Software Design Document

WearWare Study Manager

Sponsored by Dr. Kyle N. Winfree and Dr. Eck Doerry
Mentored by Han Peng

Members:
Anton Freeman, Halcyon de la Rosa, Karina Anaya, Noah Olono

Version 2.0
2/14/2022

Table of Contents

1. Introduction

Heart disease is a major problem for many people in America. In fact, according to the Center for Disease Control and Prevention, nearly 1 in 4 deaths in America every year are caused by heart disease. To analyze the issue and prevent further deaths, researchers need to gather health information (such as sleep, heart, and fitness data) on people with heart disease so they can figure out how to treat it. This health information gathering is known as the field of bioinformatics, and data collection on study participants is often done with the use of tools like heart rate monitors and fitness trackers.

The field of bioinformatics is incredibly important to people involved in the health field, such as doctors and medical researchers, as its results can help to save lives. For example, bioinformatics could be used to find an exercise routine or medication which could help people recover from heart disease and thus help to save the lives of the previously mentioned 1 in 4 people who die from it. In fact, results from studies using bioinformatics could end up influencing the National Institute of Health's views, policies, and recommended treatments, meaning that they could affect the lives of every single person in America. Due to the importance of bioinformatics, the field has thousands of researchers, as well as multiple dedicated major annual conferences and research journals dedicated entirely to it.

Our clients Dr. Kyle Nathan Winfree and Dr. Eck Doerry, an associate professor and a professor from Northern Arizona University, are working to introduce a new way of collecting data to the field of bioinformatics by creating a technology that will allow researchers to use Fitbits to collect information on study participants without needing to go through an existing service. Fitbits are small wrist-mounted consumer devices that can be used to collect information such as heart rate, exercise, steps, and sleep data from people wearing them. Many of the current tools used in bioinformatics, such as heart rate monitors and fitness trackers, are expensive to use and can result in biased data; our clients report that often study participants are only given these tools for about a week, during which time the participants will alter their behavior because they are actively aware that they are being observed. This results in inaccurate data when compared to a participant's normal life and behavior. Using Fitbits would lower the cost and increase the amount of time that participants could wear Fitbits compared to other measurement devices, which would make it easier for participants to adjust to their vitals being measured and allow researchers to get accurate results. Data collection with Fitbits could help almost any researcher who needs to gather heart rate, sleep, or fitness data, and as a result this project could produce value by allowing these researchers to get good results and help save lives. Additionally, the size of this business is very large, as it can potentially help any researchers in the field who have an interest in collecting this data while also reducing costs.

Although our client has already implemented a system to allow for data collection with Fitbits, the current system is plagued with a number of issues that prevent it from working properly and efficiently. Some problems with this implementation include the following:

- Exports take a long time and lock up the entire system, preventing users from doing anything else until the data is fully exported.
    - This long export time also causes issues by sometimes going for so long that it causes timeout errors, which results in users getting incomplete datasets from their studies.
- Researchers must enroll all participants manually, which can be a time-consuming process and limits the possibility of participants self-enrolling.
- Fitbit sleep data is not collected, which could be very valuable for tracking abnormal sleep patterns which are associated with disease.

Due to the issues that our client's current system has, we will be replacing the existing system with an improved one that will both solve the problems with the previous system and add improvements to make it even better than a basic fix would be. Our proposed solution is a secure, reactive web app that will allow researchers to create studies, enroll participants in those studies, collect data from participants and monitor the studies during data collection, and finally export that data for use in their own research. Some new features that this web app will include to address the previous system's issues are the following:

- Exporting will be done alongside the web app via a background task running in parallel to it, allowing for multiple exports at once while also preventing the web app from locking up during data export.
- Data export will be much faster, preventing any issues with timeout errors that could cause researchers to get incomplete data sets.
- We will collect all the data that Fitbit provides, allowing researchers to have a comprehensive view of their participants' health data.
- Researchers can choose to enroll participants manually or allow participants to self-enroll using a link, which will make gathering participants easy and simple if so desired.

As you can see from these proposed improvements, we have a solid plan on what we need to implement to address the previous system's deficiencies and make a quality product for our clients. Once we finished our plan, we then went through requirements acquisition with our client to figure out the key functional , performance (which define how our project will act), and environmental (which are defined by client request or other restrictions) requirements for this project so we could know exactly what is required. These requirements include the following:

- Functional requirements (which define our product's functionality):
    - Users will be able to log in using OAuth2 to access our web app.

    - Users will be able to create and manage studies within the web app.
    - Users will be able to enroll participants or create links for participant self-enrollment.
    - The web app will request Fitbit data using the Fitbit API.
    - All data will be stored in a relational database.
- Performance requirements (which define how our product will run and our user experience):
    - Exports must all be completed and returned within 12 hours of their requests.
    - Exports must not time out or error.
    - The web app must still be accessible during data export.
    - Errors must be human readable and include information for troubleshooting.
    - Exports will notify the users that requested them after they are completed.
    - The web app must be navigable with a tree structure and breadcrumbs.
- Environmental requirements (which are defined by client request or other restrictions):
    - The web app must be hosted on AWS.
    - The web app must run on Ubuntu 20.04 LTS.
    - Our web server must run within a Docker container.
    - Previously stored exports must be deleted when storage is low.
    - Our web app must pull data at regular periods in accordance with Fitbit specifications.
    - In the case of participants joining late or server outage, our web app will backfill any needed participant data.
    - Our database must be regularly backed up.

With all of these requirements, we have a solid foundation of what is needed for this project that we can use to make a fully functional web app that will suit all of our client's needs.

2. Implementation Overview

Now that we have introduced our project and shown its planned features, we can move on to the specifics of what we are building. As previously stated, our solution vision is a secure and reactive web application that will allow researchers to collect Fitbit data from study participants for their research and then retrieve that data using our product. This web app will consist of three main parts. One is the frontend, which users will interact with via a web interface. The other two parts are the backend and database, which will control everything running behind the scenes and hold all of our data respectively. All database connections to our frontend will be done via the backend, which will be stored in a separate container for easy portability. We are also using a producer/consumer model so we can have processes like data export running in parallel to other parts of the web app, allowing researchers to work on multiple datasets at the same time without causing the rest of the web app to lock up. Additionally, we have figured out the different frameworks and languages that we are using to create this web app.

For our frontend, we are using Javascript along with the web application framework Vue.js, also simply known as Vue. Vue is a template-based framework used for building frontend web applications that will allow us to easily create a responsive and usable frontend for our users. Notably, its use of templates will allow us to recycle components instead of having to rewrite them from scratch for every page, allowing us to make a website that is easily scalable and can support any number of different studies without any issue.

On the backend side, we have also employed Javascript with the use of NodeJS and the NodeJS Express framework to run a server which will handle everything that goes on in the background of the application, outside of user view. We have chosen NodeJS because of its speed and the familiarity that the team already holds with it, which means that we will not have any problems with getting it up and running, thus increasing our efficiency.

To store the Fitbit information collected for our users, we needed a strong database that could handle many different parts (such as data intake and exporting) without having speed issues or encountering timeout errors. Because of this need, we have chosen MariaDB as our database framework. In addition to being specifically requested by our client, MariaDB boasts high performance and allows for parallel queries, which will help to prevent issues of the system locking up during data export. Additionally, we will be using Bash scripts to regularly create backups of our database so that nothing will be lost in the case of database failure.

Finally, for production we will be containing the different parts of our backend within Docker containers. This will provide a clean environment and allow for easy troubleshooting, while also making all the different parts of this project very portable and

thus easy to move to any machine that our client will be using to run it. With these parts combined, we have a solid foundation of different technologies that we will all use to build our project.

3. Architectural Overview

Our system will require many components to perform the tasks and functions required for a working system. All these components will have methods in place which will allow information to be used and shared between each component. The purpose of this section is to give a high-level overview of those components, the methods used for communication between components, and the architectures implemented by the components, all of which will be described in this section and the architectural diagram below.

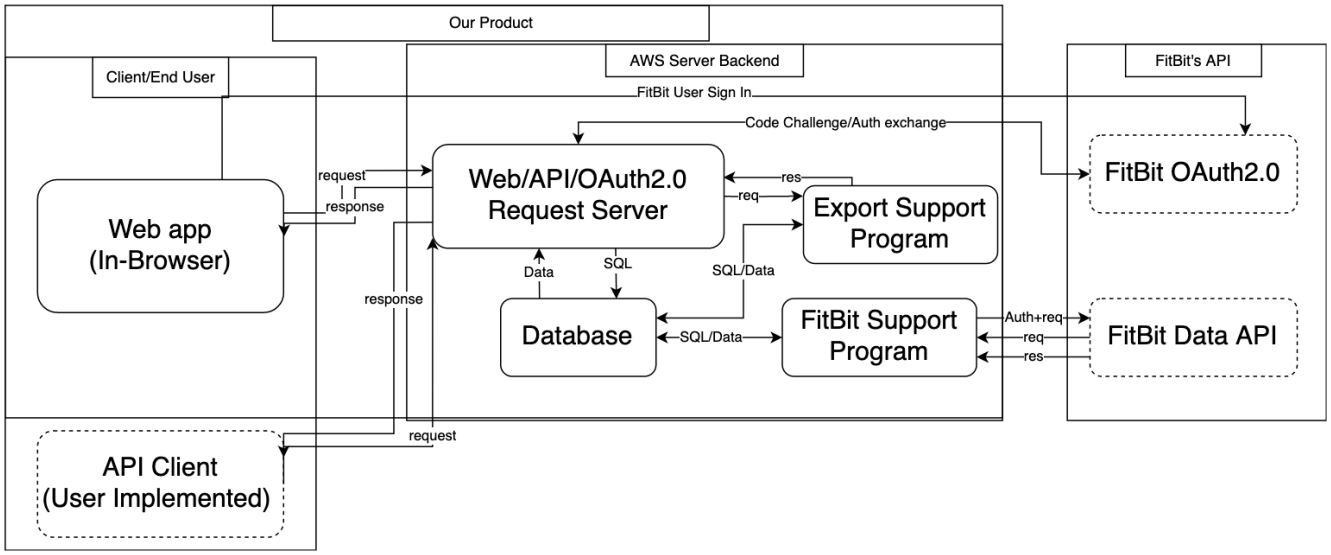

Figure 3.1: A high level architectural overview of our system.

There are a total of five important components needed for our system. These components are shown in the diagram above and are as follows: the web app component which will serve as our frontend, the Web/API/OAuth2.0 request server, Fitbit support program, and Export support program components which will serve as our backend, and our database system.

3.1 Web app

The web app component's purpose is to provide a navigable user interface to communicate with our server from the client's system. It will allow users to log in and authenticate. After login, users will be able to see their studies and modify existing studies or create new studies as needed. Users will be able to upload existing participants, generate links to enroll new participants or have participants self-enroll.

Participants will be able to use researcher-generated enrollment links to sign up for studies without making an account.

The web app will communicate with our Web/API/Oauth2.0 server instance on the AWS backend using formatted HTTP requests and responses encoded using an SSL certificate. When a new resource is needed a request will be generated and sent to the request server. If that resource is available and returned by the server, the web app will use the response to create a local object available to the user.

The web application functions as the client in a client-server architectural style by requesting needed code, styles, and resources from a central server. Additionally, the web application also functions as a graphical user interface layer of a 4 layer multi-tier architecture. This provides an abstraction that allows the end-users to interact with the system without needing to directly interact with the complex lower layers/tiers.

3.2 Database

The database will function as the main data layer for the service and will hold the user, study, participant, and participant activity data. Using a relational model and MariaDB for our database, the data will be easily accessible to the Web/API/OAuth2.0 server as well as the support programs.

The database will communicate with three other components: the Web/API/OAuth2.0 server, Fitbit support program, and export support program. All incoming data from these components will be presented as SQL requests or commands. All outgoing data will be formatted into a collection of individual row data for rows matching the requests, with the data types depending on the makeup of the individual tables.

As mentioned above the database acts as the data layer for our multi-tiered architecture providing information to the server and support layers as well as accepting new data from those layers.

3.3 Web/API/OAuth2.0 Request Server

The Web/API/OAuth2.0 request server will be responsible for handling incoming requests from the web app and client/user implemented API program and creating the responses to those requests. The request server will support multiple URL paths and arguments corresponding to the different functionality available to the web app, and client/user API program. This includes login/authentication, study functions, and participant functions. The API features will be read-only.

The Web/API/OAuth2.0 request server will communicate with the web application and the user implemented API client using HTTP requests which will be sanitized and routed to the correct request handler function. Messaging between the request server

and the export support program will take place over local socket communication by authenticating and authorizing the request and then passing it to the support program to be executed. The server will also handle first-time Fitbit authentication by handling Fitbit's code challenge and saving the auth token information.

This component is the server in a client-server architecture and handles all resource requests from the web application and user API program. The request server acts as an abstraction layer or the logic tier of the multi-tier architecture between the web application and the backend support programs passing requests. Additionally, all request handlers for API resources will implement a RESTful type architecture, caching wherever available to improve performance.

3.4 Export Support Program

The main function of the export support program is to manage all requested data exports in a secure and timely manner. This program will maintain an export queue and current export progress log. This program will run continuously in the background communicating with the web/API server and database about exports and export progress Using the log the support program will also be able to resume exports from the last successful point in the case of a crash or stall.

The export support program will receive export requests passed on by the server over a local socket and query the database for information about how large the dataset required for the export will be and how long the export is expected to take to complete. Additionally, the support program will accept requests from the request server for the status of the specific exports and notify the request Web/API/OAuth2.0 server when an export is available, including in the response the specific file location for the export.

The export support program acts as a secondary resource server to support the main Web/API/OAuth2.0 request server with large workloads. This is done by implementing an internal multithreaded architecture to work on multiple large exports concurrently and pass a reference back to the request server acting as the worker tier in the systems multi-tier architecture.

3.5 Fitbit Support Program

The Fitbit support program will be responsible for handling the post OAuth2.0 authorization communication with the Fitbit API and our local database to ensure participant information and activity are regularly updated and logged. This program will run continuously in the background issuing requests to Fitbit for every known participant in our database and waiting for the subscription responses. In the case of Fitbit API outage, this program will also backfill data once it is available to ensure timeline integrity.

The Fitbit support program will only need to communicate with our database through SQL requests for participant data and the Fitbit API using the stored OAuth2.0 tokens and refresh tokens. Fitbit will notify the server of updated or newly available data using webhooks, or as Fitbit calls them subscriptions, which will cause the program to request the new data and add it to the database.

This component is the client in the client-server architecture between the Fitbit API and our system. It exists alongside the export support program in the system worker tier handling a large amount of participant data in the background to prevent overworking the request server.

4. Module and Interface Descriptions

This section will dive deeper into the five components outlined in the previous section and describe the details of each component. The dive will also cover all the main internal classes within each component and their public interfaces. For the sake of clarity and simplicity, we will also combine the Fitbit Support Program and Export Support Program components into a single support program subsection.

4.1 Database

4.1.1 Component Description

The main responsibilities of the database are to provide an organized system for data storage and retrieval by the other components of the system. Our database sits at the deepest level of our system with the bulk of the requests being small requests for the high level user, study, and participant data by the request server. Larger export and update requests will be served by the appropriate support programs.

4.1.2 UML Class Diagram and Descriptions

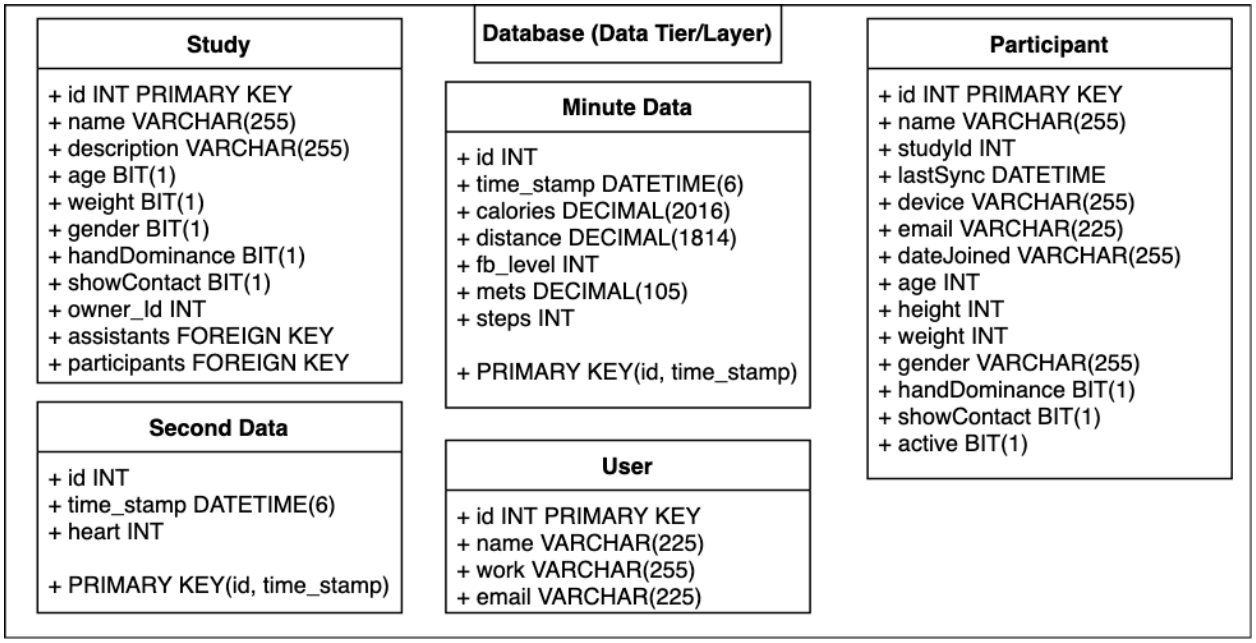

Figure 4.1.1 A diagram of tables and attributes included in the database module

- **User**
  A table in our database containing the basic information about the users registered using local accounts. Used to provide personal information when requested or allowed by the user. Additionally, user data here helps track which user made changes to a study in the given log file information.

Attributes:

- **id**: Local unique identification number used to identify which information belongs to which user, must not be used as the sole method for authentication or authorization. This value is automatically generated when a user first signs up for an account.
- **Name:** A string containing the name provided by the user or through an OAuth profile request.
- **Work:** An attribute to hold the organization or position name if one is supplied by the user.
- **Email**: A string containing the email address provided by the user or through an OAuth2.0 profile request for contact.

- **Study**

  A table containing the important information or metadata for all the studies created using the web app. Used to track who owns or created the study, who is working on the study or has access to the information, and what information the study should ask participants for when signing up. The study table also contains information about which participants are active in the study.

  Attributes:

  - **id**: Unique id number assigned to the study at time of creation used to query the database.
  - **name:** A string containing what the study is called, provided by the user during the creation process.
  - **description:** A short description of the study, what data it is collecting, and what the goals of the study are. Will be provided by the user during the creation process.
  - **age:** A flag that will specify whether the study will track age data from participants.
  - **weight:** A flag that will specify whether the study will track weight data from participants.
  - **gender:** A flag that will specify whether the study will track gender data from participants.
  - **handDominance:** A flag that will specify whether the study will track right or left handedness data from participants.
  - **showContact:** A flag that will specify whether the study will be able to contact participants who opt-in and provide contact data.
  - **owner_Id:** The id number of the user who created the study

- **assistants:** A list of other users who have access to the study identified by their local user id.
- **participants:** A list of all participants who have successfully been added to the study.

- **Participant**
  A table containing the provided personal information for all the participants registered in the system. Used to provide demographic and contact information to researchers/users where appropriate and as allowed by the individual participant during the enrollment process.

  Attributes:
  - **id**: Unique id number assigned to the participant when they sign up for their first study.
  - **name:** String holding the participant's name if provided during sign up.
  - **studyId:** The id number of the first study the participant attempts to enroll in.
  - **lastSync:** The timestamp of their last device synchronization cycle as provided by the Fitbit API.
  - **device:** The device model and type as provided by the Fitbit API.
  - **email:** String containing email address of participant if provided.
  - **dateJoined** Date that participant enrolled in first study using the site.
  - **age:** Number representing participant's self-reported age.
  - **height:** String containing participant's self-reported height.
  - **weight:** Number representing participant's self-reported weight.
  - **gender:** String containing participant's self-reported gender.
  - **handDominance:** Flag showing whether the participant is right or left hand dominant.
  - **showContact:** Flag showing whether the participant has opted to let researchers/users use their contact information to contact them.
  - **Active:** Flag showing whether the participant has opted out of further data collection.

- **Minute Data**
  The minute data tables hold all participant activity that comes in minute by minute from the Fitbit API. This includes data from calorie, Fitbit_level, distance traveled, metabolic units, and steps. Each row in the table is a one-minute datapoint for the participant.

  Attributes:

- **id**: Local unique identification number used to identify which information belongs to which participant.
- **time_stamp:** A date object containing the year, day, hour, minute, second, and microsecond provided by the Fitbit API for the given category data values.
- **calories:** Decimal number containing the amount of calories Fitbit estimated the participant burned in the specified minute of time.
- **distance:** Decimal number containing the distance Fitbit estimated the participant has traveled in the specified minute of time.
- **Fb_level:** Metric used by Fitbit to determine effectiveness of workout.
- **mets:** Metric used by Fitbit to determine effectiveness of workout.
- **steps:** Decimal number containing the number of steps recorded by Fitbit in the specified minute of time.

- **Second Data**
  The second data tables hold all participant activity that comes in second by second from the Fitbit API. At the moment this is only data from intraday heartbeat readings. Each row in the table is somewhere between a one second and one minute datapoint for the participant.

  Attributes:
  - **id**: Local unique identification number used to identify which information belongs to which participant.
  - **time_stamp:** A date object containing the year, day, hour, minute, second, and microsecond provided by the Fitbit API for the given category data values.
  - **heart:** Integer number containing the participants BPM values for the given second interval**.**

4.1.3 Public Interface Description

Our database will not have any functions, as it simply stores all of the data that we will be using for this project. All services involving the database will instead use the backend or the support programs to gather and send information from or to the database via SQL queries sent by the handler functions as defined in section 4.3.2 or the DBConnector as defined in section 4.4.2.

4.2 Frontend Web App

4.2.1 Component Description

The frontend web app component consists of all code and objects provided by the server to the client machine's browser and will be the main interface for users to interact with the components of the backend system.

4.2.2 UML Class Diagram and Descriptions

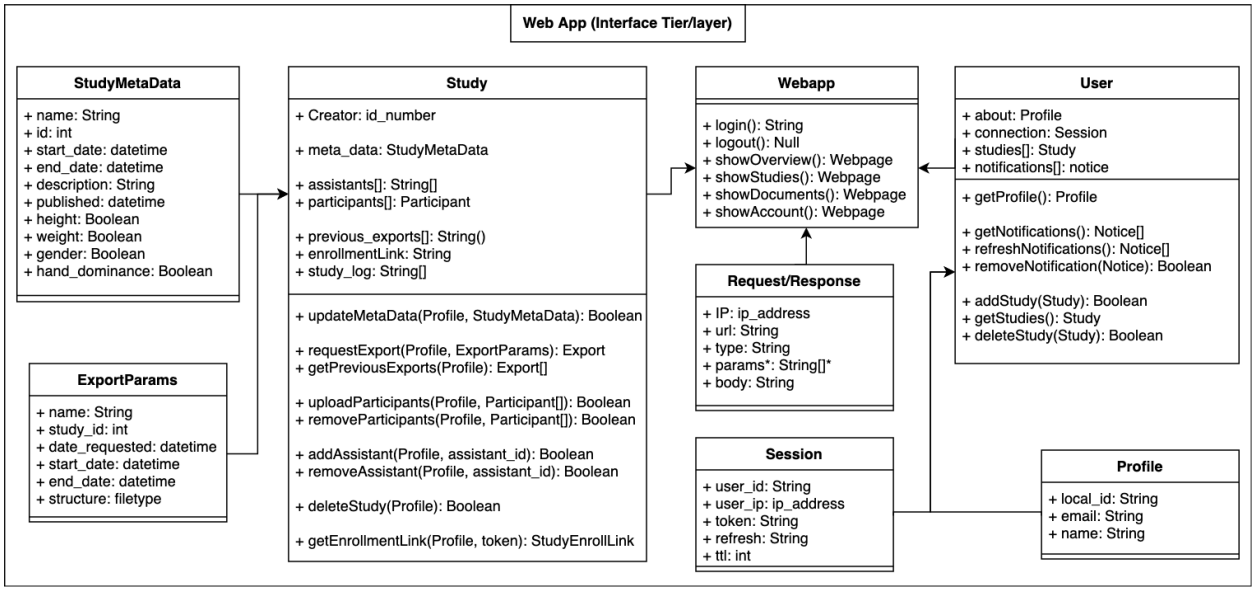

Figure 4.2.1 A diagram showing our web app component's classes.

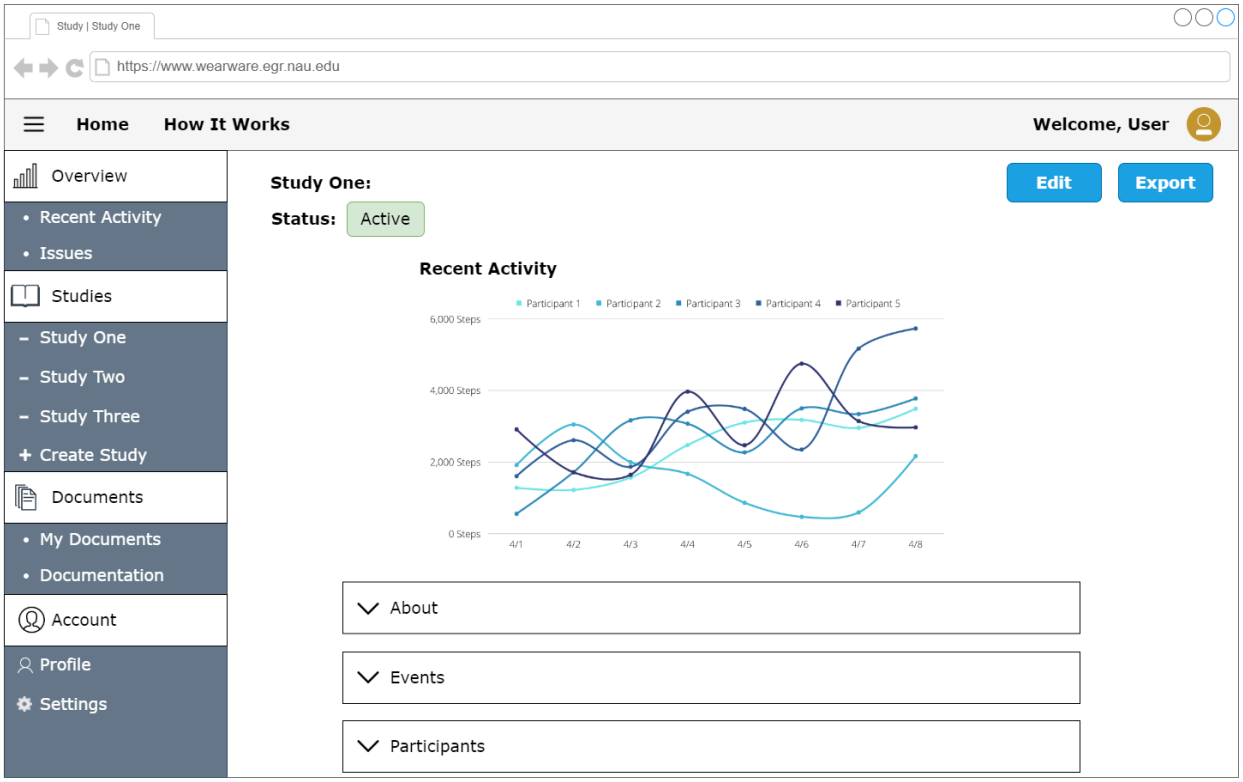

Figure 4.2.2 An example mockup of the user's view web app interface.

- **Web App**
  A user interface that will allow users to interact with the system. The frontend is the main way that our users will interact with the system outside of the API. Due to its nature as a front-facing application it will not do much computation on its own; instead, it will mainly rely on the backend to accomplish most things.

  Methods:
  - **login(String):** Method used to create user authentication requests and capture required session packets.
    - Return Type: String
  - **logout():** Method used to notify server that user has requested session termination and will require a new login request regardless of timeout on next site login.
    - Return Type: Null
  - **showOverview():** Method used to request and display the recent and important changes to participant activity to the user after login.
    - Return Type: Webpage
  - **showStudies():** Method used to display the studies to the user after login.
    - Return Type: Webpage
  - **showDocuments():** Method used to show user's saved documents and site documentation. These include consent forms, study papers, API documentation, and tutorials.
    - Return Type: Webpage
  - **showAccount():** Method used to display the user details and account functions for the current logged in user. Additionally handles app settings page where users can modify select app behavior.
    - Return Type: Webpage

- **Request/Response**
  As shown in Figure 4.2.1 the request/response object is representational of a standard HTTP request that contains Requests/Responses headers, parameters and body information about the resource requested by the Web Application as follows:

  Fields:
  - **IP:** IP address for the web server
  - **URL:** New URL navigating to the new webpage
  - **Type:** Type of request used (ex. Post, get, update, ect).

- **Params*:** Arguments specifying header and query fields.
- **Body:** data payload of the request/response. Could contain HTML, JSON, or XML data depending on the request.

- **StudyMetaData**
  A key-value data object that contains configuration and meta information about a given study. Provides an easy way to pass all data to element creation subfunctions when study data needs to be accessed.

  Fields:
  - **name:** String containing the study name.
  - **id:** unique numerical identifier used to identify study.
  - **start_date:** Date stored in datetime format when the study is starting. Value is set by the user creating the study.
  - **end_date:** date stored in datetime format when the study is ending. Value is set by the user creating the study.
  - **description:** String containing a brief description of the study by the researchers.
  - **published:** Date stored in datetime format indication when the study results will be published.
  - **height:** Boolean flag indicating whether the value will be needed from participants in the study.
  - **weight:** Boolean flag indicating whether the value will be needed from participants in the study.
  - **gender:** Boolean flag indicating whether the value will be needed from participants in the study.
  - **hand_dominance:** Boolean flag indicating whether the value will be needed from participants in the study

- **ExportParams**
  A key-value data object that contains information about a requested study participant activity export. Passed along to the server as part of the export request and used to query export status if needed.

  Fields:
  - **name:** String containing name for requested export, will default if null.
  - **id:** unique numerical identifier used to identify study.
  - **date_requested:** Date in datetime format when export was requested.
  - **start_date:** Date in datetime format of earliest date data to query for export.

- **end_date:** Date in datetime format of latest date data to query for export.
- **structure:** File and folder structure for the completed export.

- **Study**
  An object-oriented representation of a study including information about study setup and owners/users. Used to store and quickly retrieve data about studies when needed.

  Fields:
  - **Creator:** Id number of the user who created the study.
  - **meta_data:** StudyMetaData object with study metadata.
  - **assistants[]:** List of Profiles objects, one for each study assistant.
  - **participants[]:** List of Participant objects, one for each study participant.
  - **previous_exports[]:** List of existing previous export filenames.
  - **enrollmentLink:** Url used to enroll in study.
  - **study_log:** File data from study history log file. Contains attributed records of all user actions such as participant uploads, study modification, and exports requested.

  Methods:
  - **updateMetaData(Profile, token, StudyMetaData):** Method to change current study metadata object/information.
    - Return Type: Boolean
  - **requestExport(Profile, token, ExportParams):** Method to create export request for client to send to server or for server to pass to support program.
    - Return Type: ExportParams
  - **getPreviousExports(Profile, token):** Method to return list of available previous exports.
    - Return Type: Export[]
  - **uploadParticipants(Profile, token, Participant[]):** Method to add participants to current study.
    - Return Type: Boolean
  - **removeParticipants(Profile, token, Participant[]):** Method to remove participants from current study.
    - Return Type: Boolean
  - **addAssistant(Profile, token, Profile):** Method to add new assistant/researcher to study.
    - Return Type: Boolean

- **removeAssistant(Profile, token, Profile):** Method to remove assistant/researcher from study.
    - Return Type: Boolean
- **deleteStudy(Profile, token):** Remove all access permissions from all users to study.
    - Return Type: Boolean

- **getEnrollmentLink(Profile, token):** Method to get all active enrollment links for study.
    - Return Type: StudyEnrollLink[]

- **Profile**
  A key-value data object that contains the current user profile information. Used to easily identify the user and as an argument for requests.

  Fields:
  - **Local_id**: A string containing the id used to locally identify the user in our database.
  - **Email**: A string containing the email address provided by the user or through OAuth profile request for contact.
  - **Name:** A string containing the name provided by the user or through an OAuth profile request.

- **Session**
  A key-value data object that contains current connection session information for the current user including the authentication and refresh tokens and expiration information. Must be included in requests to the request server for any resource.

  Fields:
  - **user_id:** String containing user's local id.
  - **user_ip:** IP address used by user for current connection.
  - **token:** String containing authentication/security token for current session.
  - **refresh:** String containing refresh authentication/security token for next session.
  - **ttl:** Number of milliseconds before current session expires and refresh/new authentication token is needed.

- **User**
  An object-oriented representation of a user account, including information about accessible study and account personal information and settings.

Fields:

- **about:** Profile object containing information about current user.
- **connection:** Session connection information about current connection.
- **studies[]:** List of studies the user has access to represented by study objects.
- **notifications[]:** List of user notifications in the form of notice objects.

Methods:

- **getProfile():** Method to retrieve user profile information.
    - Return Type: Profile
- **getNotifications():** Method to retrieve user notifications.
    - Return Type: Notice[]
- **refreshNotifications(Notice):** Method used by application to check for and add new notifications to user account.
    - Return Type: Null
- **removeNotification(Notice):** Method ideally used to remove old or unneeded notifications from the user's account.
    - Return Type: Null
- **addStudy(Study):** Method to create new study and add it to the user's account.
    - Return Type: Boolean
- **getStudies(int):** Method to return all study objects associated with a user using the current user's unique id.
    - Return Type: Study[]
- **deleteStudy(Study):** Method to delete study from user's study list.
    - Return Type: Boolean

4.2.3 Public Interface Description

The public interface of the web app is the Request/Response class, which will communicate with the backend to both send and receive data to and from the database and associated support programs.

4.3 Backend Web, API, and OAuth2.0 Server

4.3.1 Component Description

This section will cover the request server and the interactions with the frontend as well as the backend support programs. The backend of our project contains the components that send all requests/response data back and forth between the frontend and any systems it needs to interact with (such as the support programs or the

database). This includes authentication of the user accounts as well as setting user accounts to the appropriate profile information, as well as accessing our API to get information from our database.

4.3.2 UML Class Diagram and Descriptions

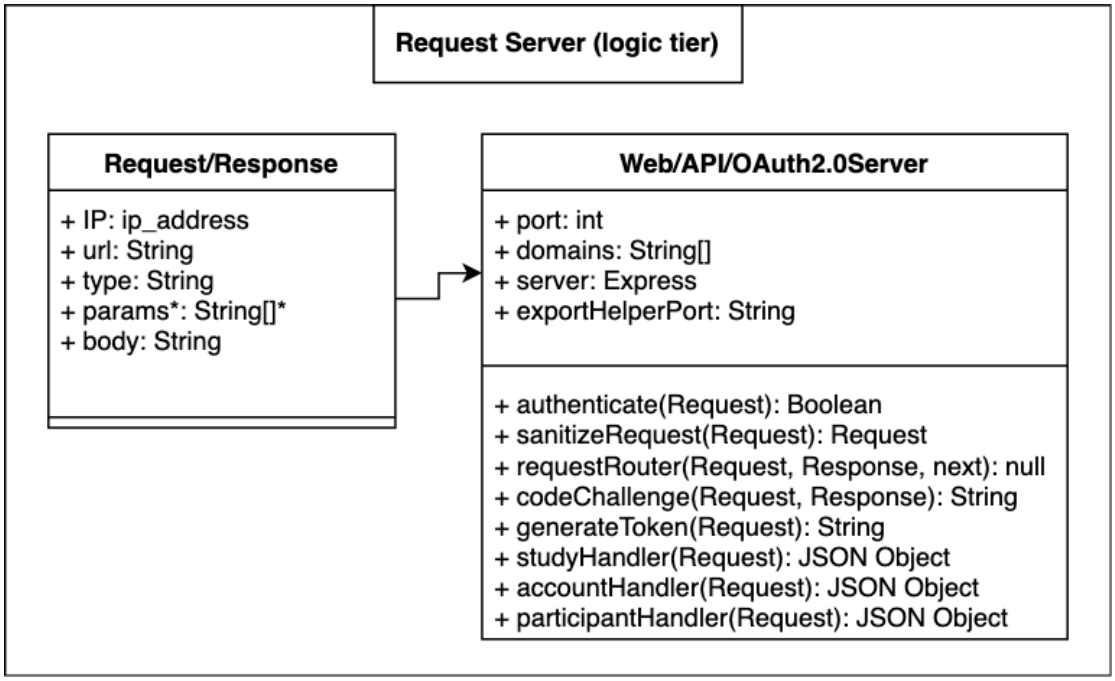

Figure 4.3.1: A diagram showing the Web/API/OAuth2.0 Server component's class and how it connects to the Request/Response class.

- **Web/API/OAuth2.0 Server**
  A NodeJS server which will be used to handle all of our webpage, API, and OAuth2.0 connections by receiving and sending back data via Request/Response.

  Fields:
  - **Port**: Port number needed for the web server for requests.
  - **Domains**: Contains a list of valid hostnames and domains for the server to accept requests for.
  - **Server**: ExpressJS server object containing the objects and requests needed for connections.
  - **exportHelperPort**: Contains port information for the Export Support Program.

  Methods:

- **authenticate(Request):** Using OAuth2.0 to authenticate the user and return the User object.
    - Return Type: User
- **sanitizeRequest(Request):** Parses all incoming requests to ensure they fit required structure and do not contain malicious or harmful data/arguments. After parsing it will return the clean request.
    - Return Type: Request
- **requestRouter(Request, Response, next):** Takes in a request or response and then routes it to its destination depending on either the url and params within the request or the next parameter if a response is provided.
    - Return Type: Null
- **codeChallenge(Request, Response):** Takes in a request or response for authorization and generates a code verifier and a code challenge, which then retrieves an authorization code which can be used to connect to the Fitbit API.
    - Return Type: String
- **generateToken(Request):** Takes in a request containing an authorization code and returns a token which can then be used for user authentication.
    - Return Type: String
- **studyHandler(Request):** Handles any study related requests and forwards requests/messages to appropriate private or support functions/programs if needed. If the request is intended for database alteration or access, it will then return a JSON object with the requested data; if not, then it will simply return a JSON object that indicates function success.
    - Return Type: JSON Object
- **accountHandler(Request):** Handles the account related requests and forwards requests/messages to the appropriate functions. If the request is intended for database alteration or access, it will then return a JSON object with the requested data; if not, then it will simply return a JSON object that indicates function success.
    - Return Type: JSON Object
- **participantHandler(Request):** Handles the participant related requests and forwards requests/messages to the appropriate functions. If the request is intended for database alteration or access, it will then return a JSON object with the requested data; if not, then it will simply return a JSON object that indicates function success.
    - Return Type: JSON Object

4.3.3 Public Interface Description

The public interface of the backend is the Request/Response class as defined in Section 4.2.2, which will be used to communicate with the frontend and any support programs. All other functionality within the backend cannot be accessed directly without using a Request/Response.

4.4 Backend Support Programs

4.4.1 Component Description

This section contains the Fitbit Support Program and Export Support Program modules, as well as how they connect to the database. These support programs will work alongside the backend to export or import data from the database or from Fitbit respectively. Exported data will be sent by the support program to the backend to then be sent to the frontend, while imported data will be sent by the Fitbit API to our support program and then added into the database.

4.2.2 UML Class Diagram and Descriptions

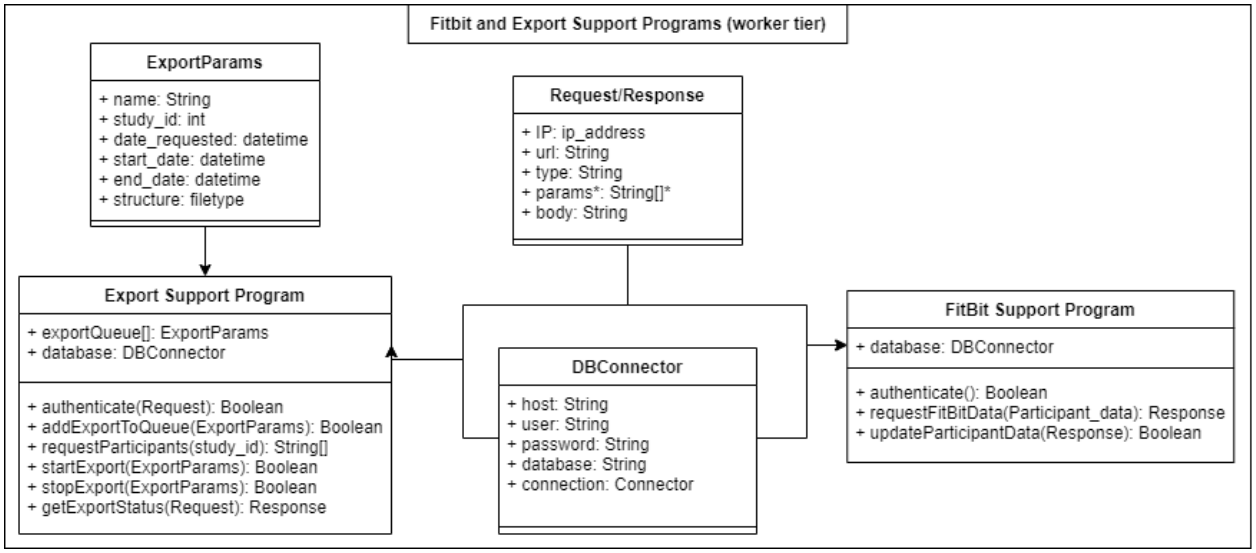

Figure 4.4.1: A diagram showing our backend support programs and their utilized classes.

- **Fitbit Support Program**
  The Fitbit Support Program will interact with Fitbit's API to request participant data and insert received data into the database.

  Fields:

- **database:** Database Connector object instance that allows for connection to database and command execution.

Methods:

- **authenticate():** Requests new auth token set using refresh token provided by original Fitbit code challenge and authorization exchange.
  - Return Type: String containing new token
- **requestFitbitData(Participant_data):** Requests Fitbit data for a given participant and adds it to the database. This will run regularly to ensure data is up to date and accurate, but can also collect and backfill any missed data.
  - Return Type: Response containing Fitbit activity or error code
- **updateParticipantData(Response):** Adds the participant's Fitbit data to the database and returns confirmation that data was added correctly.
  - Return Type: Boolean

- **Export Support Program**
  The Export Support Program will handle all export requests as a background task notifying the web or API when an export is complete.

  Fields:
  - **exportQueue[]:** Queue structure containing all incomplete export requests from client/users.
  - **database:** Database Connector object instance that allows for connection to database and command execution.

  Methods:
  - **authenticate(Request):** Checks token provided by in the request against the token saved for the user and returns a boolean flag with the result.
    - Return Type: Boolean
  - **addExportToQueue(ExportParams):** Adds requested export to export queue. Returns a boolean flag representing success of the operation.
    - Return Type: Boolean
  - **requestParticipants(study_id):** Requests all participants for the studies in the export queue and calculates export resource requirements,
    - Return Type: Participant[]
  - **startExport(ExportParams):** Pulls next study from the queue and begins the export process creating a new worker and log file for each export. Returns boolean success flag.
    - Return Type: Boolean

- **stopExport(ExportParams):** Pauses or cancels export to reclaim system resources, prevent further errors, or comply with user requests.
    - Return Type: Boolean
- **getExportStatus(Request):** Creates response with completion status of the requested export. Statuses include queued, working, and completed. For queued status the response should include a count of exports ahead of the requested export in the queue, and for working a rough estimate of the time needed to complete the export.
    - Return Type: Response

- **DBConnector**
  This class contains all of the details needed to connect the support programs to the database. It has no functions of its own, but rather allows the support programs to connect to the database and run their own functions using the connection it provides.

  Fields:
  - **host:** The host used for database connection.
  - **user:** The user that will connect to the database.
  - **password:** The password used to connect to the database.
  - **database:** The database which the support programs wish to connect to.
  - **connection:** A connector used to connect to the database.

4.4.3 Public Interface Description

The public interface of the support programs will be the Request/Response class, which will be used by the backend to send requests and receive responses from the support programs. All other functions and classes will only be accessible internally and will function based on the contents of the request they recieve.

5. Implementation Plan

We have already introduced the different parts of this project, so now we will move on to showing you how we plan to implement all of them. Figure 5.1 shows a Gantt chart that we will be using to illustrate our current timeline for this project.

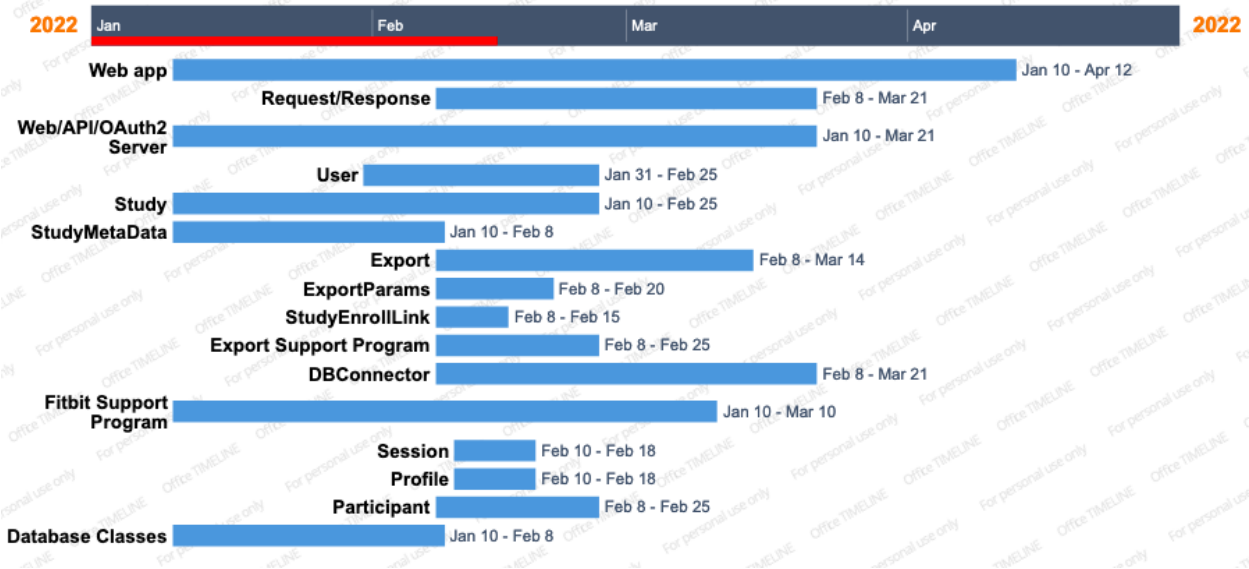

Figure 5.1: A Gantt chart showing our current implementation schedule for each class.

At the time of writing, we are working on the Web app, Study, Web/API/Oauth2 Server, StudyMetaData, User, and API classes, while also researching how to create a Fitbit support program and connect it to the rest of our project. Essentially, these modules make up the study creation portion of our project while also covering things like our login system. Because the Web app and Web/API/Oauth2 Server will be continuously worked on throughout the entire project as we add the rest of the modules, they have been given the longest time slot and thus will be worked on for the majority of the project. The Fitbit Support Program has also been given a very large timeslot due to both its importance and its complexity, as it is something that we will be building this entire project around and thus need to get working properly. We will have the StudyMetaData module and all database classes finished by February 8th and ready for use, and then will work on the main meat of the project after that date.

Starting on February 8th, we will then build on these modules to create our data export and study enrollment parts, as shown by the Export, ExportParams, Export Support Program, Participant, and StudyEnrollLink modules on our chart. By February 8th our Study module will be completed, with changes only needing to be made to make sure everything works together. We will also be expanding on our User component in that time to add the Session and Profile modules so that they can connect together to make a complete user experience. When these modules are completed, we should

have a fully functioning project done by March 21st. Afterwards, our only work on the project will be working on the web app to make sure that its user interface is fully functional and easily navigable according to the guidelines given to us by our client, finishing on April 12th.

6. Conclusion

Collecting data for health research is a very valuable and important field, and the use of Fitbits could help to revolutionize it and bring new insights that could help save lives to researchers who may not normally have the means to access expensive health monitoring equipment. Our clients Dr. Kyle Winfree and Dr. Eck Doerry aim to make the use of Fitbit data easier and more accessible for researchers, which could potentially allow for better studies than are currently available thanks to their low cost when compared to the expensive equipment that is usually used for heart disease research. This low cost could allow researchers to get more data for long periods of time and enroll more participants in their studies than they could normally, which could help save the lives of people affected by heart disease. Therefore, our project aims to build a secure and reactive web app that will allow researchers to use Fitbits to collect health data from study participants and then export that data for use in their research. A few highlights of this web app include the following:

- Full Fitbit integration, collecting all the data that Fitbit provides via their API.
- Systems for both manual study enrollment and participant self-enrollment, so it will be easy for researchers to add study participants however they please.
- Reactive and responsive functionality that will allow for data exporting that runs in the background, letting researchers continue to use the web app while they wait for it to retrieve their data.

We have moved past the requirements acquisition phase and are now working on building this web app. Currently we have finished our foundation and are moving towards the meat of the project: our user profile system, study enrollment system, and data export systems. In this document we have defined the three main parts of our project: the frontend (which users will interact with), the database (which will store all of our data), and the backend (which will connect everything together). We also described how these main parts all fit together and how they all connect. Additionally, we have defined the main five components that will be inside of these three parts, which are the following:

- The web app, which will serve as our frontend and user interface.
- The Web/API/OAuth2 server, which will handle and route all web requests while returning requested data as needed.
- The database, which will store all of our data.
- The Fitbit support program, which will collect data from Fitbit and put it on our database.
- The export support program, which will export study data from our database.

After defining these components, we have also defined the different classes within those components. Furthermore, we defined our current plans for implementation, including listing out what we expect each module to do and what kind of data they will

use or store. Finally, we have laid out a timeline of when we expect each module to be complete and when we will be working on each one. With all of these parts together, we have provided a clear and easy to follow description of what exactly we plan to implement, how we plan to implement it, and our schedule for implementation. With this information, we can be confident in our current progress while also looking to the future and knowing that we will be able to implement everything we have planned.