Orange Team

Health Agent Assistant
Software Design Specification

Arif Topcu
Dan Chu
Dhonovan Hauserman
Dominic Defuria
Eric Smith
George Holland
Jared Kaczynski
Lucas Meira
Kavya Krishna
T.J. Wilder
Tyler Zentz

Table of Contents

Section 1 - Introduction

This Software Design Specification (SDS) describes the system design for a program designed to assist with health inspection work. The focus of this document is on the basic database, database federation, and tablet user authentication design details. When combined with the designs assigned to other groups, the final product will be a functional design for a desktop application and a tablet application to store and retrieve various health inspection forms for restaurants, septic tanks, and water wells. This section details the purpose of the document, the intended audience of the program, and the scope of the program.

1.1 Purpose

The purpose of this document is to specify the design and implementation of the program in a clear and consistent manner. This document will focus on the design details dealing with the database, database federation, and tablet user authentication for which the Orange Team is responsible. The other teams will be dependent upon the Orange Team and this document will allow them to make sure their design specifications are compatible with ours, what they must provide us for what returns they need, and what each of our teams are responsible for dealing with.

1.2 Intended Audience

This document is intended to give developers (the front-end tablet and the web application team) a concrete definition of the design of the backend and the user authentication for the tablet. Other developers will be able to use this document to learn about the design considerations and restraints, and they will have a better understanding of the overall design architecture. In the design considerations section, developers will be able to see what assumptions the backend team is making, along with the limitations of the database and user authentication. In the next section, we give the developers a system overview and an overall system diagram. The architecture design can be found in the last section. Here the developers can learn about all of the API and technologies that we are working with. This will help them keep the technology they use in the front-end (tablet and web application) both consistent and compatible with the backend technology.

1.3 Scope

This document explains the design and architecture of the basic database, the database federation, and the tablet user authentication. It also explains the design considerations and restraints that resulted in the design and architecture decisions for each of the three sub-projects. Finally, it provides diagrams outlining the system both as a whole and in each of its sub-projects, along with descriptions of what those diagrams represent.

1.4 Definitions

Term	Definition
User	Staff who use the application.
API	Applications Program Interface.
UI	User Interface.
Front End	The part of the application the users interact with.
Back End	The server and database that sends, receives, and stores data to and from the front end.
Protocols	The rules of communication between systems.
Interface	A device or program enabling a user to communicate with a computer.
Query	The search in a database or search engine.
SDS	Software Design Specifications. This document, details how we will implement our program.
SRS	Software Requirements Specifications. A document detailing what the program must be able to do.
Web Application	The portion of the project on the internet. The user will use the web application as one of two ways to access data, the other being the tablet application.
Tablet Application	The portion of the project that is on a tablet. The user will bring the tablet with him for inspections and be able to access data on the tablet.
Database (DB)	The place where we will be storing the data needed for the project. We will be using SQLite3 for our database.
Module	A portion of a program that carries out a specific function and may be used alone or combined with other modules of the same program.
Federation	The federation server will take requests from the Tablet and Web applications and communicate pass them on to the database. The federation server handles communication between applications and the DB.
Node.js	From Wikipedia, “Node.js is an open-source, cross-platform runtime environment for developing server-side web applications.” We will be using Node.js for the federation server.
SQLite3	We will be using SQLite3 for the database portion of the project. SQLite3 is a SQL database engine that does not require a server.

UML	Unified Modelling Language, a way to represent how the data in a program will interact.
Passport	A user authentication library provided by Node.js. We will be using the Passport library for tablet user authentication.
JSON	File format for passing data between front-end and federation server.

Section 2 - Design Considerations

This section details the assumptions, dependencies, and general constraints of the program. The assumptions and dependencies will describe the assumptions that must be made in the design of the program. The general constraints will describe some of our design limitations such as budget for servers and equipment, limited internet access, etc, and how these limitations will be worked around.

2.1 Assumptions and Dependencies

The design of the back-end and tablet user authentication makes several assumptions about software and hardware, and has a few software dependencies. The first assumption we make is that the user will have a dedicated server and they will be communicating with the server through the internet. Since the user will be carrying a tablet to inspection sites, possibly in remote locations, we assume that a stable internet connection will not always be available. Tablet authentication will rely on a third party library and the user's password and username will be stored in a local database. Since the user should be able to access all of the records on their tablet, we must assume that the tablet has enough memory and capacity to store all current records along with future ones. In addition, the technology used in the front-end must be compatible and consistent with the technology used in the backend/server (Node.js and SQLite3).

2.2 General Constraints

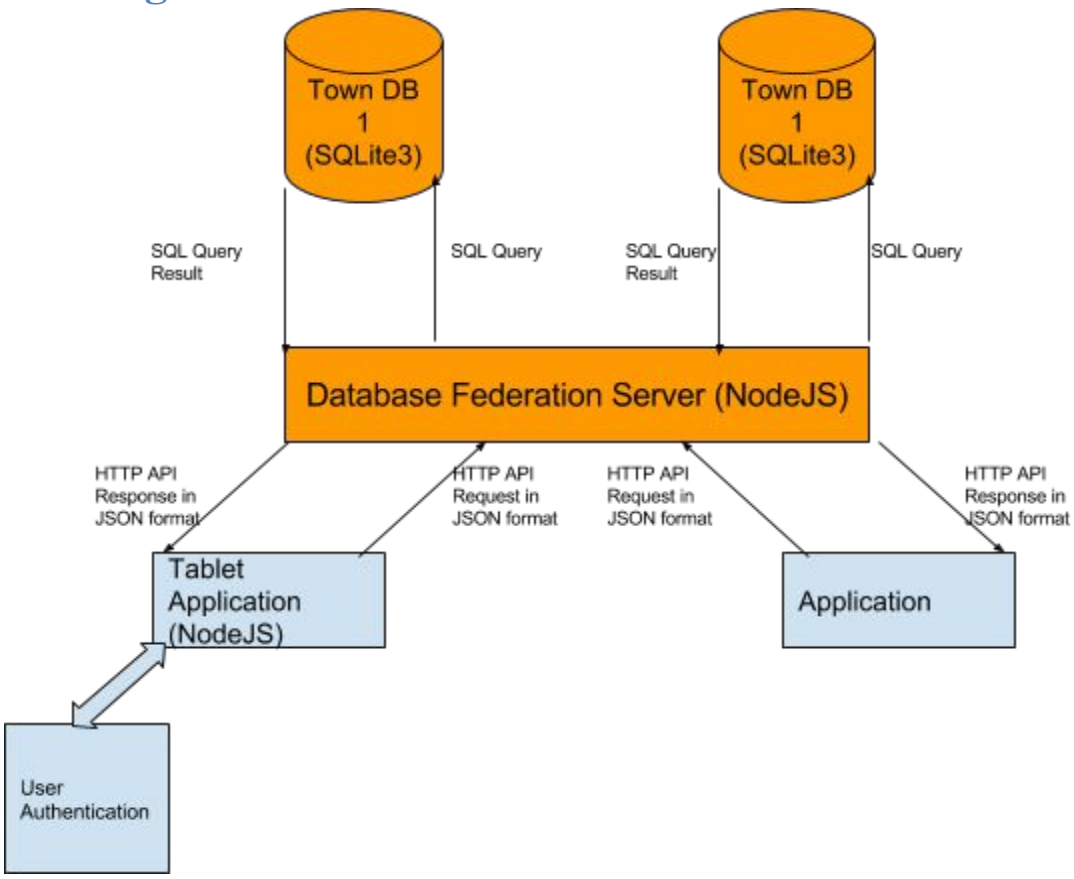
Database federation and basic database are constrained generally by the processing and memory resources available on the local government computers and tablets. Since the database will only contain textual and numeric information, however, these constraints are not realistically expected to impact performance and design. A lack of reliable internet for the tablet application means that both federation and the basic database must be capable of being stored and running on the tablet locally.

Tablet User Authentication is constrained primarily by the fact that an internet connection cannot be guaranteed during use of the tablet application. As a result, user authentication must be designed in such a way that user information can be correctly verified using only data available on the tablet. This same constraint also resulted in requiring a login token that can be verified locally on-demand, rather than being checked by the main database remotely. Because the main database cannot be reached while using the tablet application, account creation and password recovery are unavailable for tablet users.

Section 3 - System Overview

The main entry point for users is through the Federation Database Server. This is a bridge between the town database or basic database, and the external applications such as user authentication. The database contains all the information required for the system. The federation server handles and directs API requests.

3.1 Diagram



The main storage for all data is separated into town specific databases. There can be many of these but they will all have the same general setup and formatting, only differing with the data added and town it belongs to. They will be using SQLite3 as a database architecture. These are connected together for the purposes of this project with the use of a database federation server.

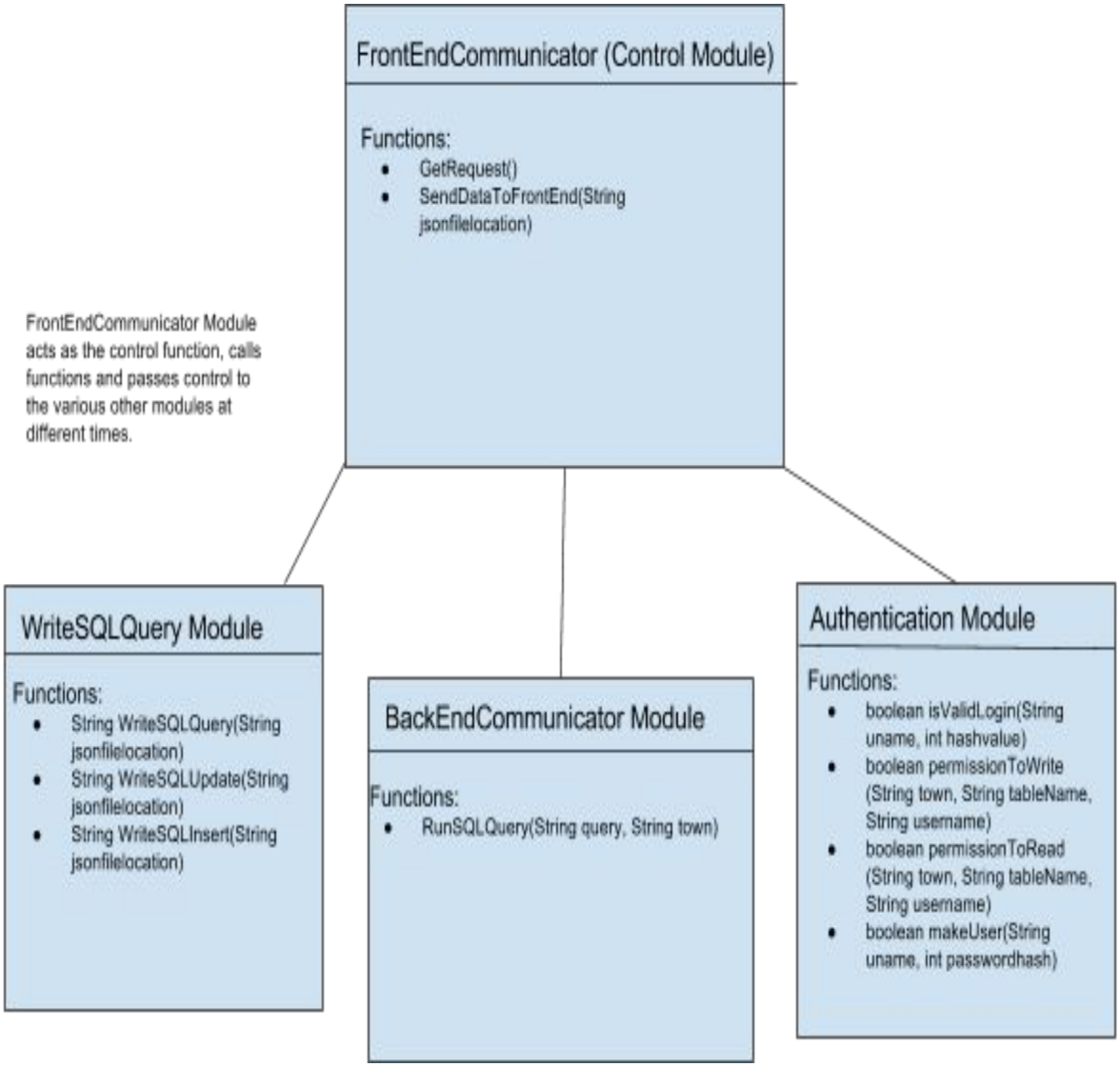
The database federation server is the main hub of the back end. This server will be coded in Node.js. It provides access to the databases for external applications with the use of a HTTP restful API. The federation server manages the directing of API calls and is the only way to interact with the databases for non server administration.

Above the federation server is any API call it receives. In this case it is from a tablet application and a web application. The type of application is irrelevant however as the API is the only way they interact with the database(s).

Section 4 - Architectural Design

This section describes the architectural design for the basic database, the federation server, and the tablet authentication. For the database we will use the relational database management system SQLite3, which can run cross-platform and is perfect to store the small amount of data produced from the health inspector's forms. The federation server will be implemented using the JavaScript framework Node.js. The Node.js server will receive requests from the application, process those requests to see if they are valid, communicate with the database to receive the correct data, then finally send a response back to the application with either the data requested or an error. User authentication for the tablet application will be implemented using the Passport Library and will have to communicate with the server in this fashion to check whether usernames and passwords are valid.

4.1 Server Class Diagram



4.2 Server Architecture Description

The above diagram represents the relationship between our four modules. The modules include "Authentication," "WriteSQLQuery," "BackEndCommunicator," and "FrontEndCommunicator." The authentication module will be used to establish, if the user who sends a particular request has permission to access or edit the data that was requested. For this we will use the library "Passport". We chose this library becauses it is the most popular authentication library to use with node and is proven to be secure and easy to use. The database query module will use the SQLite3 interface in order to turn the requests into database queries in order to retrieve the information from the database. SQLite3 is a good choice for our database because it is easy to use and works natively on android devices without internet connections. It was recommended by our professor for these reasons. The back end communicator module will take the requests for data and determine which town's database to query and whether it is a valid request or not. Finally, the front end communicator will act as the main module, where it will take the information that the other modules find and send the correct response to the user. Data will be sent and received through the use of JSON files. For testing we will use the mocha library.

The job of the authentication module is to store user authentication data if needed and to check whether a user is allowed to make a certain request. It will have functions like loginUser(username, password, hash), makeUser(username, password), which will login user if the correct credentials are found in the database, otherwise sending an error message, and creates a user account if the username and password do not already exist in the system. The other function would be something like hasPermission(user, JSON req), which would take the request and determine whether the user is allowed to do what they are trying to and if not send back an error. We will implement these functions using the Passport library in Node.js.

The database queries module is in charge of verifying if a request is valid and can be queried. If so it will create a query to find the data in the database. If the data is valid and retrieved correctly from the database, it will send a JSON file to the front end module and be forwarded along to the user. If the data is meant to be posted to the database, it will be verified and correctly added to the database. The functions we will use to accomplish this are verify(JSON req) and query(JSON data).

The back end communication module will distinguish which town's database the data is supposed to be retrieved from or posted to. It will do this using the function getTown(JSON file). The query module will use the information to form the correct query and send it to the front end communicator.

The front end communication module will be doing the communication with the user. It will initially receive the request from the user and then forward it along to the appropriate module. Once the request is dealt with, it will come back to this module and be forwarded to the user. If the request is to add data to the database a success or error message will be given to the user/front end. The functions we will use to implement this are app.get(req,res), app.post(), and other node.js functions a server uses.

4.3 API Architecture

For the federation server's open web API, request and response data will all be dealt with in JSON (Javascript Object Notation) format. The specific format for each request is outlined below and a few examples are provided. Note that all "/api/add", "/api/edit", "/api/remove", "/api/user/register", and "/api/user/login" calls must be done via html POST requests and all "/api/get/" requests must be done via html GET requests.

Main API Calls:

Required Fields for main API calls:

"location" - as a list (ie. ["Leverett"] or ["Leverett", "Sunderland"])
"type" (ie. owner, property, restaurant, restaurant inspection, violation, septic, system pumping record, well, water quality report, user, saved search) [this is almost any of the tables in the database]

/api/add

returns the id of the new entry or -1 if the operation failed
Must include every field listed in the database for the given table

Example:
POST → /api/add
Data: {
"location": ["Leverett"],
"type": "Restaurant",
"Name": "WcDonalds",
"Address": "999 Lois Lane",
"Telephone": "1113335555",
"Owner": "Bobby Malone",
"Person in Charge": "Frederick Malone"
}

/api/edit

returns a boolean stating whether the operation was a success
Requires "id" of whatever entry you are intending to edit
Any additional fields you include will be the new value of that field for the entry

Example:
POST → /api/edit
Data: {
"location": ["Leverett"],
"type": "property",
"id": 3,
"state": "New York"
}
This updates the Property in Leverett with ID 3 to have {State = New York} without changing any of the other properties

/api/remove
returns a boolean stating whether the operation was a success
“id” of whatever entry you want to remove

Example:
POST → /api/remove
Data: {
	“location”: [”Sunderland”],
	”type”: ”restaurant inspection”,
	”id”: 555
}

/api/get
returns a list containing the rows in the given table that match the search
Can include any/all of the database fields in the given table as search parameters.

Example:
GET → /api/get
Data: {
	“location”: [”Leverett”, “Sunderland”],
	”type”: ”restaurant inspection”,
	”type of operation”: 1,
	”risk level”: ”High”
}
this returns all restaurant inspections in Leverett or Sunderland that have a “type of operation” field equal to one (which is food service) and a “risk level” field equal to “High”, regardless of its other fields

One special case of /api/get is for time: Because specifying an exact time would be useless, you can specify any time value (time in, time collected, etc. that appears in the table) as a range (ie. “time in”:”100-1000” which will return only those data that have a “time in” value between 100 and 1000 [inclusive]). This allows you to search for entries like “between January 1st and February 5th” by calculating the range on the front end. This will, as with other fields, default to any time value.

/api/database
returns full databases of each location listed requested to be used by offline operations
Takes only a single field, a list containing which locations you want to retrieve the current database for

Example:
GET → /api/database
Data: {
	“location”: [“Leverett”]
}

User API Calls:

All saved search calls require a user to be logged in.

/api/user/register
returns a boolean stating whether the operation was a success
Registers a new user with the given username and password.

Example:
POST → /api/user/register
Data: {
"username": "4dm1n",
"password": "lolnopasswordplease"
}

/api/user/editpassword
returns a boolean stating whether the operation was a success
Either requires the given user to be logged in already or it requires an admin with sufficient privilege level.
It takes a Username and the new Password for that user. This is different than sending a /api/edit request to the User table because the system needs to hash and salt the password before storing it in the database.

Example:
POST → /api/user/editpassword
Data: {
"username": "4dm1n",
"password": "fineokpassword"
}

/api/user/login
returns a password hash the front-end application should use for all future requests by that user
Logs a user into the system.
Example:
POST → /api/user/login
Data: {
"username": "4dm1n",
"password": "fineokpassword"
}

/api/user/addsearch
returns either the id of the search in the table, or -1 if the operation failed
Adds a new saved search string.

Example:
POST → /api/user/addsearch
Data: {
"search": "IDon'tKnowHowYouFormatYourSearches, This is probably some JSON thing"
}

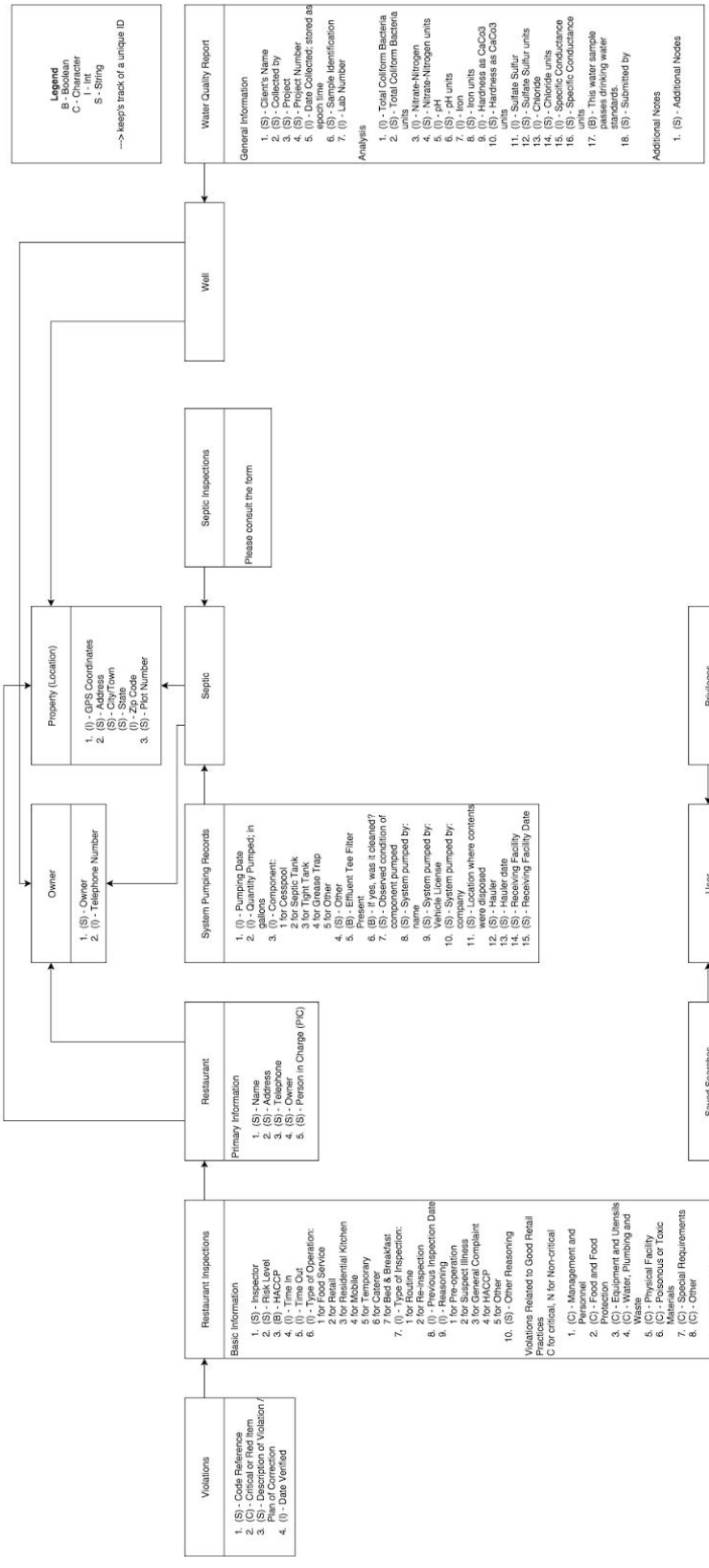
/api/user/removesearch
returns a boolean stating whether the operation was a success
Removes a saved search by id. The operation will fail if the saved search does not belong to the user. [Note: There is no way for a regular user to edit a search, the front-end should instead delete the old search and add a new, updated search should a search require a change after it's been saved]

Example:
POST → /api/user/removesearch
Data: {
"id": 9865
}
removes the saved search with an id of 9865 if the current user owns that saved search

/api/user/getsearches
returns a list of all the user's saved searches as: [{"id": 12, "search":"..."},{...}, …]

Example:
GET → /api/user/getsearches
No Data

4.4 Database Diagram



4.5 Database Architecture Description

Our database consists of 10 tables: Location, Restaurant, Restaurant inspections, Restaurant violations, Septic, Septic inspections, Wells, Well inspections, Users, and Privileges. The first eight will work in tandem with one another while the last two will be an entity of their own. Location will provide some general information as we realize that one particular site might have a restaurant, septic tank, as well as a well. The restaurant table will provide some details regarding the various restaurants such as name, owner, and person in charge. The Restaurant inspections table will keep track of the client's visits and what problems arose. The Restaurant violations table is on its own because we realize that there may be more than one violation per restaurant. For the most part, Septic, Septic inspections, Wells, and Well inspections should be nearly identical.

The database will be built with SQLite3 and will be stored on a server provided by the client. We chose SQLite3 for a number of reasons. First, it was recommended by Professor Ridgway. Second, it is touted for its ease of use. Additionally, it is lightweight and runs natively on Android.

4.6 Tablet Login Architecture Description

Tablet User Authentication receives a password and username from the user, then hashes the password and sends both to be checked against the local database's records. If the password and username are matched to a user record, the user is authenticated. If the details are not matched to any user record, the user is prompted to try again. This is accomplished using the Passport library, and specifically with the passport-local strategy. Once a user has been authenticated, Passport will give them a security token that will be required to access the main program. Passport was selected for tablet login because some of our members have prior experience using it, it is secure, simple to use, and one of the most common login libraries for Node.js. Passport-local was selected because it is the most popular strategy that does not require an internet connection to authenticate users.