# Rutgers University
# Software Engineering

## Group 5

Rizwan Chowdhury, Nathaniel Arussy, Dang Khoa Dinh, Smeet Kathiria, Eric Rivera,
Hersh Shrivastava, Suva Shahria, Khalid Akash

# Table of Contents

# Individual Contributions Breakdown

| Name | Percentage | Contributions |
|---|---|---|
| Khalid Aaksh | 12.5% | Team Lead<br>1. Interaction Diagrams<br>2. Data Structures |
| Rizwan Chowdhury | 12.5% | -Report Management<br>-System Architecture and System Design(1-3)<br>- Discussion and Notes |
| Dang Khoa Dinh | 12.5% | Class diagrams and Interface Specification |
| Smeet Kathiria | 12.5% | Global Flow Control<br>Hardware Requirements |
| Eric Rivera | 12.5% | Project Management & Plan of Work |
| Suva Shahria | 12.5% | -Persistent Data Storage<br>-Network Protocol<br>-Algorithms |
| Hersh Shrivastava | 12.5% | Design of Tests |
| Nathaniel Arussy | 12.5% | User Interface Design and Specification |

# Discussions and Notes:

This section is not apart of the report 2 standard; rather it is meant to discuss questions, concerns and provide additional information to the reader about our choices of the technologies that we use in our project.

## Ethereum Costs and Feasibility

The core of our system is using the Ethereum Blockchain to store data. The primary concern with using this existing blockchain infrastructure is the cost of using it. To store data into the blockchain and to run functions that require going through that data require currency called Ether. Ether is valued at $189.20 per Ether. For our system we have to consider the price of Ether and the data we store as well as whether or not actually using the blockchain for our problem is a feasible idea.

For our population descriptor service, we will analyze how much data we utilize and its price in the Ethereum Blockchain. Within the Ethereum blockchain there are costs to uploading/updating data, they are done through transactions in the blockchain which. Transactions, since they change the state of the blockchain, require Ether(money). Reading data from the blockchain is free of charge, unless special functions are employed. For our system, the main costs (for the blockchain) arise from uploading/updating user population descriptors. We will have many users who will regularly update/upload their data. The costs, however, depend on how much ether is actually charged for data put into the blockchain. The price fluctuates but from our testing so far we have seen that uploading 1000 samples of data requires around 3 Ether. Which, if we are to have many customers with many updates/uploads, will become a significant amount of money. Our estimates, with the *current* price of Ethereum is that 1 mb of data will take around $600 to store. This is currently a lower price than that of approximately 1 year ago when Etherium hit its peak price, making the price of 1mb upwards of a few thousand dollars.

Ultimately, the price of using the Ethereum blockchain will vary based on how much data we store within the blockchain. If we store heavy data, magnitudes in gigabytes or higher, then the costs are far too great. In that case, this particular blockchain is not feasible for our system and should not be used. If, however, at the end it ends up the data we store is in the magnitudes of kilo-bytes to megabytes, then costs can be more manageable for the system.

## Why use Ethereum Blockchain?

With the concerns about costs and etc., a question that we considered was why use the Ethereum Blockchain at all. Why not implement our own? We decided to use Ethereum due to the already existing infrastructure it provides and the benefits that come from it. Having an already existing infrastructure provides greater persistence and allows for focus on other features. We will discuss two particular important reasons for our use of the Ethereum blockchain.

The original blockchain paper for bitcoin invented a novel designed called the Blockchain that was a way to persist data and keep consistency among all participating nodes. The specific implementation of this concept was called the Bitcoin which was a way to transfer digital currencies to participants. However, other blockchain projects also came up with their own implementation of the blockchain concept, such as Namecoin which persists domain names over the network. Etherium was novel because it was able to use the blockchain concept to **generalize** the specific implementation of the use of blockchain by allowing users to upload code to customize the behavior of the blockchain. Concepts like Bitcoin and Namecoin can easily be implemented on the ethereum network without having to start a new blockchain from scratch. This leads to the next important point.

The integral focus of the block-chain is data persistence and redundancy. For a blockchain persistence is based on how many nodes exist within the network to hold data/ledgers. If there are not many nodes (computers connected to the network) then the there is a greater impact whenever a node stops participating within the network, valuable redundancy is lost. Furthermore, when there are less nodes there is less incentive for new nodes to join or old nodes to continue participating in the network. These events cause the blockchain to lose places to store data/ledgers thus making data persistence an issue. With Ethereum, we do not have this issue since it is an established blockchain network, currently the **2nd most popular blockchain implementation.** The Ethereum blockchain network has many nodes which guarantees data persistence. Working with a blockchain that does not have the participants that etherium already has makes it so that participants are less incentivized to pursue in mining.

Furthermore, since Ethereum is an established network, there exist many tools and apis that can be used to make our system better. For example, we utilize the web3 library built specifically for the Ethereum blockchain to carry out many functions relating to Logins and carrying out function on data in the blockchain. Also we are planning on utilizing a tool called MetaMask to make logging in simpler and more secure for our users.

Another reason for using the Ethereum blockchain is to save effort on constructing a new blockchain infrastructure and instead focus on features. We get additional time to work on better graphics and other services that we would not be able to offer otherwise.

## Our Plans and Possible Changes

Initially our group heavily advertised the idea of using population descriptors to ultimately obtain data and diagnose illness or prescribe possible treatments. However, at this point we will probably not pursue this idea any further. This is due to the immense amount of domain research and extra functionalities we would have to undertake. At this time it is clear, this feature is beyond our scope. Also, since we are affiliated with any medical institution we feel we should not try to undertake this feature. From this point and on we will focus on fitness data. We will obtain data related to fitness and ultimately give users references to websites or sources that can help with their fitness. Fitness includes parameters such as amount of exercise or weight.

# Interaction Diagram

The following are system sequence diagrams for our most important use cases.

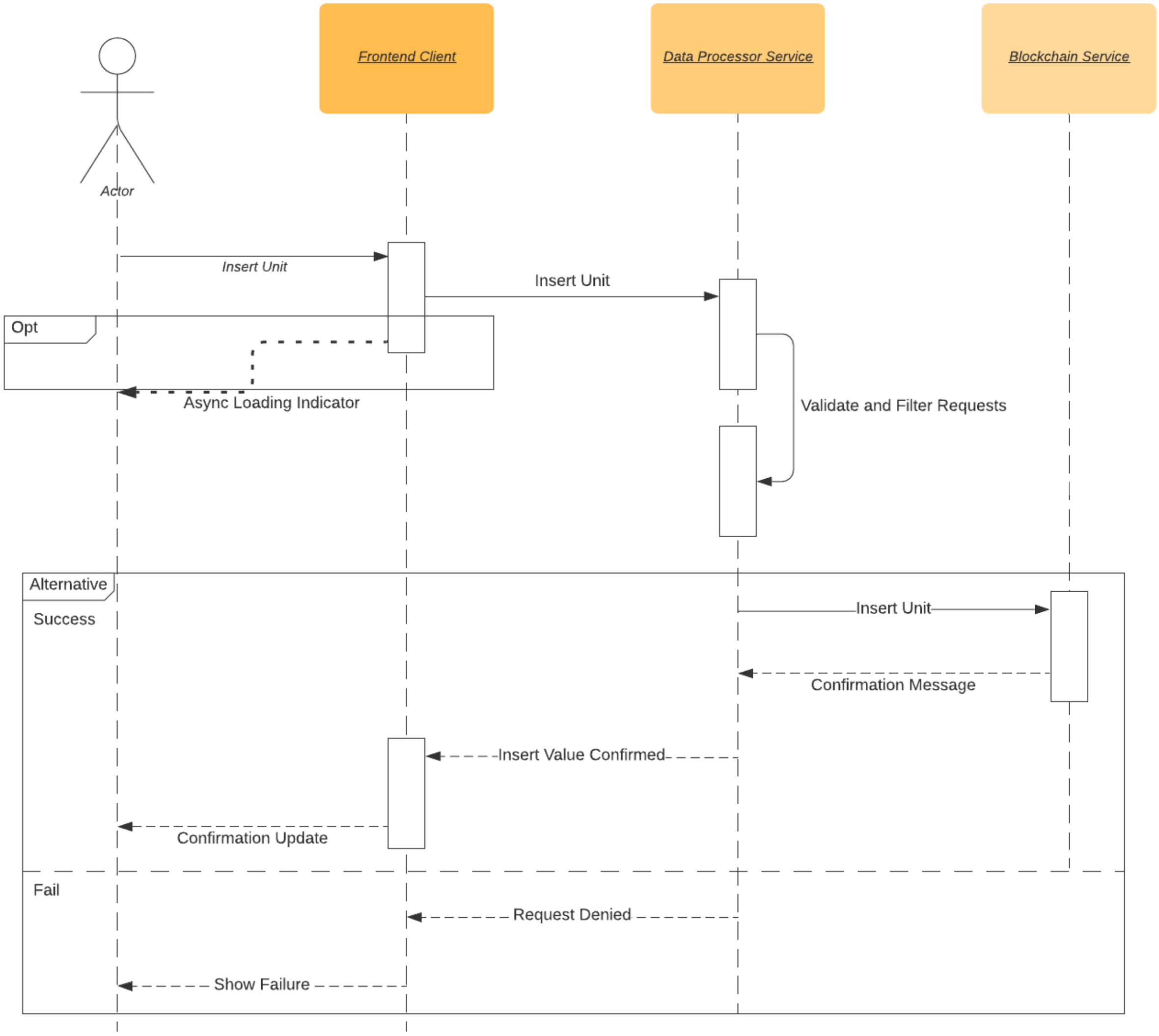


Figure 3.c.1.1 UC-3

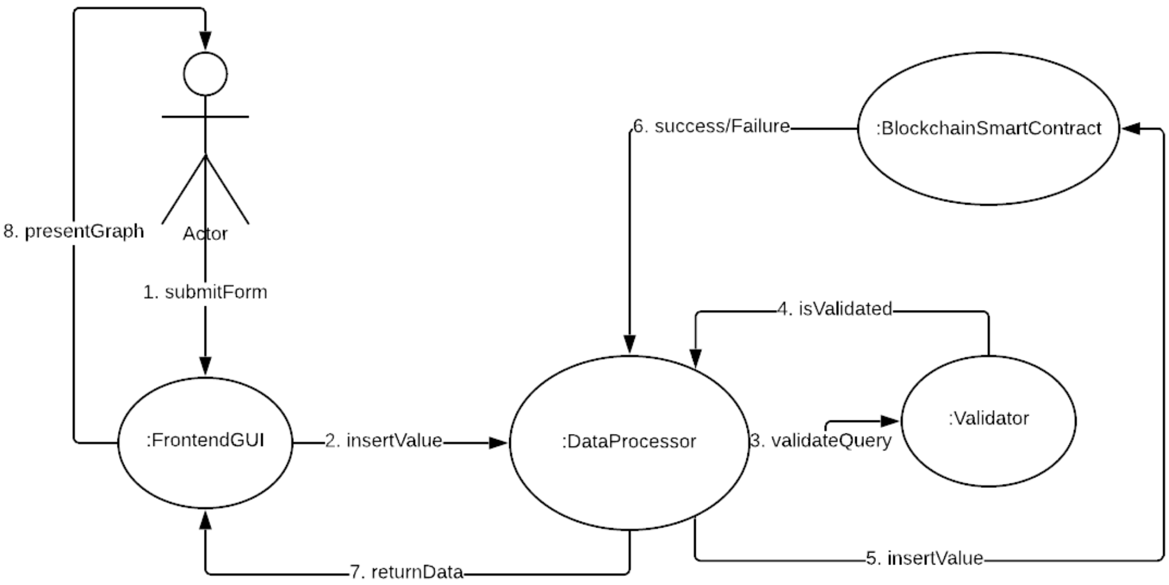


Figure 3.c.1.2 UC-3

This first use case shows the sequence of the insertValue use case where users will need to insert some sort of value to the blockchain network for personal and global use. The three participating systems are the Frontend Client (GUI), the centralized backend server (Data Processor Service), and the Blockchain (Blockchain Service). First the user (Actor) interacts with the Frontend Client to insert some value via a form. A loading indicator is optionally shown depending on implementation. The client sends a request to the backend service which validates/filters any requests that are out of the norm (outliers in the data). Then, if data insertion in the blockchain is successful, the chain will bubble back to the actor, alternatively failure will flow back to the user as well. In terms of design principles, we tried to make the most of the system as stateless as possible; only keeping state in the blockchain. In addition to this, we used a three tiered architecture pattern to separate the concerns of the frontend, the processor, and the storage.

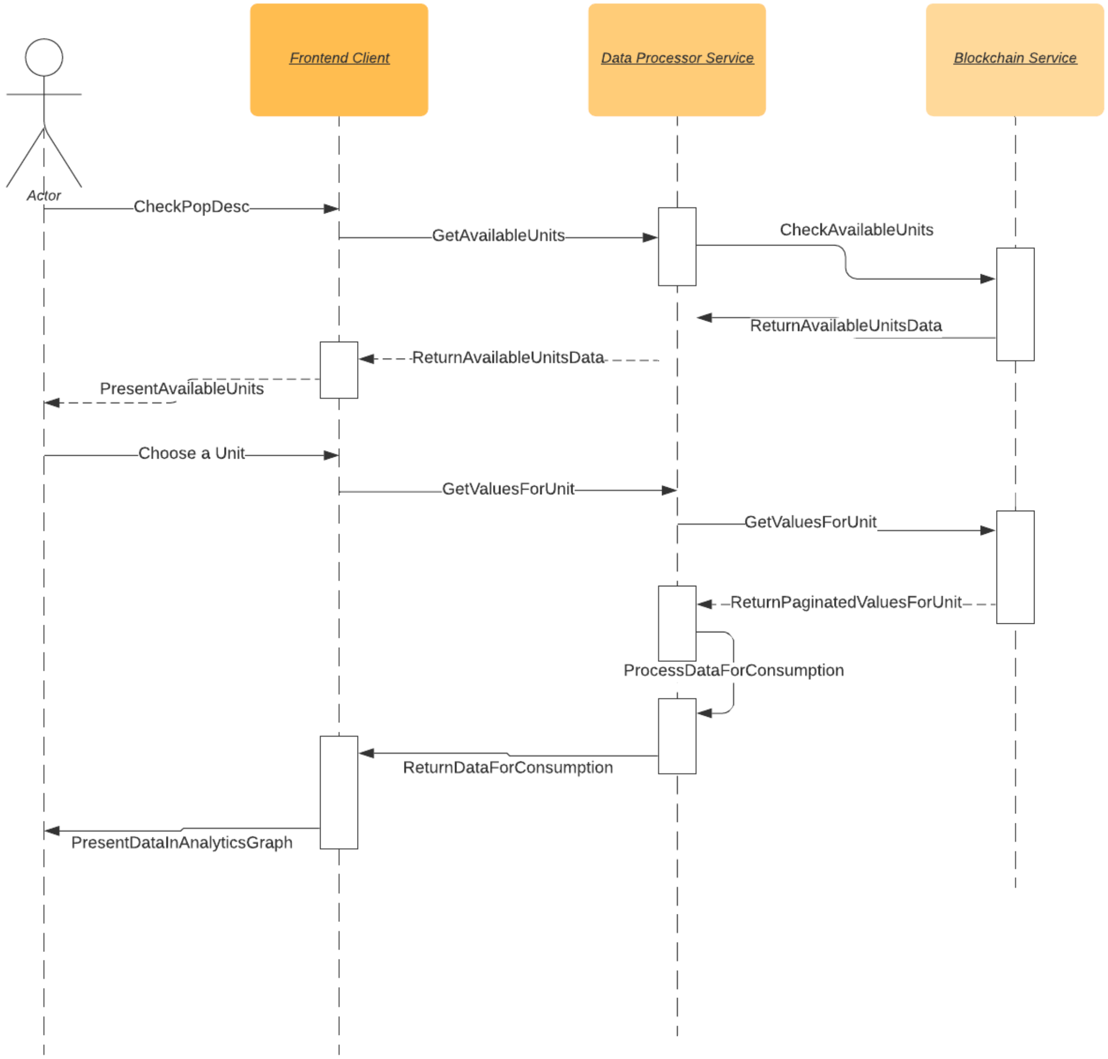


Figure 3.c.2.1 - UC-5

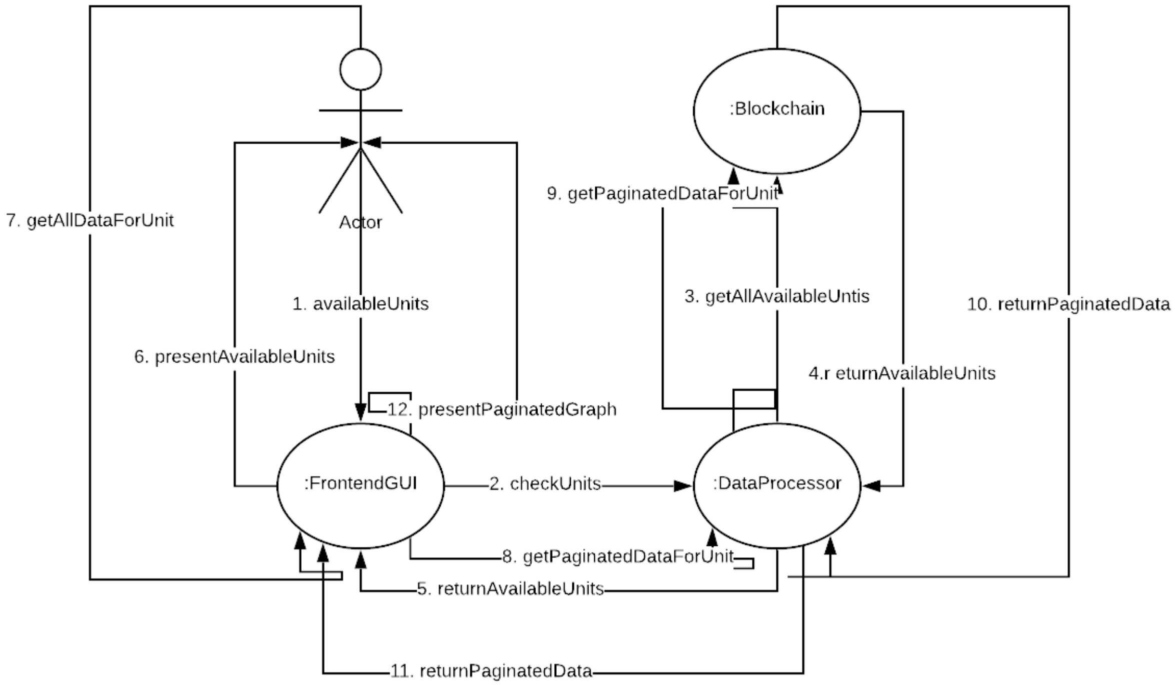


Figure 3.c.2.2 - UC-5

This use case shows how data of population descriptors are retrieved from the blockchain. The three participating systems are the Frontend Client (GUI), the centralized backend server (Data Processor Service), and the Blockchain (Blockchain Service). The user (Actor) first asks what units of descriptors are available to the client, which traverses through the centralized backend, to the blockchain. This returns all available units which the user chooses from. Then the user asks the user to gather data for a specific unit and this traverses through to the blockchain into a paginated data request. Depending on the graph that the page wants to render, different pagination counts may be chosen. Then the backend formats the data in a format that the frontend wants to consume, at which point the data traverses back to the frontend client to render to the user. For brevity, UC-4 also has a very similar sequence but only involves user specific data (in which case a user context will be passed to retrieve the data).

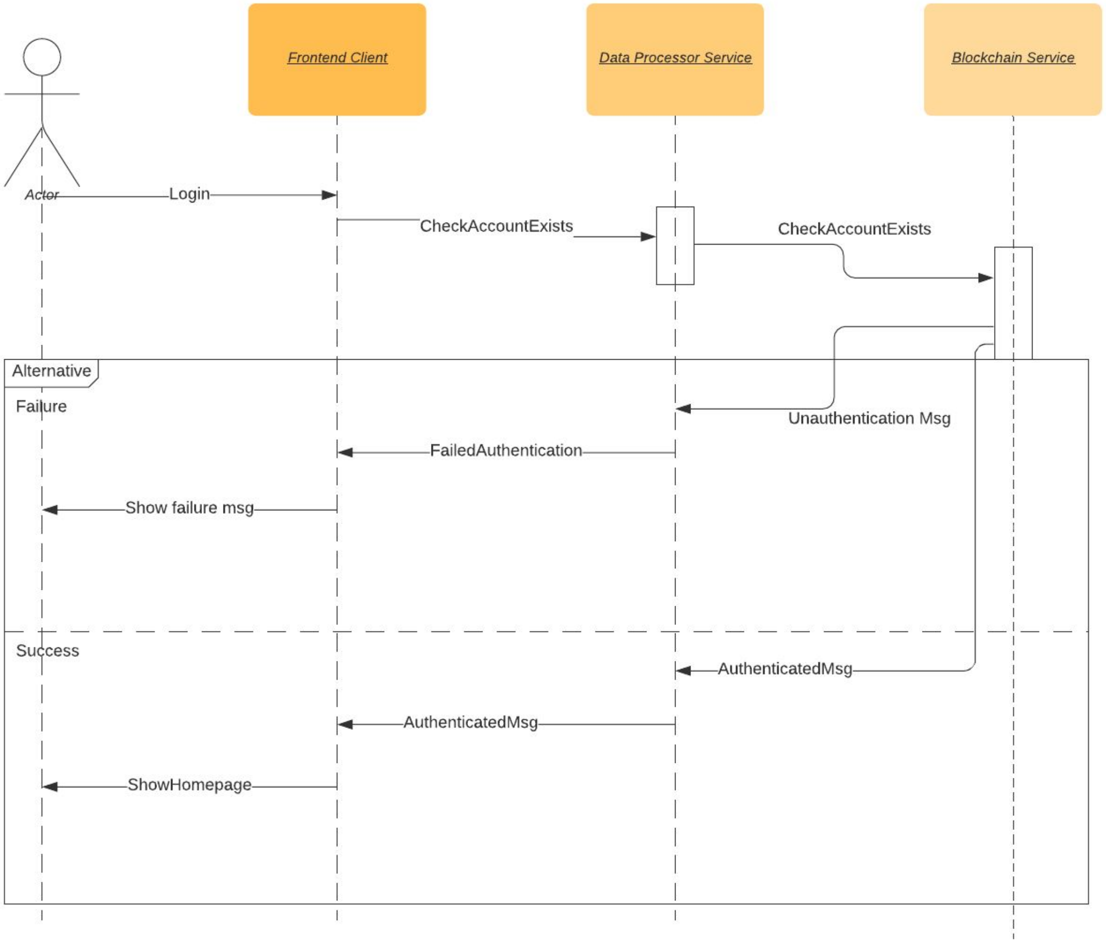


Figure 3.c.4.2 UC-1

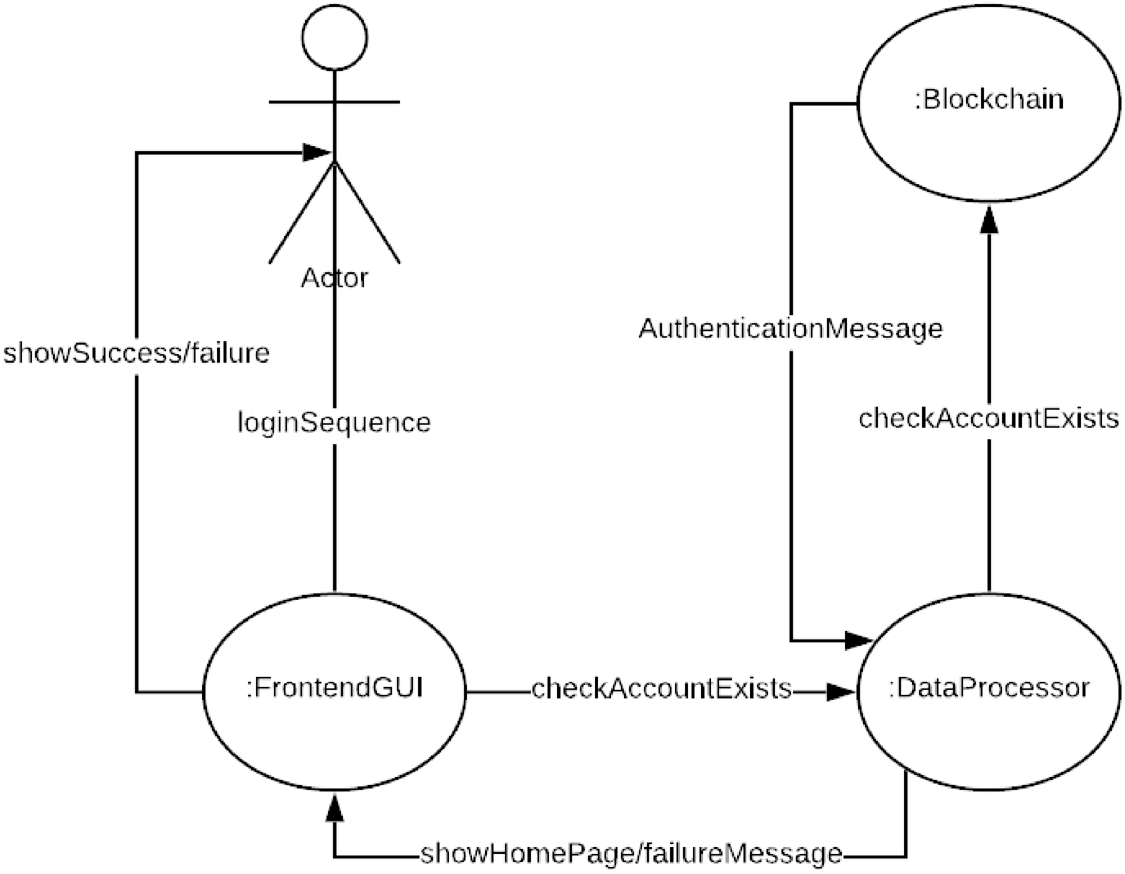
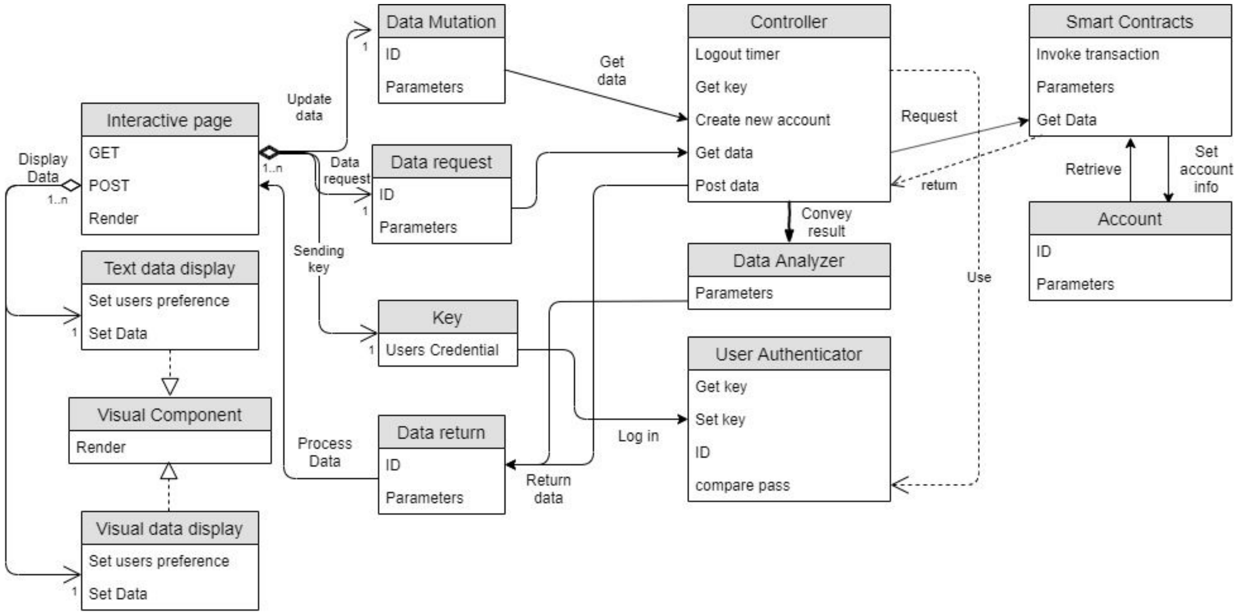


Figure 3.c.4.2 UC-1

This use case pertains to logging into the system and identifying the user for Authentication/Authorization. This is an estimated sequence as the full implementation of the authentication must be actively researched. However, the only place we expect things to change is in the CheckAccountExists method in DataProcessorService and BlockchainService. Failure and success produces sequences that bubbles up to the user.

# Class Diagrams and Interface Specification

## Class diagram



Compare to the domain analysis in report 1. We have rearranged the class into 4 stages: frontend, transport layer (graphql), the backend, Ethereum network. Most of the concepts remain the same as in report 1. After an iteration, we decided to add in some data objects to represent the transport layer in practice.

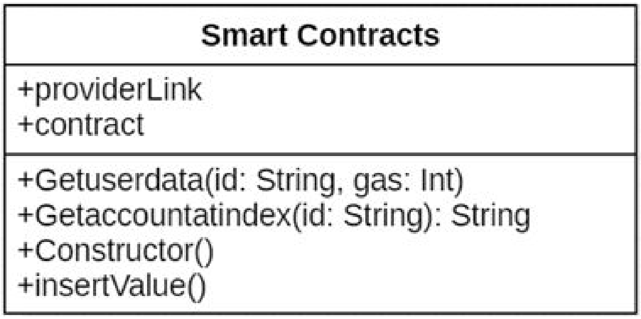
| Class | Operation and Attribute | Description to in the class diagram |
|---|---|---|
| Text Data Display | Set user preference | Customize text display to the users' preference then send it to the controller |
| | Set Data | The server will pass the data into the method |

| | | |
|---|---|---|
| Visual Data Display | Set user preference | Customize the visual display to the users’ preference then send it to the controller |
| | Set Data | The server will pass the data into the method |
| Interactive page | Get | The primary getter and setter for the front end to connect to the backend |
| | Post | |
| User Authenticator | Compare pass | Comparing the password to verify the user’s id for the controller |
| | Set key | Receiving the user credential from the key |
| | Get key | Get the hashed key to compare with the password |
| Data Analyzer( Data processor) | Parameters | Analyze the data to determine the mode, median or other important indicators and sending it back to the server |
| Data Return/ Mutation/ Return | ID | The user’s id |
| | Parameters | The request and return data |
| Smart Contracts | Invoke transaction | A transaction usually occurs when a new user signup or the old user update their data |

| | | |
|---|---|---|
| | Add parameters | Add flexibility to the system as the administration can add new parameters |
| | Get Data | The interface for the blockchain to connect to the blockchain |
| Key | User Credential | User’s identification information, such as user ID and password and passing to the controller |
| Account | ID | The user’s id |
| | Parameters | The user’s health information |
| Controller | Logout timer | After a certain amount of time, the system will log the user out. |
| | Create new account | Set up the new account when request on coming in for the page and invoke a transaction |
| | Listen | Receive request and send the data from the interactive page via the transport layer |
| | Set data | |

## Data types and operation signatures

### 1. Smart Contracts



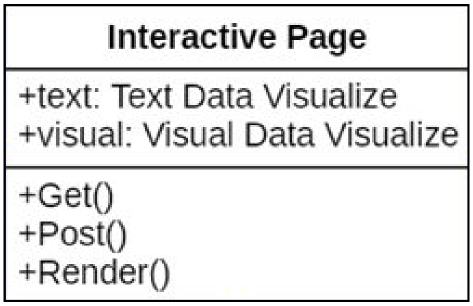
**Smart Contracts**

+providerLink
+contract

+Getuserdata(id: String, gas: Int)
+Getaccountatindex(id: String): String
+Constructor()
+insertValue()

- This is the interface between the blockchain and the serve
- The constructor with take care of invoking transactions or deploying the blockchain and hiding the implementation of the Smart Contracts to the rest of the system
- Insert value is for adding new parameters
- Getuserdata is for get to the user data. Gas is variable for the network and is set depending on the use cases
- Parameters attribute is missing in this case. That is stored in the blockchain. This class is interface not the whole system but for all intended purposes the patent parameters can be seen as an attribute of the system to simplify the analysis.
- ProviderLink is the link to the blockchain
- Contract is actual node where the users' data is stored

### 2. Interactive Pages



**Interactive Page**

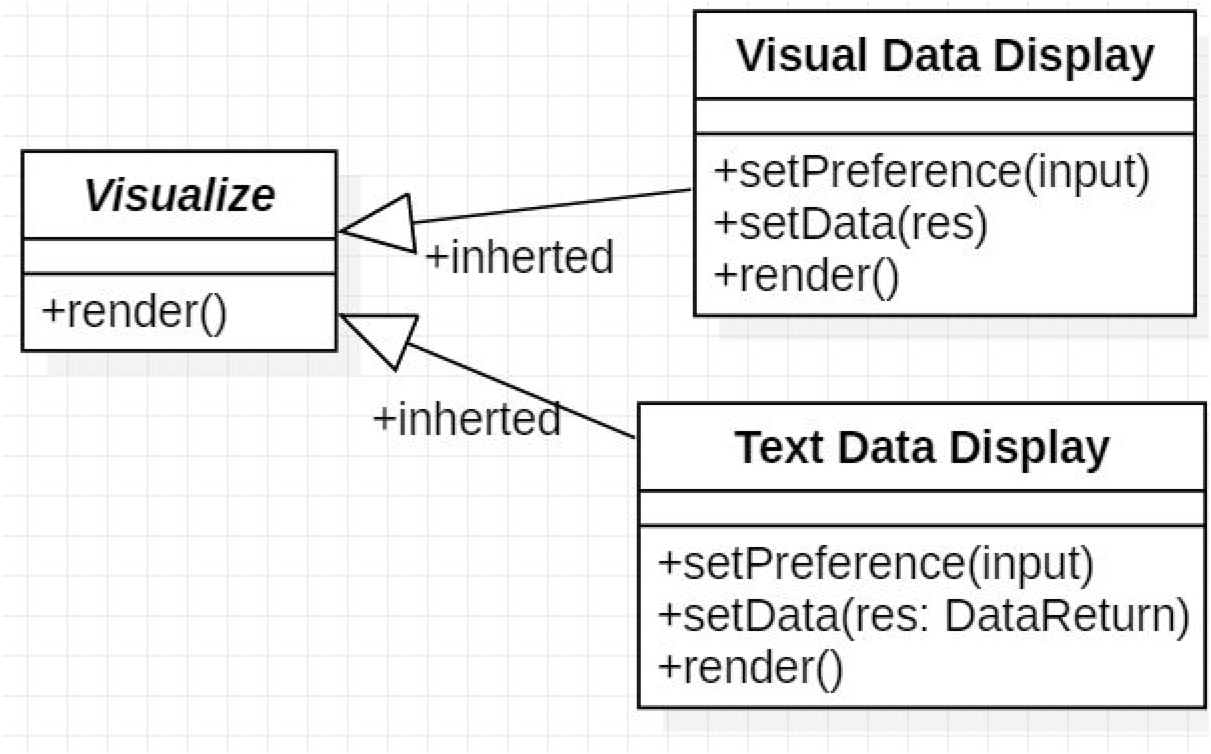
+text: Text Data Visualize
+visual: Visual Data Visualize

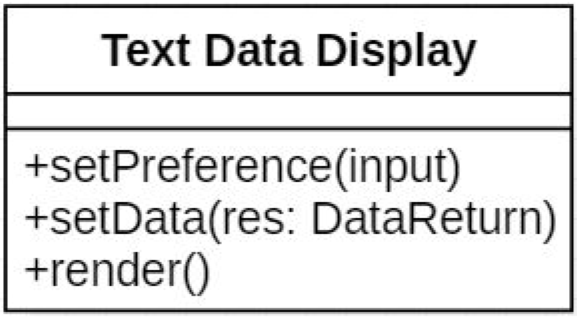
+Get()
+Post()
+Render()

- The render operation will render the element of the visual and text display data to the users
- The get will send an http request to the server request
- The post will receive a response from the server after the request

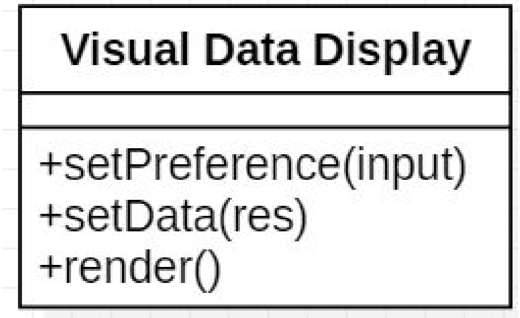
## 3. Text/Visual Data Visualize

They are inherited the Abstract Visual Components





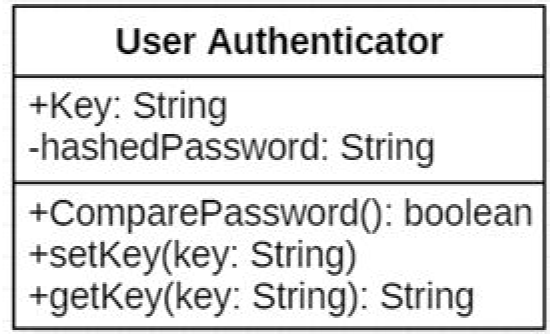
- setPreference(input) allowing the user to set the system to their taste like font, text color, text size
- setData(res) receiving the data from the server and displaying it to the webpage
- render displaying this particular components to the users



- setPreference(input) allowing the user to set the system to their taste like types of graph, color, zoom

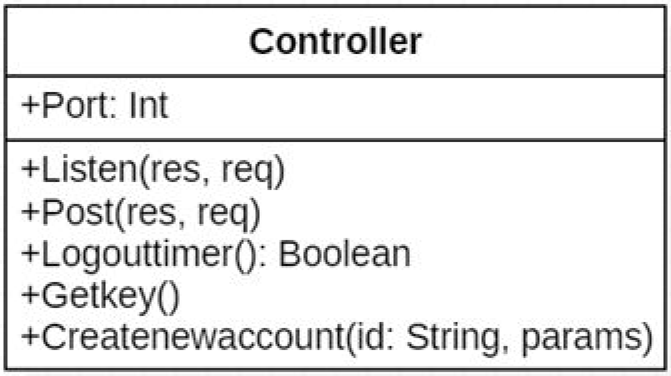
## 4. User Authenticator

This is the login mechanism for the system



- ComparePassword: Boolean verify the user’s id
- setKey: receiving the key
- getKey: get the key
- hashedPassword: String is the encrypted password
- Key:String store the key

## 5. Controller



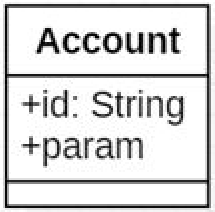
**Controller**

+Port: Int

+Listen(res, req)
+Post(res, req)
+Logouttimer(): Boolean
+Getkey()
+Createnewaccount(id: String, params)

- This is the primary components of the service provided
- Port is the address where requests will be sent to like “https:localhost:” + port
- Listen method where the controller will continuously listen for the requested data passing it to the rest of the system
- Createnewaccount with add new account for the users

## 6. Account



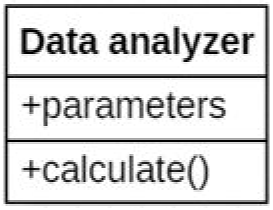
**Account**

+id: String
+param

The object contains all the information associated with the users in the blockchain

- Id: the users individual identification tag
- Param: the parameter of the associated patient

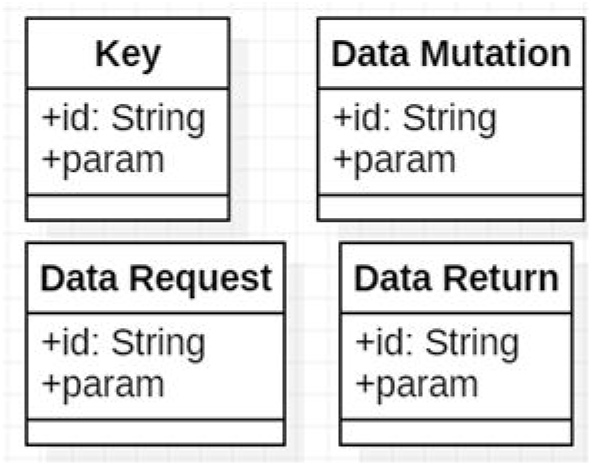
## 7. Data analyzer( Data processor)



This object will receive the input of the data server and output the correspondent result for the data

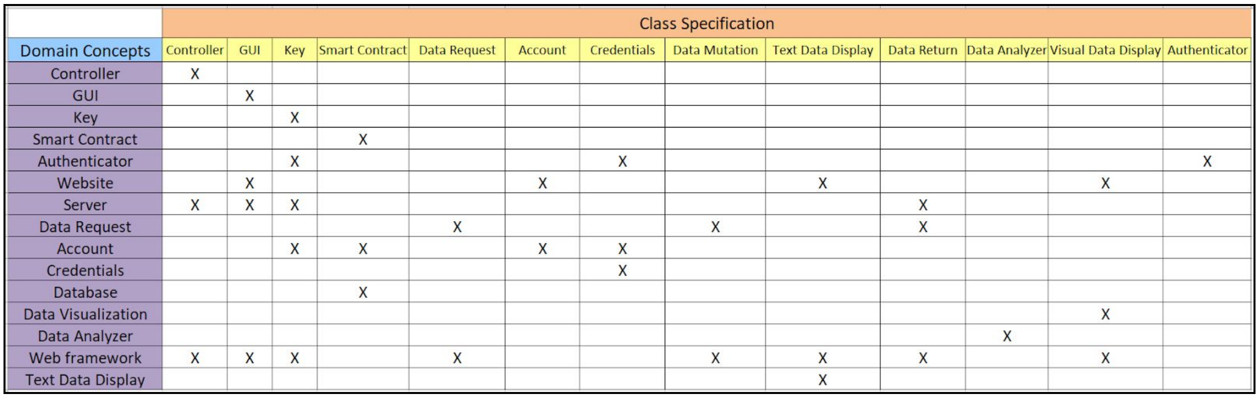
- The calculate the median, average, mode, percentile, percent of risk which is then displayed to the user

## 8. Data Mutation/Data Request/Data Return/Key



This is the link between the client and server. The interactive page will send a request with a Key/Data Mutation object with the field id and other corresponding parameters field. After processing the request, the controller will send back a Data Return object to the frontend for processing.

## Traceability Matrix



| | Class Specification | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Domain Concepts | Controller | GUI | Key | Smart Contract | Data Request | Account | Credentials | Data Mutation | Text Data Display | Data Return | Data Analyzer | Visual Data Display | Authenticator |
| Controller | X | | | | | | | | | | | | |
| GUI | | X | | | | | | | | | | | |
| Key | | | X | | | | | | | | | | |
| Smart Contract | | | | X | | | | | | | | | |
| Authenticator | | | X | | | | X | | | | | | X |
| Website | | X | | | | X | | | X | | | X | |
| Server | X | X | X | | | | | | | X | | | |
| Data Request | | | | | X | | | X | | X | | | |
| Account | | | X | X | | X | X | | | | | | |
| Credentials | | | | | | | X | | | | | | |
| Database | | | | X | | | | | | | | | |
| Data Visualization | | | | | | | | | | | | X | |
| Data Analyzer | | | | | | | | | | | X | | |
| Web framework | X | X | X | | X | | | X | X | X | | X | |
| Text Data Display | | | | | | | | | X | | | | |

# System Architecture and Design

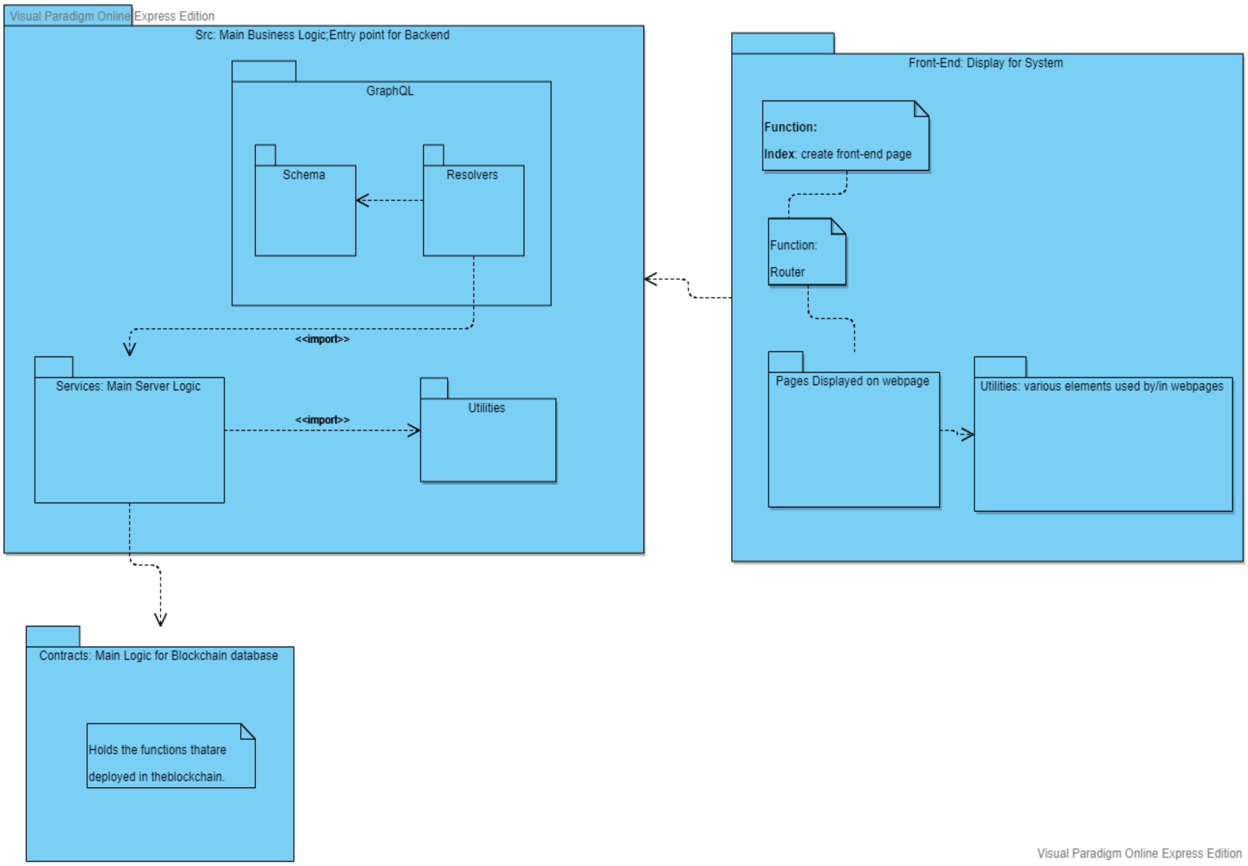
## Architectural Styles

Our system employs different types of architectural styles across different scope. From the scope of the entire system a Three-Tiered architecture is implemented. The client-side user interface acts as the presentation layer where a user can make requests and view the results. User requests are handled by the application layer, a Node JS Express server. The model in this architecture is composed of the various files and functionalities of the GraphQL api as well as the Smart-Contracts for the Ethereum Blockchain. The GraphQL api interacts with the database ( the blockchain in our case) and handles all queries. Furthermore, GraphQL provides the capacity to apply business logic to the data from the database through resolvers. Resolvers are functions that can either simply retrieve data from storage or retrieve data and manipulate them as appropriate. The Smart-Contracts are called through the resolvers in order to obtain the data from the blockchain, they are a tool necessary specifically for the blockchain. Through the user-interface, Express server, and GraphQL api a MVC architecture is obtained.

While from a larger scope, our application is Three-Tiered, inside each of the tiers (particularly the client and server), we employ a Model-View-Controller architecture. The Express server file resides within the same package as GraphQL schemas and functions. This may be viewed as the controller and model being one large backend server. Since the controller and model share a package they can be treated as a single entity in which case this becomes a client-server architecture. However, the controller and model functionalities are kept separate and one does not depend on the other for functionality.

The model as a collection of several modules and files itself has a layered architecture. The model is composed of several functions and schemas relating to the GraphQL api and Smart-Contracts for the blockchain. The highest layer is comprised of the “serveGraphQLRequest” contained within the graphql index file. This function receives the query request and then passes it onto the different resolvers. The GraphQL resolvers are a layer below and they process the query to determine which data is requested. The resolvers then call on the Smart-Contracts through the services layer. The services layer receives function calls from the resolvers then sends back results, sometimes by calling on Smart-Contracts. The Smart-Contracts interact directly with the blockchain and obtain the information requested by the resolver and returns. The resolver can then carry out logic with the data and then pass it back to “serveGraphQLRequest” which will send the results to the controller to be displayed. Each of these layers interact only with the adjacent layer, the Smart-Contracts can only be accessed through the services layer which is accessed only through the resolvers and so on. There is a clear hierarchy which establishes a layered architecture.

## Subsystem Package Diagram


Visual Paradigm Online Express Edition
Src: Main Business Logic,Entry point for Backend
GraphQL
Schema
Resolvers
<<import>>
Services: Main Server Logic
<<import>>
Utilities
Front-End: Display for System
Function:
Index: create front-end page
Function:
Router
Pages Displayed on webpage
Utilities: various elements used by/in webpages
Contracts: Main Logic for Blockchain database
Holds the functions thatare
deployed in theblockchain.
Visual Paradigm Online Express Edition

## Mapping Subsystems to Hardware

Our system breaks down to three subsystems that will run in three different sets of hardware. The client side front end, the web server where the controller and several services for the database reside, and finally the blockchain.

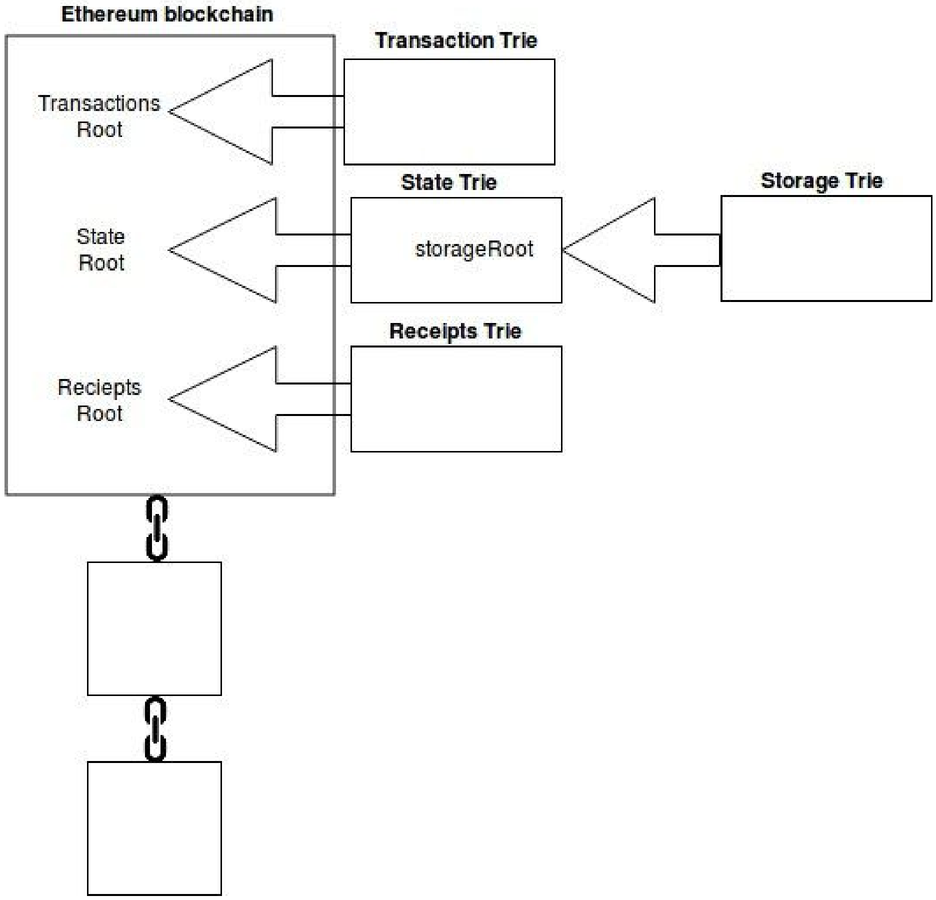
**Client-Side Frontend**: This is a collection of webpages and user-interfaces that the user will interact with. This subsystem will reside in the user’s device through a web browser, and will use whichever device the user is accessing the website with.

**Web-Server:** The web-server will contain the Express Server and some other services that will be used to process the data. The webserver will also act as an intermediate between user interface and the data in the blockchain. This will have to be hosted on servers specifically for web-servers. We do not yet have anything specific in our plans but services such as Amazon Web Services or Github can provide the necessary hardware and services to host the web-server.

**Blockchain**: Essentially the database of the system, will house the data collected from users. Blockchain is a decentralized peer-to-peer system that exists with many devices. Each device in a node contains data. Since blockchain as a concept requires many devices it cannot be mapped to a single piece of hardware, rather the requirement for a blockchain network would be as many computers as possible. More computers mean more devices to store data and more security. Blockchain does not require any special hardware, regular computers are adequate, the requirement is to have several to many devices. For an Ethereum blockchain specifically, devices that are already part of the Ethereum Blockchain network will be necessary.

## Persistent Data Storage

Users in our system will  be interacting with the ethereum blockchain to manage their data. Ethereum uses a tree data structure as illustrated in the figure below.



Underlying the structure:

State trie

There is only one state trie. The state trie contains a key and value pair for every account that exists on the Ethereum network. It constantly gets updated.

Storage trie

Each Ethereum account has a storage trie. All the contract data lives here.

Transaction trie

Each block in the ethereum network has its own Transaction trie . Transaction data is stored here.

Merkle Tree:

Ethereum stores two types of data permanent data and ephemeral(temporary) data. Transactions are considered permanent data and stored in the transaction trie and is never altered. Ethereum account address is considered temporary data and is stored state trie.

Blocks are stored on a multi-level data structure. The hashed address of a block points only to the block header which contain the timestamps, previous block hash, and the root hash of the merkle tree data structure that has all the transactions in the block. Each node of the tree is the hash of its children. A “full node” following the merkle tree protocol takes up about 15+ GB of disk space.

Ethereum also supports a protocol called "simplified payment verification" (SPV). This allows “light nodes to exist” which download the block headers and only download the branches with transactions that are relevant to them. Proof of work is verified on the block headers.

## Network Protocol

We will be using  a backend proxy server that retrieves data from the blockchain. Reading and processing data from the blockchain is expensive, and results in high latency, as a result we will use the proxy server for caching and processing. The server will make JSON-RPC (Remote Procedure Calls) to communicate with the blockchain.
The frontend will use the query language Graphql to query from the backend proxy server.

JSON-RPC is a stateless, light-weight remote procedure call (RPC) protocol
To talk to an ethereum node from inside a JavaScript application use the web3.js library
JSON can represent four primitive types (Strings, Numbers, Booleans, and Null) and two structured types (Objects and Arrays).

Ethereum runs on a decentralized network. There is no central server. It's a peer-to-peer system. Each node can read and send data to other node.
All nodes can request from another node information about Ethereum’s current state such as smart contract, account balance. To facilitate this form of communication,
Ethereum uses a Kademlia-like protocol for node discovery. Each node has an id which is hashed. Each node stores a table of known nodes. To look for a node
the node can asks other known nodes to look for a node.

The Design Philosophy behind Ethereum.

*Simplicity*: The ethereum protocol is designed to be as simple as possible at the cost of time efficiency and storage so that anyone can implement their own blockchain.

*Modularity:* Ethereum is designed to be modular and as separate as possible. If you made a small protocol modification in one location, the application stack will continue to run.

*Non-discrimination and non-censorship:*The protocol will not restrict or prevent specific use cases. Ex. You can run an infinite loop on the blockchain as long as you are willing to pay the per-computational-step transaction fee.

## Global Control Flow

The system is event driven. The user interacts with the system by first registering his username and password. The user login and the next step is the user input the necessary data to the system and has access to view their data and standing compared to public data which is obtained using blockchain service and is presented in form of bar charts and line graphs. The system does not log anyone in until the initial event is driven by an external request. For time dependency, the backend of the system periodically processes pending data.

To tackle the issue of concurrency in our application. Considering the use of blockchain, having many threads accessing the same memory(multithreading) can produce race conditions that are very hard to reproduce and fix.

For a better solution than a multithreaded approach, our code runs things in parallel, which refrains us from creating new threads and allowing the need to sync them. The virtual machine and the operating system run the I/O in parallel and for sending data back to the Javascript, the JavaScript part is the one that runs in a single thread.

Our code consists of small portions of synchronous blocks that run fast and pass data back end and front-end processing functions. This process is faster since it doesn't block the execution of other pieces of JavaScript. Our application just invokes the function and does not block the execution of other pieces of code. It will get notified through the callback when the query is done, and we will receive the required user health data.

To decrypt the values of the JavaScript query that returns a few thousand results of different user's health data. Task at the back end is performed in such a way that it will split into smaller synchronous code blocks that will notify Node.js to split the task into smaller chunks using callback functions and so it can continue executing pending things that are in the queue after receiving callback.

We are using asynchronous I/O approach. The code consists of small portions of synchronous blocks that run fast and pass data to different places where needed. This approach doesn't block the execution of other pieces of JavaScript.

There are no timers in the system with no time constraints, since the system runs on blockchain data. Any user who is familiar or new to the system can input their data by logging in and compare it with the public.

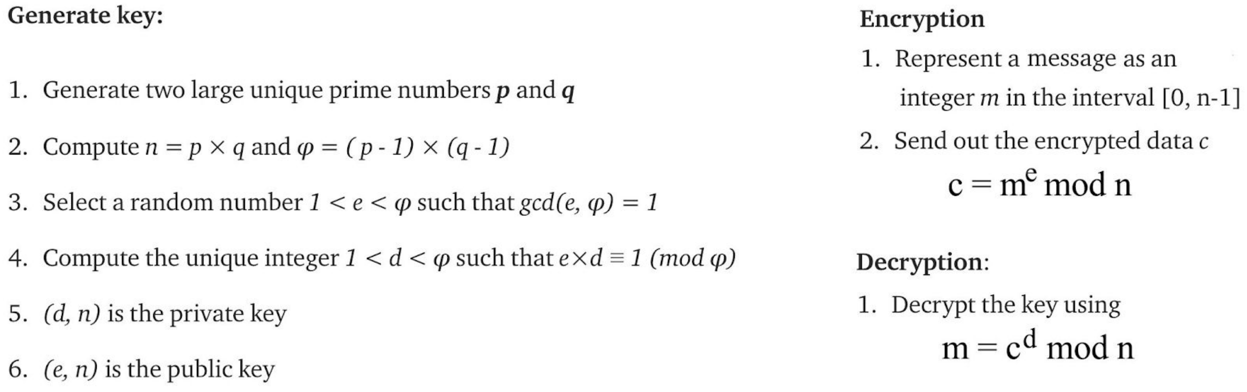
## Hardware Requirements

- Processor (CPU) with 2 gigahertz (GHz) frequency or above.
- A minimum of 2 GB of RAM.
- Monitor Resolution 1024 X 768 or higher.
- A minimum of 20 GB of available space on the hard disk (for blockchain nodes).
- Keyboard and Mouse or compatible pointing device.
- Network card should be enabled and installed to access internet.
- A bandwidth that can successfully load a modern Javascript application. Since there is no time sensitivity, a recommended 1 mbps should be more than enough.

# Algorithms and Data Structures

## Algorithms

RSA (Rivest–Shamir–Adleman)
We will be using RSA to encrypt the user’s data. Rsa is an asymmetric cryptographic algorithm, meaning that it uses a public and private key system. The public key can be shown to everyone, but the messages encrypted by the public key can only be decrypted by the private key. Here is a breakdown of how RSA encrypts data.



**Generate key:**

1. Generate two large unique prime numbers $\boldsymbol{p}$ and $\boldsymbol{q}$
2. Compute $n = p \times q$ and $\varphi = (p - 1) \times (q - 1)$
3. Select a random number $1 < e < \varphi$ such that $gcd(e, \varphi) = 1$
4. Compute the unique integer $1 < d < \varphi$ such that $e \times d \equiv 1 \ (mod\ \varphi)$
5. $(d, n)$ is the private key
6. $(e, n)$ is the public key

**Encryption**

1. Represent a message as an integer $m$ in the interval [0, n-1]
2. Send out the encrypted data $c$

$$c = m^e \bmod n$$

**Decryption**:

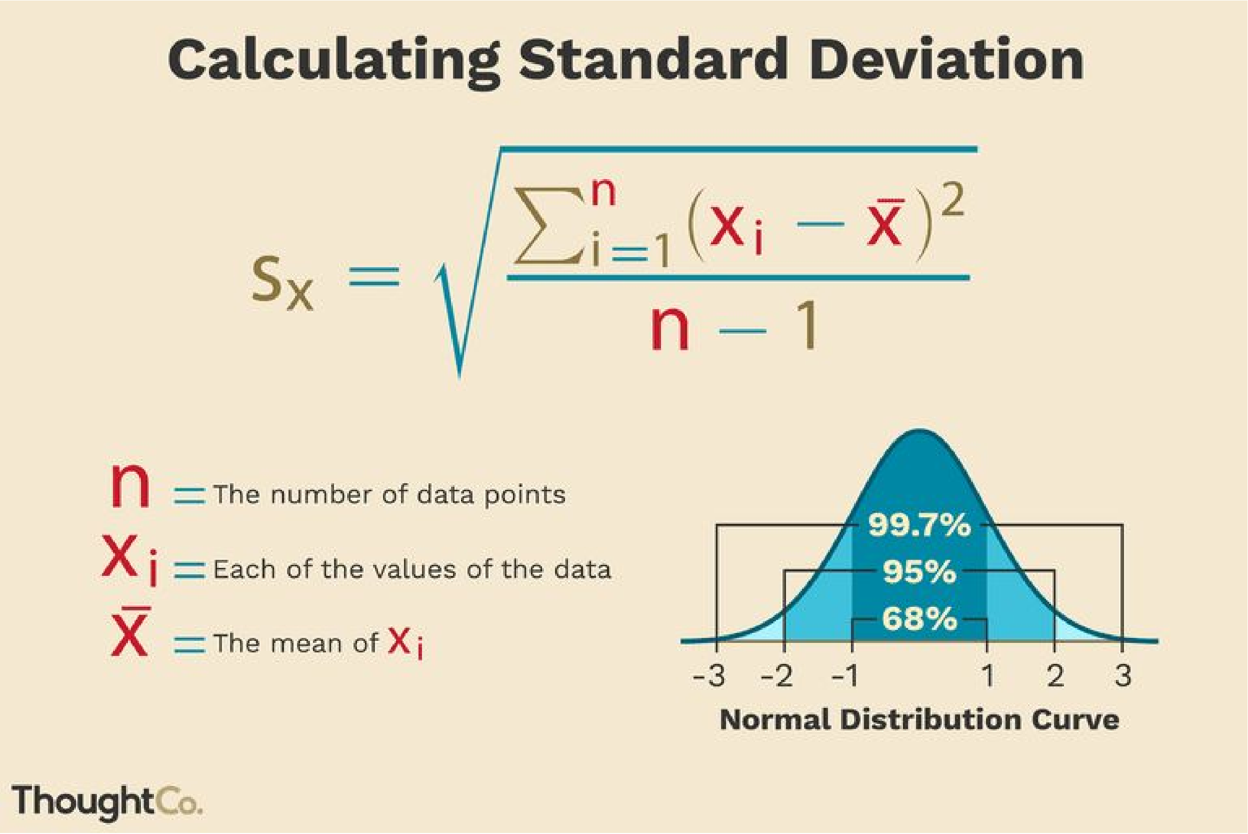
1. Decrypt the key using

$$m = c^d \bmod n$$

We will also be doing computations on user data. We will be finding the average and standard deviation of the data.
Average is the sum of values divided by the number of values.

Standard Deviation

The standard Deviation measures how spread out the numbers are in a set of data.


Calculating Standard Deviation
$s_x = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \bar{x})^2}{n-1}}$
$n$ = The number of data points
$x_i$ = Each of the values of the data
$\bar{x}$ = The mean of $x_i$
99.7%
95%
68%
-3
-2
-1
1
2
3
Normal Distribution Curve
ThoughtCo.

## Data Structures

Data structures that have relevance to the blockchain itself should be referenced in the Ethereum Yellow Paper. We will not discuss the use of Merkel Trees or Hashes that references previous structures as it is not directly related to our work. Rather, we can talk about the structures we use in our Smart Contracts (Etherium code running on the EVM) to store data about users.

Before we jump into specifics, it's important to understand how Hash Tables work, particularly in the Ethereum blockchain. It is a structure that offers fast access to a particular key and maps to a particular value. The structure does this by quickly hashing the key and looking at the key up on an indexable array. In ethereum, hash tables are virtually initialized such that every possible key exists and is mapped to a value whose byte-representation is all zeros: a type’s default value. The similarity ends here, though: The key data is not actually stored in a mapping, only its keccak256 hash used to look up the value. Because of this, mappings do not have a length or a concept of a key or value being “set”. Mappings are only allowed for state variables (or as storage reference types in internal functions).

### User Descriptor Smart Contract

Our smart contract that keeps data relevant to a particular user keeps user data on a few important data structures. The most important data structure used is the concept of a Hash Table (Hashmap or Dictionaries in many languages). For a user’s data to be secure, we created a Mapping of all user’s private addresses to their data. This is known to us as the User Descriptor table. When invoking to contracts to retrieve data for a particular user, only the user’s ID is used to access this table. As smart contracts are read only, as long as the methods defined use the current user’s address to access the data, user’s data are opaque to other users.

Our user’s are expected to store data for an infinite number of units; so what we have decided to do is create a second Hash Table that is used as the value of the User Descriptor table. This second Hash Table, known as the Unit table, is a key of units (‘lb’, ‘inches’, etc) to a value that is a growable vector of unit values. The unit values furthermore is a structure that stores a unit value, the timestamp of the submission of that value, and the latitude and longitude (if entered) for that particular submission.

One problem we had with this data architecture was that Hash Tables were not iterable. What this meant was that we could not easily iterate on the Unit table to retrieve the key units for the user. We had to supplement this Hash Table with a growable vector of units that we could present to the user before we accessed the Hash Table with the key unit.

We use the growable vector of unit values to easily index the end of the array and paginate responses to them. Since arrays are contiguous and indexable, pagination is as quick as possible.

As we traverse and transform data structures from the backend to the frontend, pagination data largely become growable Javascript arrays that the charting libraries can easily consume.

# User Interface Design and Specification

## Changes We Made

| Area of Change | What changed | Why we changed it |
|---|---|---|
| Graphical Visualization of Aggregate Population Data and Non-Graphical Data Display | Rather than offering a way to search for the graph the user wants, we offer all the graphs and non-graphical data on a single page. | This offers the user access to graphs and data that they might not have known about. Additionally, they can consider more information in relation to each other (i.e. consider weight, age, and sex simultaneously). |
| Data Entry Page | Rather than having user fill in a number value for a pre-assigned unit, the user may now choose from a list of offered units, or input a new unit, and simply explain its relation to the SI unit. After the first time they input this information, it is stored for their further use. | This offers the user to use units that they are more comfortable and confident with. Comfort of entry is key when entering in personal medical information. |

## Ease-of-Use Employed

A large portion of our process was making the User Interface as simple as possible. Certain ways we do so are by providing example inputs in the data entry page, graphical and non-graphical data in a combined page, scrolling over the graphs for more precise numbers, and offering the user the option to save the graphs.

Example inputs are placeholders in the data entry text boxes for numbers, units, and sex, until replaced by entered text. Seeing this allows the user to navigate the page easier on the first time, and recognize more clearly which text entry boxes are associated with which prompts.

Graphical and non-graphical data being all presented together offers a singular page to look through, rather than having to navigate through a search bar and find the exact one they need.

Offering the user to scroll over their graphs to see more precise numbers allows the user to understand precise values when wanted, without having to scroll through a table of data or having to approximate based on the graph and its axes.

Offering the user the option to directly save their graphs allows for an easier way of saving graphs that a user might want to share with their friends, family, doctors, etcetera. If not offered, the user would have to screenshot and crop out their preferred parts of the graph, whereas here we offer the user to circumvent that effort by giving them the option to simply save their information directly.

# Design of Tests

## Test Cases

1.

| | |
|---|---|
| Test Case Description: Test for user descriptors. This test will be used to check the user's inputs for gender, weight and weight units, and height and height units.<br>Related Use Cases:<br>Criteria for Passing: The units of weight must be pounds or kilograms; gender is male, female, other; and height is meters/centimeters or feet/inches. | |
| Fail Procedure: | Pass Procedure: |
| First try to input the wrong information, such as typing the incorrect type of units or invalid text, or not writing male, female, or other in gender.<br>The user after inputting the wrong information will be alerted with an error and will be prompted to change the input to the given valid inputs. | Input the correct information for weight, height, gender.<br>The user's correct information will be recorded by the system in order to create the user's health stats. The user will be able to proceed to use the app. |

2.

| Test Case Description: Integration test for GraphQL api. The test determines if the GraphQL endpoint gives an OK response<br>Related Use Cases:<br>Criteria for Passing: Test finds the status code of GraphQL, which is 200 | |
|---|---|
| Fail Procedure: | Pass Procedure: |
| GraphQL endpoint does not have a status code of 200.<br>The endpoint does not work and the test fails. | GraphQL endpoint does exist. The test looks for the 200 status code by requesting server to expect this code.<br>The server finds the status code, so GraphQL endpoint works. The test passes. |

3.

| Test Case Description: For global descriptors.<br>Related Use Cases:<br>Criteria for Passing: | |
|---|---|
| Fail Procedure: | Pass Procedure: |
| TBD - Report 3 | TBD - Report 3 |

## Test Coverage

We are using unit testing and integration testing in this project. This means we are able to account for most errors that may occur, because unit testing may check a component (unit) of the code but integration testing will check for errors that may occur combining these components. We will continue to make more unit and integration tests before the second demonstration to make sure there are no errors. Otherwise we have covered most of the major errors that the application could possibly have.

## Integration Testing

Our integration testing has a top-down approach. We will test the big concepts and modules of our code before moving on to the smaller modules. These big concepts, which are the GraphQL API, global descriptors, and user descriptors, take priority for our group.

## Plans for More Tests

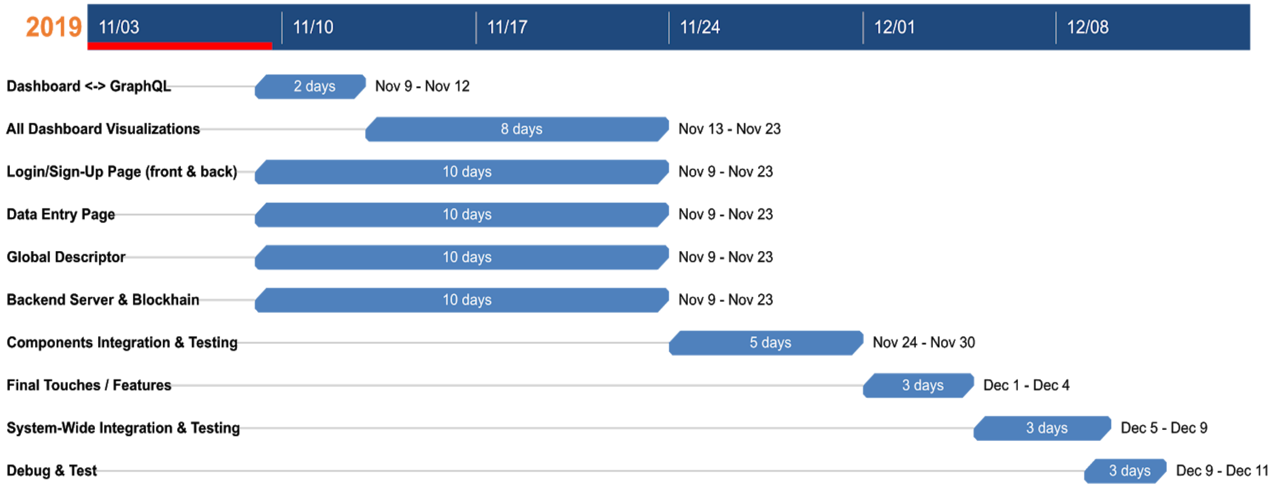
We plan on testing more requirements or use cases such as user authentication, public access of health data, and data administration

# Project Management and Plan of Work

## Project Coordination and Progress Report

| Use Case | Status | Status Details |
|---|---|---|
| UC-1: Login | Active | In development |
| UC-2: Auth | New | To be implemented |
| UC-3: InputData | Active | Data entry form exists, more input fields to be added |
| UC-4: ReceiveDataForUser | Active | User data available, more to be developed |
| UC-5: CompareData | Active | Comparative data available, more to be developed |
| UC-6: PublicAccess | New | To be implemented |
| UC-7: DisplayVisualAnalytics | Active | Visualizations available, more to be developed |
| UC-8: LogoutUser | Active | In development |
| UC- 9: Register(Account Creation) | Active | In development |
| UC-10: Data Administration | Active | In development, some smart contracts implemented |
| UC-11: User Notification | Active | Implemented in data entry, in development for login and sign up pages |

# Plan of Work



# Breakdown of Responsibilities

*Integration*: System-level integration will be coordinated by team lead Khalid Aaksh. He will direct the parties of corresponding components to both perform the integration and test functionality. As larger-scale integration continues, responsibilities will be adapted.

| Developer | Current Responsibilities |
|---|---|
| Khalid Aaksh | - Exploring Web Crawler for suggestions<br>- Performance optimizations in backend |
| Rizwan Chowdhury | - Login/SignUp Page (backend) |
| Dang Khoa Dinh | - Login/SignUp Page (frontend) |
| Smeet Kathiria | - Analytics Dashboard<br>- Connect Components to Mock Server (GraphQL) |
| Eric Rivera | - Analytics Dashboard<br>- Connect Components to Mock Server (GraphQL) |
| Suva Shahria | - Global Descriptor |
| Hersh Shrivastava | - Login/SignUp Page (frontend) |
| Nathaniel Arussy | - Data Entry Page |