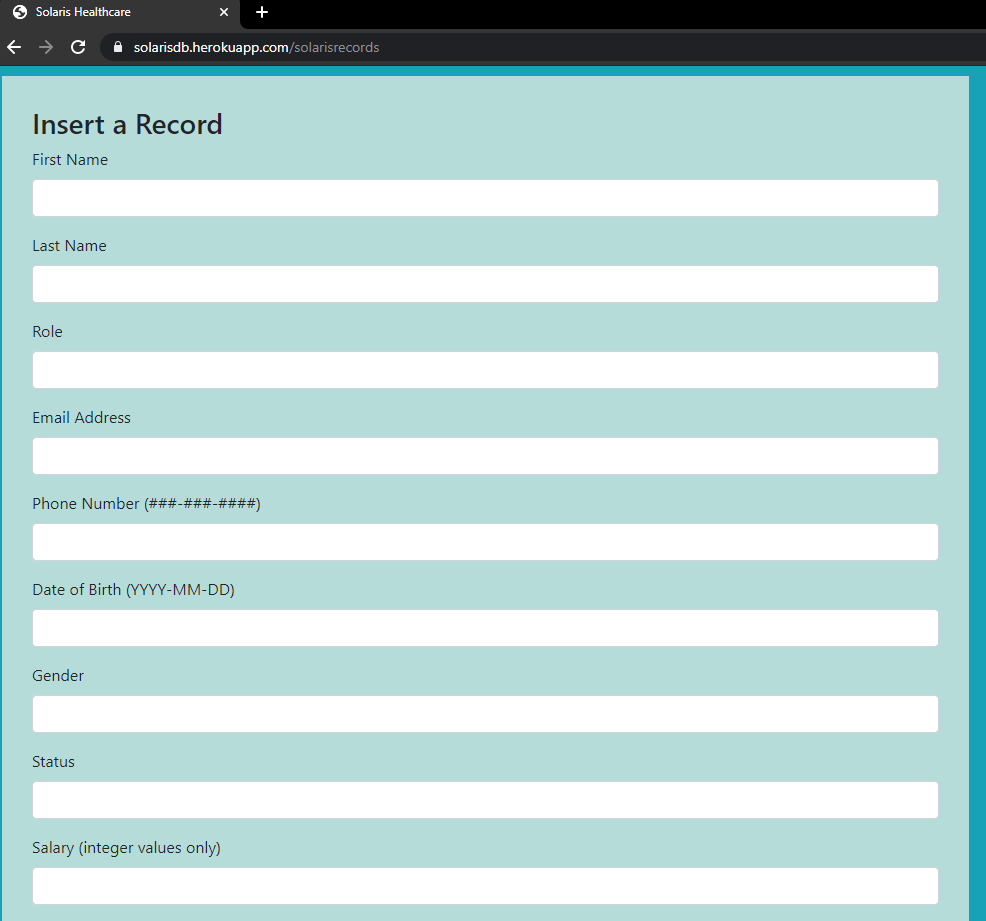
<https://solarisrestapi.herokuapp.com/records/birthdayafter> [Date of Birth Later than 1985-01-01]

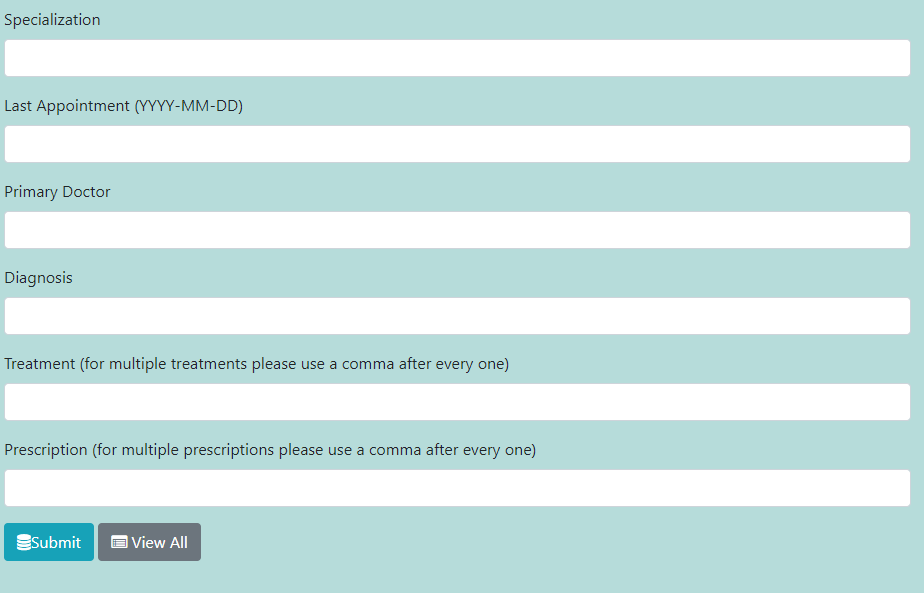


**A CRUD Web Application Results:**

<https://solarisdb.herokuapp.com/solarisrecords>

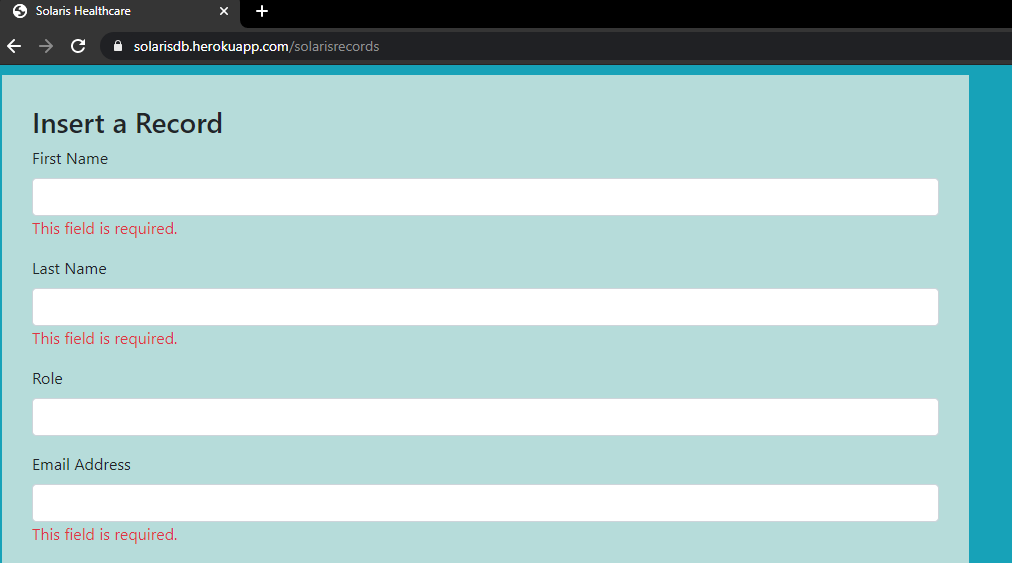
[List of Entries Ready to be Submitted]





<https://solarisdb.herokuapp.com/solarisrecords>

[Warning Message for Required Fields]



<https://solarisdb.herokuapp.com/solarisrecords>

[List of All Records Before Inserting a New Document]



<https://solarisdb.herokuapp.com/solarisrecords>

[Update First Name of “Nathan” to “Abel” & Change email Address]



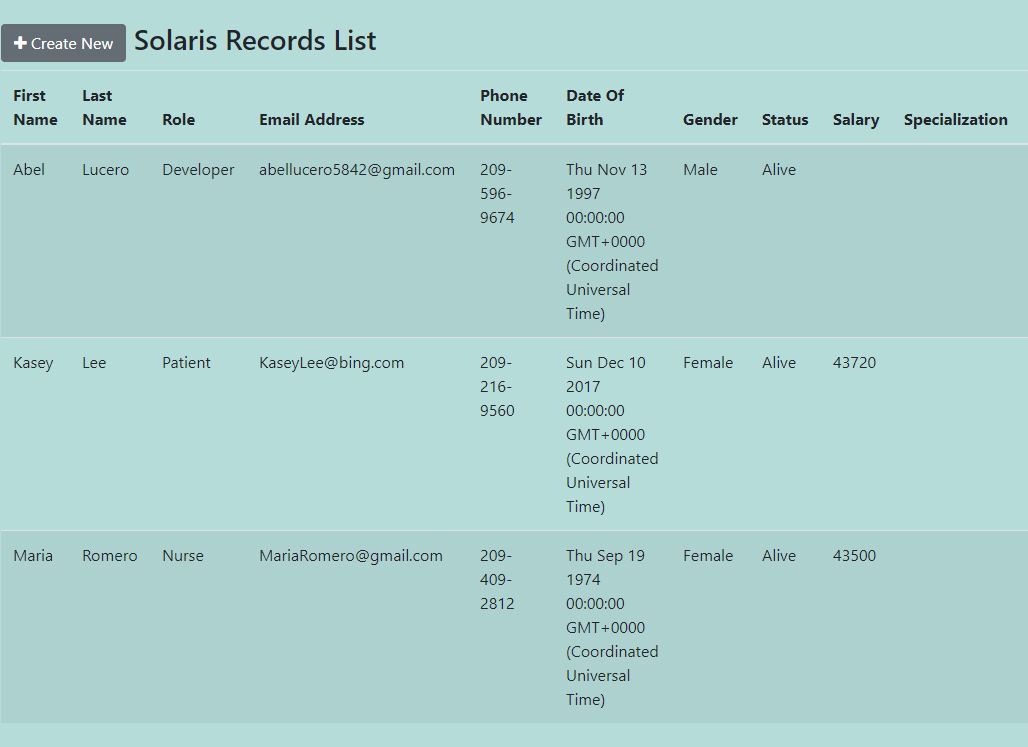
<https://solarisdb.herokuapp.com/solarisrecords>

[Display changed Record]



<https://solarisdb.herokuapp.com/solarisrecords>

[Insert a Nurse Record]



## REST API: Evaluation

This application is by far the easiest to implement. The code necessary to construct this application was short, simple, and effective for the GET method. The “.find()” method was very similar to the structure of MongoDB’s compass which allowed finding and retrieving specific filtered records quick and convenient. Also building the routes to each specified variable containing the unique filter was also an easy feature for this project.

The detriment of this application is the lack of security for the website. As long as you hold the links you are able to see every patient’s history of information. This leakage of private information is horrible especially for any healthcare system. Another downfall of this application is the limitations that work when CREATING, PATCHING, or DELETING. What I mean, is in order to perform an insert of a document a third-party application is needed. Also, the data must be formatted exactly as a JSON file should be which can be time consuming for those who are handling the data. Also, since the code necessary to update and delete is hard coded into the application. This makes the modification and deletion of specific records a hassle as, you have to specify a new Object ID and update the values within the actual code. Once the code is set then the application must be redeployed to the web server and from here a need to execute a PATCH or DELETE is required in order for the MongoDB server to be able to update or remove a record.

## A CRUD Web Application: Evaluation

Developing this application was much more complex as you have to deal with HTML code which I was unfamiliar with. The referencing of where data is going can be a bit tricky since the data is being sent to multiple files at the same time. However, even with that finding the errors was of no concern since there were error logs implemented frequently to see where the problem can be found and debugged.

After getting the application to work I can say that is my preferred implementation for a simulated healthcare system. Unlike the simple REST API this application does not depend on hard coded files and also does not depend on any other third-party application. The user can simply load the page insert a new document, read the contents of all the stored records, update any entry of choice, or delete any entry of choice all with the click of a button.

Now with that being said. Just like the simple REST API this project also suffers from data security. Since anyone is allowed to update and delete logs of records is extremely dangerous for hackers or individuals with malicious curiosity. If this CRUD web-based application had some sort of security protocol forcing only authorized personal to access this database then this application can be more acceptable to use.

# Conclusion

Both programs suffer from security integrity. Data stored within each respected server can be modified or deleted if someone were to do so with malicious curiosity or intent. I would not recommend these systems to be an absolute solution to a real healthcare system as these records are extremely sensitive documentation in which only authorized personnel should be able to view these unique records.

I do recommend either application if someone was interested in either adding more strict security protocols or if someone was just learning how to develop a web-based application using Node.js which was the main facet of what I got out of this project.

I learned not only how to log and retrieve in records but I learned most importantly where the data was being allocated to. It can be overwhelming at times when the server would not function properly and there was no error message explaining where the problem is occurring. I also ran into syntax error due to systematic human error where there were at times missing curly brackets, commas, semicolons, and misspellings. Also, one important error that I kept receiving was the error message, “Handlebars: Access has been denied to resolve the property "First\_Name because it is not an "own property" of its parent.” Now this was fixed by following an online article called, “<http://www.prowebguru.com/2020/08/nodejs-express-handlebars-access-denied-resolve-property-solution/#.X8UznGhKhhE>”. Here the website provided the necessary code to add to my “server.js” which allowed the logs to be retrieved at runtime.

This project also gave me more perspective onto its applicability for the general public. Say this simulated healthcare system were to have the stricter security protocols. What can these logged records do for an organization? These logged entries can provide some inference for some Doctor during their Diagnosis.

Maybe there is a correlation between the type of medication being prescribed and other factors such as an overdose, sickness, or maybe the patient is assigned a bad doctor. This healthcare database can also be broken into subsections where you can have a pharmaceutical database containing each drug’s: side effects, quantity, and cost. A prognosis database where a doctor can add descriptions of how patients are feeling and add any previous symptoms). Lastly, this healthcare system can take form of a logistics database where the primary concern is where medical equipment is being sent, how much it costs, and the quantity of tools being counted.

After developing both applications, I definitely recommend using the CRUD web-based application as the graphical user interface is simple for anyone to navigate through which will help expedite the process of entering and modifying many records within a for a single day.

##### References

1. MENEGASSO, Andre Eduardo. “What Is NoSQL?” Amazon, NOVAS EDICOES ACADEMICAS, 2018, aws.amazon.com/nosql/.
2. MongoDB Inc. “What Is MongoDB?” MongoDB, [www.mongodb.com/what-is-mongodb](http://www.mongodb.com/what-is-mongodb).
3. “Node.js - Introduction.” Tutorialspoint, [www.tutorialspoint.com/nodejs/nodejs\_introduction.htm](http://www.tutorialspoint.com/nodejs/nodejs_introduction.htm).
4. “Node.js - Express Framework.” *Tutorialspoint*, www.tutorialspoint.com/nodejs/nodejs\_express\_framework.htm.
5. Arif, Hussain. “How to Use Mongoose With Node.js.” *Medium*, Better Programming, 20 May 2020, medium.com/better-programming/how-to-use-mongoose-with-node-js-913a8073b29c.
6. “REST.” *REST API Tutorial*, restfulapi.net/.
7. “What Is CRUD?” *Codecademy*, www.codecademy.com/articles/what-is-crud.
8. “Experian Health Bringing Patient Identification into Modern Era with MongoDB.” *MongoDB*, www.mongodb.com/press/experian-health-bringing-patient-identification-into-modern-era-with-mongodb.
9. Wolfson, Ben. “Apervita: With MongoDB, Vital Healthcare Data Stays Accessible AND Secure: MongoDB Blog.” *MongoDB*, MongoDB, 10 June 2020, www.mongodb.com/blog/post/apervita-vital-healthcare-data-stays-accessible-secure.