Stress Health Mobile Research Platform

Group 6:
Adam Levy
Brian Boudreaux
James Mirvil
Jesse Roberson

April 28, 2016

Sponsors:
Dr. Andrew Raij
UCF Institute for Simulation and Training

Dr. Julie Hinkle
UCF College of Nursing
# Table of Contents

Executive Summary ................................................................. 1

1. Introduction ............................................................................... 2

1.1 Project Motivation ................................................................. 2

1.2 Personal Motivation ............................................................... 2

  1.2.1 Adam Levy ................................................................. 2

  1.2.2 Brian Boudreaux .......................................................... 3

  1.2.3 James Mirvil ................................................................. 3

  1.2.4 Jesse Roberson ............................................................. 4

1.3 Broader Impacts ........................................................................ 5

1.4 Goals, Objectives, and Challenges 1.4.1 Goals ....................... 5

  1.4.2 Objectives ........................................................................ 6

  1.4.3 Challenges ....................................................................... 6

1.5 Project Requirements ............................................................. 7

  1.5.1 Web Server ..................................................................... 7

  1.5.2 Database ......................................................................... 8

  1.5.3 Web Dashboard .............................................................. 8

  1.5.4 iOS .................................................................................. 8

1.6 Tools ....................................................................................... 9

  1.6.1 Team Organization .......................................................... 9

  1.6.2 IDEs ............................................................................. 13

2. Research .................................................................................. 15

  2.1 Fitbit ................................................................................... 15

  2.2 iOS ........................................................................................ 17

  2.3 Machine Learning and Dlib C++ ........................................... 17

3. High Level Design ..................................................................... 18

  3.1 General Design Overview .................................................... 18

  3.2 High Level Architecture ....................................................... 19

    3.2.1 Component: Server ..................................................... 21

    3.2.2 Component: Mobile App ............................................. 23

    3.2.3 Component: Dashboard ............................................... 27
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2.4 Component: Database</td>
<td>30</td>
</tr>
<tr>
<td>4. Design Details</td>
<td>31</td>
</tr>
<tr>
<td>4.1 Web Dashboard</td>
<td>31</td>
</tr>
<tr>
<td>4.1.1 Technologies</td>
<td>31</td>
</tr>
<tr>
<td>4.1.2 Web Pages</td>
<td>33</td>
</tr>
<tr>
<td>4.2 Mobile App</td>
<td>37</td>
</tr>
<tr>
<td>4.2.1 Technologies</td>
<td>37</td>
</tr>
<tr>
<td>4.2.2 Mobile Application Screens</td>
<td>37</td>
</tr>
<tr>
<td>4.3 Server</td>
<td>55</td>
</tr>
<tr>
<td>4.3.1 Technologies</td>
<td>55</td>
</tr>
<tr>
<td>4.3.2 Implementation</td>
<td>59</td>
</tr>
<tr>
<td>4.3.3 Possible Issues and Solutions</td>
<td>60</td>
</tr>
<tr>
<td>4.3.4 API Documentation</td>
<td>61</td>
</tr>
<tr>
<td>4.4 Database</td>
<td>67</td>
</tr>
<tr>
<td>4.4.1 Technologies</td>
<td>67</td>
</tr>
<tr>
<td>4.4.2 Implementation</td>
<td>70</td>
</tr>
<tr>
<td>4.5 Machine Learning Classifier</td>
<td>73</td>
</tr>
<tr>
<td>5. Testing</td>
<td>85</td>
</tr>
<tr>
<td>5.1 Test Plan</td>
<td>85</td>
</tr>
<tr>
<td>5.1.1 Description of Test Environment</td>
<td>85</td>
</tr>
<tr>
<td>5.1.2 Stopping Criteria</td>
<td>85</td>
</tr>
<tr>
<td>5.2 Description of Individual Test Cases</td>
<td>86</td>
</tr>
<tr>
<td>6. Administrative Content</td>
<td>90</td>
</tr>
<tr>
<td>6.1 Budget and Financing</td>
<td>90</td>
</tr>
<tr>
<td>6.1.1 Developmental Budget</td>
<td>90</td>
</tr>
<tr>
<td>6.2 Milestones</td>
<td>91</td>
</tr>
<tr>
<td>7. Project Summary</td>
<td>92</td>
</tr>
<tr>
<td>8. References</td>
<td>93</td>
</tr>
</tbody>
</table>
Table of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1</td>
<td>Group 6 Github Commit Log</td>
<td>10</td>
</tr>
<tr>
<td>Figure 2</td>
<td>Group 6 Github Issue Tracker</td>
<td>10</td>
</tr>
<tr>
<td>Figure 3</td>
<td>Group 6 Slack Channel</td>
<td>12</td>
</tr>
<tr>
<td>Figure 4</td>
<td>Example of Github Slack Integration</td>
<td>13</td>
</tr>
<tr>
<td>Figure 5</td>
<td>XCode Playground Example</td>
<td>14</td>
</tr>
<tr>
<td>Figure 6</td>
<td>Wearable Market Share 2015Q4</td>
<td>15</td>
</tr>
<tr>
<td>Figure 7</td>
<td>Global Wearable Market Share 2015</td>
<td>16</td>
</tr>
<tr>
<td>Figure 8</td>
<td>Use Case Diagram</td>
<td>20</td>
</tr>
<tr>
<td>Figure 9</td>
<td>High Level Design Diagram</td>
<td>21</td>
</tr>
<tr>
<td>Figure 10</td>
<td>Flow and Connectivity of Internal Modules</td>
<td>25</td>
</tr>
<tr>
<td>Figure 11</td>
<td>Sequence Diagram</td>
<td>26</td>
</tr>
<tr>
<td>Figure 12</td>
<td>Web Dashboard Sequence Diagram</td>
<td>27</td>
</tr>
<tr>
<td>Figure 13</td>
<td>Web Dashboard Workflow Diagram</td>
<td>28</td>
</tr>
<tr>
<td>Figure 14</td>
<td>Activity Diagram</td>
<td>29</td>
</tr>
<tr>
<td>Figure 15</td>
<td>Database ER Diagram</td>
<td>30</td>
</tr>
<tr>
<td>Figure 16</td>
<td>Web Dashboard Login Screen</td>
<td>33</td>
</tr>
<tr>
<td>Figure 17</td>
<td>Web Dashboard Registration Page</td>
<td>34</td>
</tr>
<tr>
<td>Figure 18</td>
<td>Web Dashboard Registration Confirmation Page</td>
<td>35</td>
</tr>
<tr>
<td>Figure 19</td>
<td>Web Dashboard Main Page</td>
<td>36</td>
</tr>
<tr>
<td>Figure 20</td>
<td>iOS Login Screen</td>
<td>38</td>
</tr>
<tr>
<td>Figure 21</td>
<td>iOS Account Creation Screen</td>
<td>40</td>
</tr>
<tr>
<td>Figure 22</td>
<td>iOS Missing Mandatory Field Error</td>
<td>41</td>
</tr>
<tr>
<td>Figure 23</td>
<td>iOS Device Not Supported Error Screen</td>
<td>42</td>
</tr>
<tr>
<td>Figure 24</td>
<td>Fitbit OAuth2 Redirect Page</td>
<td>43</td>
</tr>
<tr>
<td>Figure 25</td>
<td>iOS User Home View</td>
<td>45</td>
</tr>
</tbody>
</table>
Executive Summary

Our project for the 2016 iteration of UCF’s senior design is a Stress Health Research Platform, the idea for which came from a joint effort between UCF’s College of Nursing and Institute for Simulation and Training. Our sponsors, Dr. Andrew Raij and Dr. Julie Hinkle, requested that we create a platform for researching stress health by finding a way to accurately predict if a user of the application is experiencing stress.

The guiding idea behind this project is simple: with smart wearable devices seeing a massive adoption rate (more on this later), the idea of minute-to-minute updates about your well-being has taken hold. Gone are the days of annual check-ups with your primary physician being the only window into your health; now, with easy to use apps and devices, you can leverage the power of data to track your progress.

While all of the above is true about physical health, mental health has not been paid the same attention. Our platform seeks to fill this void by utilizing smart wearables and mobile devices to monitor and help manage stress health. Monitoring is handled though capturing physiological, environmental, and contextual data and applying Machine Learning to this data to determine if a user is stressed. Management is handled by delivering stress health “interventions” via the user’s mobile device. These interventions are designed to be a simple notification that informs the user of the detected stress along with an instruction for the user to perform some action, such as “take some deep breaths” or other calming action.

Our project will consist of a Node.js server communicating with a MongoDB database. This web server component will handle storing and displaying the participant data. We also have an iOS component, where we will be querying data from Fitbit’s servers and forecast.io, rolling this data up into a set, and running it through our Machine Learning module. Our Machine Learning module then runs the dataset through a classifier, where the potential classes for the data are ‘stressed’ or ‘unstressed’. If the application detects the user is stressed, it will intelligently deliver a stress health intervention to the user. We will also be logging the classified data sets in order to investigate the efficacy of these stress health interventions and the overall stress health the participants.
1. Introduction

1.1 Project Motivation

The motivation for this project is simple: stress health is an important facet of your overall health that is frequently ignored, and our platform aims to remedy that. By keeping people informed about their stress levels, and by delivering timely interventions to help manage them, we believe that we can make a positive difference in the lives of many.

1.2 Personal Motivation

1.2.1 Adam Levy

Initially, I was attracted to this project because of the opportunity to develop a research tool which will be used to improve the quality of life for those who deal with stress. We tend to label someone as “healthy” if they have no physical ailments, but we do not take into account the person’s mental health. Although it is not something easily diagnosed, stress health can degrade the quality of life for those suffering with it.

For me, I know that I manage my stress inadequately but I also am hesitant to actively seek out help. This project excites me because it is going to give me an opportunity to help those who deal with the same sort of issues I do. It will give me the opportunity to help those who may not feel comfortable, or even see the need, to seek out professional help for stress. Our platform will give these people an opportunity to better manage their stress discreetly.
1.2.2 Brian Boudreaux

I know that I have always had problems managing my own stress. Either mismanaging my time in school, or just forgetting about important things until the final minute, I’ve had a lot of stress in my life. So I was interested in getting an opportunity to work on a project that could help more people manage their stress. More specifically, my interest in this project lies in the large scale aspect of the data that will need to be collected by the application. I study AI software and techniques, both in the classes that I take in school, and on my own time. In order for this project to make inferences correctly based upon the body of data collected, we are going to need to use some of these AI techniques. This project will be a great opportunity for getting hands on experience with data mining and Machine Learning.

1.2.3 James Mirvil

Working full time while going to school requires an enormous amount of time and dedication. The amount of stress involved in ensuring the quality of time distributed between the two is not diminished is even greater. This balancing act is what I deal with every day. When learning of a project that’s sole purpose was to investigate ways to mitigate stress levels, I was immediately intrigued. The concept of creating a platform that will allows individuals the ability to better their health was inspiring. Furthermore, seeing that the interesting in wearable devices is in the forefront of the masses, gives hope that a meaningful application can result from this project. This is something that I feel can have a great impact on the lives of individuals such as myself.

The prospect of improving the quality of life is not the only thing that interest me. Working with big data is something I have not directly dealt with. As we progress through the information age, data has become the foundation of
advancement. Understanding how to manipulate such data is essential. Thus, exposing myself to a platform which, at its core is big data, will help develop the skills that I feel are necessary to be a functioning member of the technical world. Moreover, knowing that I will be interfacing with different technologies including Machine Learning, iOS development, website creation, as well as server and database management is something that reinforces the notion that I have selected the correct project.

1.2.4 Jesse Roberson

My motivation for this project is to see people treat their mental health with the same care as they do for their physical health. Stress is a huge part of mental health, and as a society we tend to let stress permeate all aspects of our lives without care, because we regard it as a normal state of mind. In actuality, stress impacts your health in a multitude of ways, and many of them can have far reaching consequences.

On a personal note, between the time of my assignment to the project and the writing of this document, my mother has suffered a mini-stroke brought on by work-related stress. This mini-stroke has caused a noticeable decline in her ability to recall information as well as general cognitive capabilities. While her situation is relatively tame compared to other potential stress-related issues, it is still a sobering way to be informed of the importance of stress health. It is my sincere hope that this project can bring about positive change in the lives of people with poor stress health, and potentially mitigate the consequences of unchecked stress.

In addition to the impact on mental health, I am also excited to be working on the project for the technical challenges. The application we are developing will be utilizing Machine Learning, which is something that has been on my list of "cool and exciting things to learn" since I started here at UCF. In addition, this project will require refinement of my existing web development knowledge, as well as development of other skill sets, such as mobile development and device integration.

In summary, I believe that this project is a perfect fit for me, both in terms of societal impact and technical challenge, and I am thrilled to be working on it.
1.3 Broader Impacts

The American Institute of Stress in New York claims that 77% of Americans report regularly experiencing physical symptoms brought on by stress, and 73% of Americans report experiencing psychological symptoms. Despite these statistics, many people still do not regard stress as something to manage. This project’s main goal is to help people become more aware of their stress and help manage it. This project also aims to assist those who work in high-stress occupations, such as police officers, with managing their stress. Through constant monitoring, we aim to gain greater insight into stress health and learn how to effectively intervene when necessary, thereby improving the quality of life for everyone that experiences stress.

1.4 Goals, Objectives, and Challenges

1.4.1 Goals

- Develop a web server which will authorize Fitbit users and openly communicate with Fitbit servers, collecting user data
  - Web server will use Fitbit Oauth2 to authorize user
  - Web server will passively collect data from Fitbit, queuing queries that have not been completed to run at a later time
  - Web server will have endpoints that allow our classifier to access what data it needs
- Develop an iOS application that will monitor physiological, environmental, and contextual data from the user
  - iOS application will continuously collect data from the user while being non-intrusive
- Create a secure web interface for health professionals to analyze both individual and collective stress health data
  - Web interface will give the health professionals access to all data collected
  - Health professionals will be able to query data based on their interests and study needs
- Investigate and understand the relations between physiological, environmental, and contextual data
  - Build a Machine Learning classifier which will interpret the data points given
- Deliver stress health “interventions” when our classifier detects user is experiencing stress based on gathered data
  - The classifier will learn from its mistakes as user's confirm if the classifier “intervention” was correct
1.4.2 Objectives

- Leverage Fitbit API to collect user's physiological data
- Utilize The Weather Channel API to collect environmental data
- Create interactive questionnaire to periodically prompt user for qualitative contextual data
- Handle authentication for authorized users
- Secure endpoints restricting unauthorized users from accessing sensitive data
- Use Machine Learning techniques to investigate the relationships between the gathered data and stress health

1.4.3 Challenges

Due to the breadth and complexity of our project, we have several key challenges we will need to overcome if we are to complete it. Most of these challenges come directly from the Machine Learning component and the specifics of its implementation.

**Machine Learning**

Possibly the biggest challenge with this project will be the size and importance of the Machine Learning aspect. This project hinges on developing an accurate classifier that can correctly identify stress in our users, and given the limited experience that our group has with Machine Learning, this has proven to involve a lot of research and testing.

When we first consulted with our sponsors, they were quick to point out to us that different people respond to stress in different ways. Building a single classifier that would classify the data of all of our users would prove to be ineffective; such a classifier would be wildly inconsistent as it tries to learn dozens of individuals who are each reacting differently to their stress. This required our classifier to move to the iOS app, rather than stay on the server where we initially thought it would go. With the classifier on the app, each user will have their own classifier that will learn their unique reactions to stress.

Adapting a classifier to a new user is challenging because we have no reference for how that user responds to stress. A general purpose classifier will not suffice, so we have to develop a way to quickly learn a new user’s stress responses. This needs to query the user periodically for their stress levels over the course of a day, and use their responses to label the data sets as they are read in.

Moving the classifier to iOS does not come without tradeoffs, however. Being on a phone will limit the resources available like memory, and processor time. The biggest impact we will see is in usable memory. Our classifier will need
to hold on to a large number of data entries in order to train on them, so we will need smart memory usage in order to minimize the impact that the classifier has on the user’s phone.

We are relying on the user’s honesty to collect accurate data about their stress. This puts our classifier in a vulnerable position should the user either, lie about when they feel stressed, or underplay their level of stress. Without honest data, our classifier will not have accurately labeled data on which to train. However, this challenge is a bit out of our control since it relies on the user being invested enough in using the app to use it honestly. However, because this is a research platform, we feel that the users we will have will be willing to follow this more closely than an average user who downloaded the app from the AppStore.

1.5 Project Requirements

1.5.1 Web Server

**Manager: Adam Levy**

- Must make Fitbit API requests to collect user specific data (i.e. heart rate, sleeping patterns, steps taken)
- Must make forecast.io API requests to collect user specific environmental data
- Must have custom, secure, API built for:
  - Handling registration, authentication, and login for Fitbit users
  - Handling registration, authentication, login, and approval for health professionals
    - Current approval process will involve sending an email to Dr. Raij and Dr. Hinkle asking to approve the health professional requesting to register
  - Retrieving and updating user personal information
  - Storage and retrieval from the database
  - Handling user input
- Web server must handle issues that arise from attempting to access data that is not yet available from Fitbit:
  - Check if Fitbit returned valid data. If no valid data was returned, add that query to a queue.
  - Periodically attempt to run queries in our queue, if still no valid data put query to the back of the queue.
1.5.2. Database

Manager: Brian Boudreaux
- Must store user’s personal information as well as Fitbit authentication token
- Must store data points collected from Fitbit servers
- Must store weather data collected from Forecast.io
- Must be accessed solely by the Web Server
- Must host the following data:
  - Physiological data collected from Fitbit (150 HTTP requests per hour)
  - User contextual data
  - User environmental data
- Must store health professional’s personal information as well as their web session token

1.5.3. Web Dashboard

Manager: James Mirvil
- Must be able to communicate with the Web Server sending HTTP requests back and forth
  - Web dashboard should be able to send requests only if the health professional has a valid session
- Must give the health professionals an opportunity to register, login, and be approved by Dr. Raij and Dr. Hinkle
  - Dr. Raij and Dr. Hinkle will be able to approve or deny new health professionals simply using a check-box
  - If a health professional is approved, they are verified and their is_verified flag will be updated.
- Must be able to display data from the database in a readable manner
- Must have a query builder allowing health professionals to retrieve the data relevant to their research

1.5.4. iOS

Manager: Jesse Roberson
- Must be able to integrate with the Fitbit application for OAuth purposes
- Must be able to prompt user to give application access to location services
- Must be able to prompt user for contextual data on a timer
- Must be able to read from iPhone sensor data (e.g., accelerometer, GPS) to provide contextual information
- Machine Learning and data analysis
  - Must utilize Machine Learning techniques to analyze environmental, contextual, and physiological data and establish relations between this data and stress health
- Must be able to deliver stress health “interventions” when the application detects the user is under stress
- Must securely communicate with the web server and submit user responses to diagnostic questions

1.6 Tools

Due to the breadth of this project, our team is leveraging many different tools and technologies in the development of our application. Because our team is also using different operating systems, the technologies we selected must be platform-agnostic—code written by one member should be accessible by all other members, regardless of operating system or environment.

1.6.1 Team Organization
**Git and GitHub**

In order to better facilitate cooperative development, we have elected to use Git and GitHub for version control and source code storage, respectively. These technologies were selected due to our team’s familiarity with them, as well the fact that GitHub and Git are the de facto standards for their respective services. Both Git and GitHub function the same across platforms, making collaboration easy despite our team members using different operating systems for development.

GitHub provides a convenient interface for determining progress of development by keeping track of commits to the repository across multiple branches of development. It also enables our team to roll back erroneous changes, should something break when a team member updates the repository.

![GitHub Commit Log](image1)

*Figure 1 - Group 6 Github Commit Log*

In addition to the base functionality of code storage and visualization, GitHub also enables our team to keep track of issues in development. This issue tracker will allow us to easily keep everyone on the team informed about problems encountered during development, as well as serving as a convenient “to-do” list.

![GitHub Issue Tracker](image2)

*Figure 2 - Group 6 Github Issue Tracker*
Through correct utilization of all of the features Git and GitHub make available to us, we hope to keep our application development easy by mitigating the cost of making (inevitable) errors during development. For this reason, these two technologies might be the most important tools in our team’s arsenal.

Slack

Slack is a messaging application targeted at software developers, and has already seen extensive use by all of our team members. Slack provides a web application, standalone applications for Windows and OS X, and applications for iOS and Android. Because Slack has applications for all phones and operating systems our team uses, it is a clear winner for team communication. Slack’s ubiquity will enable our team to be in constant communication, regardless of our physical location.
In addition to its core messaging functionality, Slack also has plugins for a variety of features. Most notably, Slack can integrate with GitHub through one of these plugins. This plugin informs team members of changes to code repositories, new issues added to issue tracker, or pull requests. This functionality serves to keep everyone on the team informed about the status of the project at all times.
1.6.2 IDEs

Sublime Text

Sublime Text is a smart text editor with a slick user interface, fast performance, and a plethora of features that make development easier. Along with customizable themes and syntax highlighting. Sublime Text also provides the ability to traverse through files within the editor. Not only does Sublime Text provide the ability for users to set syntax highlighting for languages, but also allows users to specify which framework they are using (Node.js). Sublime Text provides fully customizability just by manipulating simple JSON files.

XCode

XCode is an IDE developed by Apple for iOS and OS X development. Because our application’s target mobile environment is iOS, XCode is essential for developing our application. Thankfully, XCode also happens to be an excellent tool for this purpose, providing developers with a good view of their project hierarchy, making it simple to set up dependencies and folders. XCode provided the ability to drag and drop UI components such as Buttons, Table View Controllers, Segues, etc. and easily connect them to your code. XCode’s playground feature even allows you code side-by-side with your UI, updating the UI live as you are coding.
Figure 5 - XCode Playground Example
2. Research

Due to the complex nature of our project, as well as the numerous options available to us, we as a group had several things to research in order to make educated decisions about our project's direction. We briefly summarize the research that we made about various facets of our project below, highlighting the reasons that led us to our ultimate decisions.

2.1 Fitbit

While there are a wide variety of wearable devices on the market that perform nearly identical functions, Fitbit blows away the competition in terms of market share. This market share, combined with a well documented API and reasonable price point, made the decision to go with Fitbit an easy one.
Another benefit of Fitbit is that its devices are platform-agnostic. Our project sponsors have expressed a desire to have the platform remain as open-ended as possible, allowing them to expand upon it after our completion of senior design. With Fitbit, our application could easily be expanded to other mobile operating systems (Windows Phone, Android, etc.), and our method of collecting data would remain unchanged.

Continuing the theme of extensibility, Fitbit’s API means that our platform can also support other Fitbit devices in the future. Currently, our application is being developed with the Fitbit Charge HR in mind, since it is one of only two models that captures heart rate data. In the future, if another model is released that captures this data; our application innately supports it due to Fitbit’s API. If a new Fitbit model is released that performs some new functions, modifying our application to access them could be done with minimal effort in the future. This is another way that our decision to go with Fitbit means our sponsor’s desire for open-endedness is met.
Fitbit has also been supportive of their development and research community, and our team is no exception. Early on, we discovered that we would need special permission from Fitbit in order to access data on a nearly continuous interval. After a short email exchange with Fitbit’s customer service team where we explained the purpose of the platform, they granted us access to the data we needed nearly immediately. This support, along with their excellent documentation, make developing for Fitbit smooth and nearly effortless.

2.2 iOS
The decision of which mobile platform to support was not an easy one, nor was it made quickly—indeed, we came dangerously close to changing our mobile platform decision just recently. That said, we ultimately came to a decision and elected to develop for iOS. This decision ultimately came down to two factors:

1. Apple’s Toolkit for iOS development (Xcode and Swift) is impressive and easy to pick up.
2. C++ libraries are natively supported, meaning we could use Dlib C++ for our Machine Learning library on the phone

These two factors tipped the scale enough in Apple’s favor, so our application is officially for iOS.

2.3 Machine Learning and Dlib C++
Machine Learning is the component of our project that we collectively have the least experience with and, as a result, had to spend the most amount of time researching. After weighing our options for how to implement a Machine Learning solution in our project, we eventually decided to use decision-tree learning in order to classify our data. We recently met with Dr. Kenneth Stanley here at UCF to get an expert opinion on the matter, and he advised us on many of the techniques we had available to us.

Our problem is one of classification; we have sets of data that need to be labeled as being in one of some number of classes. In our case, we are measuring a user’s vitals (heart rate, activity, sleep, their local weather, etc.) and have to classify each set of data as either stressed or unstressed. There are a number of different algorithms available for doing classification, but we needed a library that could be used natively in an iOS app.

Our research led us to Dlib C++, a comprehensive Machine Learning library written in C++. Dlib C++ supports a number of different algorithms for a variety of Machine Learning tasks and, since it is written in C++, we can include it as an Objective-C header in our Xcode project.
3. High Level Design

3.1 General Design Overview

The Fitbit device, also known as tracker, tracks three data points that is necessary for the stress recognition: heart rate, sleep, and activity level. This information is sent to the Fitbit server through a syncing device, for our purposes it will be a mobile phone, and is stored for later access. The key point about this process is that the Fitbit server and the sync device must be connected at the same time for the data to be transferred. Data cannot be retrieved directly from the device or the sync device from a third party; it must be retrieved from the Fitbit server with their API.

Our server will be used to fetch the user data from the Fitbit server. Before the server can perform these calls, it first must receive authorization from the user. Upon initial registration, the user will be prompted to authorize our system with the ability to access their data on the Fitbit server. Once the proper authorization has been granted, the server will access the profile data of the user and gather the device Id, user Id, and the device type. The server uses this information to validate that the tracker that is tied to the account is a supported model. If the user is approved, the authorization is sent to the app, and then the data is used to establish a new user entry into the data. If it is not, a message is sent to the app notifying the rejection.

Once a user is accepted, the server will periodically gather the data from the external Fitbit server and send it along to app for later processing. The frequency of calls is determined by the app. As long as an active internet connection is available, the app will request a new request a new set of data from the server every 10 minutes. In addition to this, the app requires environmental data. The app will need access to the location data of the user's phone. The application will then periodically request weather information from an external server.

The app will the physiological, environmental, and contextual data gathered from the various sources to feed the Machine Learning algorithm. The physiological data will be used as the basis for determining the level of stress. The environmental and contextual data will be used to support the decisions and rule out false positives. Through validation of the systems assumptions, the classifier
will be tailored to each individualize. This classification will be the output of the Machine Learning system. If the system detects that the user is currently experiencing a stress event, a notification will be sent to the user. The results will be sent to the server, which will be saved to the database.

The information held in the database will be accessible to the researchers through a web dashboard. Researchers will be able to login onto the portal and request data on specific participants as well as groups in a study. The request will be sent to the server to be processed. This will then be retrieved from the database. Once the returned from the server, the information will be graphed or reported in the manner it was requested.

3.2 High Level Architecture

To facilitate the needs of the system best, we will be using a client server model. As the server serves the center point in communication between subsystems, gathering, validating, and sending data, this model provides an adequate overview of what will be needed by the system. The major components of the system are the Server, App, Web Dashboard, and the Database. In addition to the four major component, there are three external components: Fitbit Server, Weather Server, and Fitbit Tracking Device. The figure below illustrates their connectivity.
Stress Research Platform

Figure 8 - Use Case Diagram
3.2.1 Component: Server

The server is a component that communicates to each of the other major components. It processes the data request from each client and performs the requested action when valid.

**Server Functionality**

- Create new user
  - Get user id, device id, and device name from the profile data from the Fitbit server
  - Check if the device of the is supported
    - If the device is not supported, the account is rejected
      - Failure notification sent to app
    - If the device is supported, the account is accepted
      - A new user entry is created in the database with Fitbit data
      - Success notification sent to the app
      - User Id, Device Id, and Device name sent to app
- Get Existing User Data
- Check if Fitbit and user already exist
  - If user does not exist, the login is refused
    - Refusal notification sent to app
  - If user does exist, the login is accepted
    - Success notification sent to the app
    - Last classification profile is retrieved from the database
      - Classification is sent to the app
- Get Fitbit Data
- Checks is new data is present sync last sync
  - If no new data, send no new data message to app
  - If new data exist, send sleep, heart rate, and fitness
  - If authorization is revoked or token is invalid, do not process request
    - Revoked authorization / Invalid token notification sent to the app
- Save Participant Data
  - Check if request is a duplicate request
    - If duplicate, do not store
    - If not duplicate, save profile to the user database
  - Send success or failure notification to app
- Query Database
  - Authenticate credential of the requesting researcher
    - If user is not authorized, send a unauthorized notification
    - If authorized, perform query
      - Send results of query back to requested researcher
- Update Survey Questions
  - Authenticate credential of the requesting researcher
    - If user is not authorized, send a unauthorized notification
    - If authorized, update list of available survey questions
3.2.2 Component: Mobile App

The mobile app is the component that interfaces with the participants of the research study. The app analyzes the collected data then classifies whether a stress event has occurred, and delivers a stress health intervention when appropriate.

Mobile App Functionality

- Device Access
  - Request permissions to access storage and location (Installation)
  - If access is revoked, notify the user to re-enable access
- Create account
  - Transition to account creation flow
  - Keep continue button greyed until participant agrees to Terms
  - On continue validate account details
  - Do not proceed, report error if Name, Occupation, or Date of Birth are not set
  - Transition the user to Fitbit Authorization (OAuth2)
  - Authorization is denied, return to previous screen
    - Indicate that authorization is required to use the app
  - Authorization approved but rejected by server, return to previous screen
    - Indicate account already exist, device not supported, unknown error (try again later)
  - Authorization approved, success reported from the server
    - Update settings with device Id, user Id, and Device name
    - Transition user to home screen
- Login with existing Account
  - Transition the user to Fitbit Authorization (OAuth2)
  - Authorization is denied, return to previous screen
    - Indicate that authorization is required to use the app
  - Authorization approved but rejected by server, return to previous screen
    - Indicate account no account exist, device not supported, unknown error (try again later)
  - Authorization approved, success reported from the server
    - Update setting and classifier with profile data from server
    - Transition user to home screen
- Store Stress event
  - Triggered from home page, automatically records date and time
  - Triggered from log screen, record user entered data
    - Ignore if there is already an event for this +/- 5 minutes
  - Result of a classification
    - Within 1-2 minute window, alert the user of the event
    - Outside of window, save data point for later
• Request Fitbit data from Server
  o Send request to the server every 10 minutes
    ▪ Send time of last successful request
  o Receive response from the server
    ▪ No new data, keep time of last new data received
    ▪ New data, send data points to classifier
• Request Weather data from Weather Server
  o Get the current location from the device
  o Send request to the weather server every 15 minutes
  o Receive response from the server
    ▪ No new data, keep time of last new data received
    ▪ New data, send data points to classifier
• Update Settings
  o Validate setting changes
  o Invalid Changes, do not update
    ▪ Report error if Name, Occupation, or Date of Birth are invalid
  o Valid Changes, update to the user input
    ▪ Save updates for all settings to internal memory
    ▪ Save updates for DOB and Occupation to Server
  o Request user profile change from the server
• Prompt contextual prompts
  o Request update from server for new context questions (weekly)
  o If new question, save new questions into memory
  o Prompt the user randomly throughout the day
Figure 10 - Flow and Connectivity of Internal Modules
Figure 11 – Sequence Diagram
3.2.3 Component: Dashboard

![Web Dashboard Sequence Diagram](image)

*Figure 12 - Web Dashboard Sequence Diagram*
Figure 13 - Web Dashboard Workflow Diagram
Figure 14 - Activity Diagram
3.2.4 Component: Database

The database is a component that will hold the records of both the participant and the researchers. The database can only be accessed directly through the server.

**ER Diagram**

![Database ER Diagram](image-url)

*Figure 15 - Database ER Diagram*
4. Design Details

4.1 Web Dashboard

4.1.1 Technologies

For the development of our web dashboard, our team is using the industry-standard Holy Trinity of web technologies: HTML, CSS, and JavaScript. We are using these technologies due to their ubiquity, and essentially, monopoly on web development. Alongside these three technologies, we will also be using AngularJS.

**HTML**

HTML (HyperText Markup Language) is the industry standard markup language for building web pages. HTML uses a set of common elements to describe individual components of a webpage, which a user’s web browser then reads and interprets in order to generate the visual web page. Currently in its fifth major version, HTML has a multitude of different components that can be used to describe, and thus build, nearly anything imaginable.

**CSS**

CSS (Cascading StyleSheets) is another markup language that is used to describe styles for HTML elements. An apt comparison for the relationship between CSS and HTML can be found with adjectives and nouns: CSS describes the attributes of HTML elements in much the same that adjectives describe nouns. CSS will be used extensively in order to make sure that our web dashboard is presented clearly on a variety of potential devices, through the proper use of responsive web design.
**JavaScript**

JavaScript is a scripting language that we are using to handle actions on our web dashboard. Alongside frameworks such as jQuery, JavaScript can listen for certain events triggered on web pages, such as a user clicking a button with their mouse, or pressing a key on a keyboard. We will be using JavaScript to handle dynamic content loading, such as re-rendering our data table when the user only wants specific information.

**AngularJS**

AngularJS is an open source JavaScript framework that we are using due to its powerful two-way data binding functionality. Two-way data binding refers to the concept that any changes to an underlying model should be appropriately reflected wherever that model is referenced, and vice-versa. This, in conjunction with AngularJS’s capability to make HTTP requests will enable us to dynamically update our web dashboard with data from Fitbit, while also storing said data in our database.
4.1.2 Web Pages

**Login Page**

The login page will take an email address and password as credentials, and query the health professional database in order to verify that the user has access to the portal. If the user has entered valid credentials, they will be logged in and redirected to the main page; if the credentials entered are invalid, the user will be notified and will not be redirected.

*Figure 16 - Web Dashboard Login Screen*
Registration Page

In order to access the database and participant information, the user must register and be approved by administrators of the site. Currently, registering as a health professional only requires first and last names, an email address, and a password. This is, of course, subject to change, and our form will change accordingly. Once the user enters their information, the server will check the health professional database to verify that the user has entered a unique email address. If the email entered is unique, the server will create a new health professional entry, and the user will have access to the site once an administrator approves them. If the email entered is not unique, the page will request that the user enter a different, unique email address and the process is then restarted. Once the user successfully registers, the user will be directed to the “Registration Confirmation” page.

Figure 17 - Web Dashboard Registration Page
Registration Confirmation Page

Once the user has successfully registered as a health professional, they are redirected to the registration confirmation page. This page simply informs the user that they successfully registered and that an administrator needs to confirm their registration before they are granted access to the site. Once a user is registered and confirmed, they are then able to login and access the database.

Figure 18 - Web Dashboard Registration Confirmation Page
Main Page

Due to the simplicity of the web dashboard component, this page contains all of the functionality of our web interface. From here, the health professional is able to see a table of the participants, search for specific participants, access participant data, and log out.

The main table that the user is presented with upon accessing this page contains a list of all active participants of the study. This is a simple list containing basic information about the participants, according to the user database schema. From this table, specific user information, such as heart rate logs, can be accessed by clicking on the user’s entry in the table. This action will cause the table to re-render, containing the requested user information.

The search functionality for the main table will attempt to find matches in all columns, dynamically hiding rows that contain no matches. This serves to manage a potentially large dataset by enabling health professionals to reduce the size of the data table to make it more manageable.

![Stress Health Research Platform]

*Figure 19 - Web Dashboard Main Page*
4.2 Mobile App

4.2.1 Technologies

**Swift**

Our team elected to use Swift, Apple’s newer iOS development language, in the development of the mobile component of our application. This choice was made simply because we, as a team, wanted to gain experience with a new language and because Apple appears to favor Swift over Objective-C. Fortunately, this decision resulted in boosts to readability and performance, as Swift has dropped C-language conventions, and is nearly on par with C++ performance.

4.2.2 Mobile Application Screens

**Initial Interactive Screen**

The Initial Interactive Screen, also known as the login screen, will be the first screen that appears when the user first launches the app. This is the entry point for a participant into the app. After an account has been assigned to the app, this screen should no longer appear. This screen will appear anytime that the user has not logged on to the system with a valid account. Figure XX.X

**Interactive Interfaces**

- Register a New Account
  - On selection, the app will transition to the create new account flow
- Email Text Field
  - On selection, a keyboard prompt will appear
  - Supports text highlighting and copy / paste functionality
- Password Text Field
  - On selection, a keyboard prompt will appear
  - Hidden characters on entry
  - Supports text highlighting and copy / paste functionality
- Log In Button
  - On selection, the app will transition to home screen
  - Trigger error if either password or email is missing
  - Trigger error if email is an incorrect format
- More Information Button
  - On selection, app will transition to the “About Us” screen
**Supported Notifications / Error States**

- **Invalid Account Notification**
  - Appears when server rejects the login
  - Directs the user to create an account if none exist in database

- **Unknown Error Notification**
  - Appears when either server (Fitbit or ours) is unavailable
  - Directs the user to try again later

---

**Create Account Screen**

The Create Account Screen will appear after the user selects to register a new account on the login screen. This screen will provide a basic form for the user to fill out. This information will be used as seed data for the entry into the database. The screen can only be accessed through the new user flow.
**Interactive Interfaces**

- **Back Button**
  - On selection, the app will transition to back to the Login Screen
- **Name Text Field**
  - On selection, a keyboard prompt will appear
  - Supports text highlighting and copy / paste functionality
- **Occupation Dropdown**
  - On selection, a list of available occupations will appears
  - Selecting an occupation will close the dropdown and set the field to the selection
- **Date of Birth Date picker**
  - On selection, a date picker will appears
  - The date picker will support, Month-Day-Year selection
  - Selecting a date will close the picker and set the field to the selection
- **Notification Level Dropdown**
  - On selection, a list of notification levels will appears
    - Will include High, Medium, Low, None
  - Selecting a level will close the dropdown and set the field to the selection
- **Agree Checkbox and Link**
  - Selecting the link will bring the user to the Terms and conditions of the app
  - Checking the check box, enables the continue button
- **Continue Button**
  - Greyed out until the user agrees to the term of the app
  - On selection, the app will enter into the Fitbit authorization flow

**Supported Notifications / Error States**

- **Mandatory Field Notification**
  - Indicates that Name, Occupation, or Date of Birth are not set
- **Authorization Failure Notification**
  - Appears when user does not authorize app
- **Device Not Supported**
  - Indicates which devices are supported by the app
- **Accounts already exist**
  - Directs the user to use the login functionality
- **Unknown Error Notification**
  - Appears when either server (Fitbit or ours) is unavailable
Figure 21 – iOS Account Creation Screen
Figure 22 - iOS Missing Mandatory Field Error
Figure 23 - iOS Device Not Supported Error Screen

Device Not Supported

It looks like the device linked to your account is not supported. Currently we only support the Fitbit devices with continuous heart rate monitoring capability. This includes:

- Fitbit Charge HR
- Fitbit Surge
- Fitbit Blaze

If you have an account with any of these devices attached, please try and register again.

Dismiss  Exit App
**Fitbit Authorization Screen**

The Fitbit Authorization Screen is that appears when selecting to create a new account. This screen is managed by third party and opens from within the app. The page will request that the user log in to their Fitbit account. Once logged in, the user will be prompted to Allow or Deny the app account to their data. We require both read and write access.

*Figure 24 - Fitbit OAuth2 Redirect Page*
Home Screen
The Home Screen will appear after a successful registration or login. This will be the default screen that appears when launching the app after an account has been assigned to the app. The user uses this screen to quickly log a stress event manually.

Interactive Interfaces
- Stress Button
  - On selection, the app will log a stress event
  - Button will have an animation to indicate something happened
- Stress Statistics Module
  - Indicates the last stress event, number of stress event that current day, and the daily average number of stress events
  - Updates when the user accesses the screen and when a new stress event is detected
- Stress Log Tab
  - On selection, the app will transition to the Stress Log screen
- Settings Tab
  - On selection, the app will transition to the Settings screen

Supported Notifications / Error States
- Stress Event Detected
  - Notifies the user that they are currently stressed
  - Suggest ways to mitigate and resolve event
  - Option to indicate whether the suggestion helped
- Contextual Request Notification
  - Appears randomly throughout the day
  - On selection, the user will be transition to the screen
Figure 25 - iOS User Home View
Figure 26 - iOS User Stress Event Detection
Stress Log Screen

The Log Stress Screen is a screen in which the user can select to log stress events for any time of the current day. This allows the user to enter any events that have not yet been recorded by the system. It can be accessed from either the Home screen or the Settings screen. The user can allow view the last stress events from the last 24 hours.

Interactive Interfaces

- Log Stress Event Time Picker
  - Allows for scrolling by Hour and Minutes
  - Uses 12 hour clock, with the ability to set AM and PM
- Reset Button
  - Discards the current picker setting then set the picker to 12 AM
- Save Button
  - Saves the current picker setting as a stress event
  - Animation will appear to confirm entry of event
- Previous Stress Event List
  - Lists all stress events for the last 24 hours
  - Current day event is listed with a Today tag
  - Previous day event is listed with a Yesterday tag
  - Updates when the logs another stress event from the picker and when new stress event is detected from the classifier
- Home Tab
  - On selection, the app will transition to the Stress Log screen
- Settings Tab
  - On selection, the app will transition to the Settings screen

Supported Notifications / Error States

- Stress Event Already Exist
  - Notifies the user that there is a previous event already exist for the time selected
  - Triggers when event is +/- 5 minutes from set date
- Stress Event Detected
  - Notifies the user that they are currently stressed
  - Suggest ways to mitigate and resolve event
  - Option to indicate whether the suggestion helped
- Contextual Request Notification
  - Appears randomly through the day
  - On selection, the user will be transition to the screen
Figure 27 - Stress Health Log Event Screen
Settings Screen

The Settings Screen is a screen in which the user can view and update the app settings. Similar to the creation flow, the user will be able to change Name, Occupation, Date of Birth, and Notification Level. Upon update, the app send change to the server for validation. This screen can be accessed from either the Home screen or the Stress Log screen. The user can also view the current device and Id associated with account.

Interactive Interfaces

- **Name Text Field**
  - On selection, a keyboard prompt will appear
  - Supports text highlighting and copy / paste functionality

- **Occupation Dropdown**
  - On selection, a list of available occupations will appears
  - Selecting an occupation will close the dropdown and set the field to the selection

- **Date of Birth Date picker**
  - On selection, a date picker will appears
  - The date picker will support, Month-Day-Year selection
  - Selecting a date will close the picker and set the field to the selection

- **Notification Level Dropdown**
  - On selection, a list of notification levels will appears
    - Will include High, Medium, Low, None
  - Selecting a level will close the dropdown and set the field to the selection

- **Update Button**
  - On selection, the app will save settings to client and server

- **Home Tab**
  - On selection, the app will transition to the Stress Log screen

- **Settings Tab**
  - On selection, the app will transition to the Settings screen

Supported Notifications / Error States

- **Mandatory Field Notification**
  - Indicates that Name, Occupation, or Date of Birth are not set

- **Device Not Supported**
  - Indicates which devices are supported by the app

- **Stress Event Detected**
  - Notifies the user that they are currently stressed
  - Suggest ways to mitigate and resolve event
  - Option to indicate whether the suggestion helped

- **Contextual Request Notification**
  - Appears randomly through the day
On selection, the user will be transition to the screen.
Survey Screen

The Survey Screen is a screen in which will fill out a user contextual information. This screen can be access the survey notification prompt. The question listed on the screen are dynamic in nature; there is no preset question set.

Interactive Interfaces

- Text Fields
  - On selection, a keyboard prompt will appear
  - Supports text highlighting and copy / paste functionality
- Dropdowns
  - On selection, a list of available occupations will appears
  - Selecting an occupation will close the dropdown and set the field to the selection
- Sliders
  - The user can select a number from a range
  - Left and right arrows are used to show more options
  - Selecting a date will close the picker and set the field to the selection
- Submit Button
  - On selection, the app will save the results of the survey then transition the user back the screen that they were previously on
- Up and Down Arrows
  - On selection, the app will scroll the page to the desired position of the survey
- Cancel Button
  - On selection, the app will discard all responses and transition back to previous screen

Supported Notifications / Error States

- Mandatory Field Notification
  - Indicates that a survey question is not answered
- Stress Event Detected
  - Notifies the user that they are currently stressed
  - Suggest ways to mitigate and resolve event
  - Option to indicate whether the suggestion helped
Figure 29 - iOS Stress Event Log View
About Us Screen

The About Us Screen, also known as More Information, is a screen in that provides information on the use of the app, the purpose of the app, supported devices, and any required legal acknowledgements. This screen does not require an active account to access; the screen can be accessed from the login screen. This screen can also be accessed from the settings screen. Figure XX.X.

Interactive Interfaces

- Text Modules
  - Text that describes the current sections
- External Link
  - A link to a website for additional information of the current topic
- Up and Down Arrows
  - On selection, the app will scroll the page to the desired position of the page
- Back Button
  - On selection, the app will transition back to the previous screen

Supported Notifications / Error States

- Stress Event Detected
  - Notifies the user that they are currently stressed
  - Suggest ways to mitigate and resolve event
  - Option to indicate whether the suggestion helped
Screen Communication

The diagram below displays how each screen interacts with each other. Not all error states are present.

Figure 30 - iOS Screen Connectivity
4.3 Server

4.3.1 Technologies

**Node.js**

We chose Node.js for a multitude of reasons, not the least of which being performance. Node.js uses Chrome’s V8 engine along with a non-blocking, asynchronous, I/O providing the ability to build highly scalable, fast web servers. It was critical that we chose a platform in which we could scale exponentially while still keeping performance high. Our server will be calling approximately 150 requests per hour per user. As our user base grows the amount of requests the server needs to handle will grow at a rate of \((150 \times 24 \times \#\text{ new users})\) per day. With the asynchronous nature of Node.js, it will give us the ability to spin up multiple instances of the server, without explicitly worrying about data dependencies.

Node.js also has an amazing package manager (npm) which provides ready built packages that can be plugged into your code to abstract out different tasks. This brings me to another reason we opted to go with Node.js and that is the node package manager (npm). Npm provides us with many packages that have made tasks such as writing our own RESTful web server (Express.js), authenticating users with Fitbit (Fitbit-oauth2), MongoDB object modeling (mongoose), an extensible multi-core server manager (cluster), password salt and hashing (bcrypt), session storing (express-session), and a myriad of other packages that are explained in detail below.

We also decided to go with Node.js for reasons that are more subjective. Numerous group members have already had experience developing using Node.js shrinking the learning curve of other group members and giving us more time to focus on other aspects of the project. The members who did not have any experience with Node.js also showed favoritism towards it because they wanted to learn it.
REST

We decided to build our API using Representational State Transfer (REST). REST uses simple HTTP calls for machine-to-machine communication with message-based calls that are reliant on the HTTP standard protocol. Using the HTTP standard protocol means using a simple request, response system. RESTful API's take advantage of HTTP Verbs such as GET, POST, PUT, and DELETE to make requests. GET is used when retrieving resources from a web server while DELETE is used when removing resources from a web server. POST is used when creating new resources while PUT is generally used when updated a resource.

One key component to a RESTful API is that there is no dependency of the client-side technology being used. For example if we would like to expand and develop an Android application, we would be able to access our API just the same as on iOS. A second key component to a RESTful API is that it is considered stateless; meaning that the API is not tied to a particular server. This is important for scalability. When dealing at scale with a cluster of servers there is no way to guarantee that a certain request will always hit the same server, which is determined by a load-balancer. A third key component to a RESTful server is that it supports caching, an important aspect in scaling. Supporting caching is important in scaling because if a request is made many times we can cache and access that rather than fulfilling the request numerous times. With REST you can choose your datatypes, meaning you are free to use whatever fits your needs best, in our case JSON.
REST vs SOAP

Simple Object Access Protocol is another common protocol to developing web services. SOAP is more focused on exposing different pieces of logic as a service; essentially, it exposes functions for completing tasks. SOAP is good for applications that require a formal contract between the API and client because it enforces them by using Web Services Description Language (WSDL). SOAP also has built-in stateful operations, which was purposefully designed to support conversational state management. When using SOAP you have no choice to use XML. For some web services, these are invaluable features.

<table>
<thead>
<tr>
<th>#</th>
<th>SOAP</th>
<th>REST</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A XML-based message protocol</td>
<td>An architectural style protocol</td>
</tr>
<tr>
<td>2</td>
<td>Uses WSDL for communication between consumer and provider</td>
<td>Uses XML or JSON to send and receive data</td>
</tr>
<tr>
<td>3</td>
<td>Invokes services by calling RPC method</td>
<td>Simply calls services via URL path</td>
</tr>
<tr>
<td>4</td>
<td>Does not return human readable result</td>
<td>Result is readable which is just plain XML or JSON</td>
</tr>
<tr>
<td>5</td>
<td>Transfer is over HTTP. Also uses other protocols such as SMTP, FTP, etc.</td>
<td>Transfer is over HTTP only</td>
</tr>
<tr>
<td>6</td>
<td>JavaScript can call SOAP, but it is difficult to implement</td>
<td>Easy to call from JavaScript</td>
</tr>
<tr>
<td>7</td>
<td>Performance is not great compared to REST</td>
<td>Performance is much better compared to SOAP - less CPU intensive, leaner code etc.</td>
</tr>
</tbody>
</table>

*Figure 31 - SOAP vs REST*
NPM Modules

Express
- Express is the standard server framework for Node.js. From the Express website, "Express is a minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications". Express provides you with the tools to easily organize your web application into an MVC architecture server-side. Express makes developing a RESTful web server easy, providing the ability to get a local server up and running in three lines of code. Express also provides built in functionality to easily make HTTP requests while still keeping our code readable.

Mongoose
- Mongoose is a MongoDB wrapper for Node.js helping us easily model our objects. Mongoose uses a Schema-based solution to model application data where each Schema has its own properties or keys. When including a Schema we must convert it to a Model in order to be able to work with it. Each instance of a Model is a Document. Documents can contain their own custom instance methods, which we can define to do commonly used tasks. An example of Mongoose’s Schema-based models is shown in the Database section on Pages 70-72.

Body-parser
- Body-parser is a Node.js middleware for parsing HTTP request bodies. This module provides parsers for JSON body, Raw body, Text body, and URL-encoded body

Fitbit-oauth-2
- Fitbit-oauth2 is a client library, which supports interfacing with the Fitbit API using OAuth2. This library implements the authorization code grant flow for Fitbit. It provides the functionality to send users to the browser in order to authorize and obtain a Fitbit token. Fitbit-oauth-2 also allows a web server to make Fitbit API calls using a persisted token while automatically handling token refreshes.
Forcast.io

- The Forcast.io module provides a simple wrapper for the Forecast.io API. Forecast.io is an easy, advanced, weather API available for developers. This module provides the ability to easily identify yourself as a Forecast.io developer using your unique API key. Once identified as a developer Forcast.io provided 1000 free API calls a day for weather data. The data accessible can be as simple as temperature based on longitude and latitude or as complex as humidity percentage at a certain altitude up to 60 years in the past.

Bcrypt

- The bcrypt module is a bcrypt library for Node.js. Bcrypt itself is a key derivation function for passwords, based on the Blowfish cipher. The bcrypt function is a password-hashing algorithm. The module provides the functionality to generate a custom salt and hash code based on the user’s password.

4.3.2 Implementation

For our implementation we plan to have our server expose endpoints for handling user based logic such as registration, login, authentication, settings, logout, etc. We will also be exposing endpoints for querying data based on user ID. Our classifier will need a way to retrieve data in order to determine if the user is stressed. We first thought we would need to have our own endpoints to retrieve data from Fitbit’s servers but after a discussion, we realized this was not our best option. We realized that we are only passively collecting data with the only piece needed from the user is their Fitbit ID which, as long as they are logged in, we have access to from their Fitbit token.

We determined the best way for us to collect data would be to run a script on a continuous timer. We will be pulling groups of users at a time from the database; we will then make the group of Fitbit API calls for each user, store this data, and then pull the next group of users. We are also going to cache the most recent data so that we can reduce the time it takes to retrieve data. There are many possible issues with this model and they are discussed below.
4.3.3 Possible Issues and Solutions

There are many possible issues we could run into with the implementation of our server. There is a known bug with Fitbit's iOS app. This is that their “All-Day Sync” feature actually does not synchronize data to Fitbit all day. One issue we could face is the possibility of trying to retrieve data from Fitbit and it not being there yet. This could mean missing large chunks of data and not providing the user with the best analysis possible. A solution we have discussed for this is to check if Fitbit has actually returned data. If so, we have no issue and should continue as planned. If Fitbit did not return any data, we will add our query to a Queue and plan to re-query the data from Fitbit at a certain time interval (10-15 minutes).

A second issue we could face is getting the data to our classifier in time to do an analysis and alert the user if stress is detected. We believe with the amount of API calls we are going to be making may incur a time penalty when trying to retrieve data. A way we foresee getting past this issue is spinning up multiple instances of our server and having a load balancer make sure no one server is overloaded with requests. Luckily, Node.js provides us with a great npm module cluster, which is an extensible multi-core server manager.

We also see an issue with our endpoints not being secure. For the endpoints that require a Fitbit ID, we are not worried about security because they will not execute without a valid, active, Fitbit token. For our endpoints that do not require a Fitbit token, we plan to issue sessions to valid users where each endpoint will first check if the user has a valid session prior to executing the logic.
4.3.4 API Documentation

GET /authenticate

The /authenticate endpoint is hit when a user first opens their application or when they have to be reauthenticated for various reasons.

Example usage:

http://localhost:8080/authenticate

Parameter

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>success</td>
<td>Boolean</td>
<td>true</td>
</tr>
</tbody>
</table>

Success 200

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>success</td>
<td>Boolean</td>
<td>true</td>
</tr>
</tbody>
</table>

Error 4xx

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>success</td>
<td>false</td>
</tr>
<tr>
<td>err_message</td>
<td>Description of the error causing failure.</td>
</tr>
</tbody>
</table>

Figure 32 - /authenticate Endpoint

GET /callback

The /callback endpoint is hit when Fitbit correctly authenticates a user and sends the authentication token back to our server. If the user is not in our database they will be redirected to /register. Otherwise their authentication token will be updated in the database.

Example usage:

http://localhost:8080/callback

Parameter

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>success</td>
<td>Boolean</td>
<td>true</td>
</tr>
</tbody>
</table>

Success 200

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>success</td>
<td>false</td>
</tr>
<tr>
<td>err_message</td>
<td>Description of the error causing failure.</td>
</tr>
<tr>
<td>redirect</td>
<td>&quot;/register&quot;</td>
</tr>
</tbody>
</table>
**POST**

/registerUser

The /registerUser endpoint is hit when a new, non-health professional, user is being added to the database. The data is sent to the database as long as the user's authentication is valid and they provided the correct parameters.

Example usage:

```
http://localhost:8000/registerUser
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>firstname</td>
<td>String</td>
<td>The user's first name.</td>
</tr>
<tr>
<td>lastName</td>
<td>String</td>
<td>The user's last name.</td>
</tr>
<tr>
<td>dob</td>
<td>String</td>
<td>The user's Date of Birth.</td>
</tr>
<tr>
<td>profession</td>
<td>String</td>
<td>The user's profession</td>
</tr>
<tr>
<td>gender</td>
<td>String</td>
<td>The user's gender.</td>
</tr>
</tbody>
</table>

**Success 200**

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>success</td>
<td>String</td>
<td>true</td>
</tr>
</tbody>
</table>

**Error 4xx**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>success</td>
<td>false</td>
</tr>
<tr>
<td>err_message</td>
<td>Description of the error causing failure.</td>
</tr>
</tbody>
</table>

Figure 34 - /registerUser Endpoint

**POST**

/registerHealthProfessional

The /registerHealthProfessional endpoint is hit when a new, health professional, user is being added to the database. The user signs up with their email address and password. The password is provided a custom salt and hashed using bcrypt. It then stores the health professional in the database with their salt and hashcode as long as they provide the correct parameters.

Example usage:

```
http://localhost:8000/registerHealthProfessional
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>firstname</td>
<td>String</td>
<td>The health professional's first name.</td>
</tr>
<tr>
<td>lastName</td>
<td>String</td>
<td>The health professional's last name.</td>
</tr>
<tr>
<td>email</td>
<td>String</td>
<td>The health professional's email address.</td>
</tr>
<tr>
<td>profession (optional)</td>
<td>String</td>
<td>The health professional's profession.</td>
</tr>
</tbody>
</table>

**Success 200**

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>success</td>
<td>String</td>
<td>true</td>
</tr>
</tbody>
</table>

**Error 4xx**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>success</td>
<td>false</td>
</tr>
<tr>
<td>err_message</td>
<td>Description of the error causing failure.</td>
</tr>
</tbody>
</table>

Figure 35 - /registerHealthProfessional Endpoint
PUT

/verifyHealthProfessional

The /verifyHealthProfessional endpoint is hit when an admin approves a health professional for use of the Stress Health Platform. It changes the health professionals is_verified flag to true. Gives health professional access to the data.

Example usage:

HTTP://localhost:8080/verifyHealthProfessional

Parameter

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>email</td>
<td>String</td>
<td>Health Professional's Email address to verify.</td>
</tr>
</tbody>
</table>

Success 200

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>success</td>
<td>Boolean</td>
<td>true</td>
</tr>
</tbody>
</table>

Error 4xx

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>success</td>
<td>false</td>
</tr>
<tr>
<td>err_message</td>
<td>Description of the error causing failure.</td>
</tr>
</tbody>
</table>

Figure 36 - /verifyHealthProfessional Endpoint

GET

/logout

The /logout endpoint will be hit when the user decides to logout of their account. It will destroy their current session and redirect them to the home page.

Example usage:

HTTP://localhost:8080/logout

Parameter

Success 200

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
</table>

Error 4xx

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
</table>

Figure 37 - /logout Endpoint
**GET**

`/getUserInfo`

The `/getUserInfo` endpoint will be hit when a user accesses their profile/settings page. It will access the user by their current fitbit session token. It will return a JSON object of the user.

Example usage:

```
http://localhost:8080/getUserInfo
```

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>success</td>
<td>Boolean</td>
<td>true</td>
</tr>
<tr>
<td>user</td>
<td>JSON</td>
<td>A JSON Object containing all the user information.</td>
</tr>
</tbody>
</table>

**Error 4xx**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>success</td>
<td>false</td>
</tr>
<tr>
<td>err_message</td>
<td>Description of the error causing failure.</td>
</tr>
</tbody>
</table>

Figure 38 - `/getUserInfo` Endpoint

**GET**

`/getHealthProfessionalInfo`

The `/getHealthProfessionalInfo` endpoint will be hit when a health professional accesses their profile/settings page. It will access the user by their current session. It will return a JSON object of the health professional.

Example usage:

```
http://localhost:8080/getHealthProfessionalInfo
```

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>success</td>
<td>Boolean</td>
<td>true</td>
</tr>
<tr>
<td>user</td>
<td>JSON</td>
<td>A JSON Object containing all the health professional information.</td>
</tr>
</tbody>
</table>

**Error 4xx**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>success</td>
<td>false</td>
</tr>
<tr>
<td>err_message</td>
<td>Description of the error causing failure.</td>
</tr>
</tbody>
</table>

Figure 39 - `/getHealthProfessionalInfo` Endpoint
**PUT**

/updateUserInfo

The /updateUserInfo endpoint will be hit when a user wants to update some of their information. It will access the user by their current fitbit session token.

Example usage:

http://localhost:8000/updateUserInfo

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>firstName</td>
<td>String</td>
<td>The user's first name.</td>
</tr>
<tr>
<td>lastName</td>
<td>String</td>
<td>The user's last name.</td>
</tr>
<tr>
<td>dob</td>
<td>String</td>
<td>The user's Date of Birth.</td>
</tr>
<tr>
<td>profession</td>
<td>String</td>
<td>The user's profession.</td>
</tr>
<tr>
<td>gender</td>
<td>String</td>
<td>The user's gender.</td>
</tr>
</tbody>
</table>

Success 200

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>success</td>
<td>Boolean</td>
<td>true</td>
</tr>
</tbody>
</table>

Error 4xx

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>success</td>
<td>false</td>
</tr>
<tr>
<td>err_message</td>
<td>Description of the error causing failure.</td>
</tr>
</tbody>
</table>

**Figure 40 - /updateUserInfo Endpoint**

**PUT**

/updateHealthProfessionalInfo

The /updateHealthProfessionalInfo endpoint will be hit when a health professional wants to update some of their information. It will access the user by their current session.

Example usage:

http://localhost:8000/updateHealthProfessionalInfo

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>firstName</td>
<td>String</td>
<td>The health professional's first name.</td>
</tr>
<tr>
<td>lastName</td>
<td>String</td>
<td>The health professional's last name.</td>
</tr>
<tr>
<td>profession</td>
<td>String</td>
<td>The health professional's profession.</td>
</tr>
</tbody>
</table>

Success 200

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>success</td>
<td>Boolean</td>
<td>true</td>
</tr>
</tbody>
</table>

Error 4xx

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>success</td>
<td>false</td>
</tr>
<tr>
<td>err_message</td>
<td>Description of the error causing failure.</td>
</tr>
</tbody>
</table>

**Figure 41 - /updateHealthProfessional Endpoint**
**API Endpoints Not Listed**

We have not documented any of our endpoints for pulling data from our database. This is because as a group we have not decided what would be best for our classifier. We have not decided if we should pass the classifier a bulk amount of data and have it determine what type of data it is getting or if we should pass it small chunks of specific data at a time. We also are not sure if we want to allow the classifier to determine the period of data it wants or if we will be passing it data based on what times we deem fit. We also have not documented endpoints for health professional login. We will be concatenate the health professional’s password and salt, re-hash it and compare it to their stored hash code. If they are the same, the user is granted login.
4.4 Database

4.4.1 Technologies

**MongoDB**

MongoDB is a NoSQL database meaning it is a database that provides a mechanism for storage and retrieval of data, which is modeled, rather than data stored in tabular relations.

We decided to go with MongoDB for many reasons, one of which being that it stores JSON objects. The data we are getting back from the Fitbit API requests are coming in as JSON objects so storing them as is becomes infinitely easier for us rather than trying to get them to fit a certain data-table schema.

We also chose MongoDB because it provides high performance and easy scalability. MongoDB excels in three different metrics when discussing scalability. One of which is cluster scale. Mongo's auto-sharding makes it possible to distribute a database across 100's of nodes. Sharding is the process of storing data records across multiple machines; adding more machines to support data growth and the demands of read and write operations. We believe that, with the amount of reads and writes we are doing and if our user base gets large enough, we will reach a point where one server will not suffice without incurring heavy time penalties. MongoDB having the ability to auto-shard will make it easy for us to scale across numerous servers.

A second metric MongoDB excels in is performance scale, meaning it can sustain a large number of database reads and writes while still maintaining strict latency SLA’s. While we do not believe, we will be dealing with a large enough number of requests per second for this to be an advantage for us it still is a performance advantage over other databases.

The third metric that makes MongoDB the smart choice when discussing scalability is data scale. MongoDB provides the ability to store petabytes of data while still query and access this data at any time without a performance hit. With the amount of data we are storing per user, it is pertinent that we can still access data from any time period on-demand.
Relational vs NoSQL

Relational databases such as MySQL are structured to recognize relations among stored items. It stores items in a set of formally described tables separated into rows and columns. The rows in a relational database table contain all of the information about one data point. The columns are each one separate data point. Relational databases allows users to access or reassemble data in many different ways using joins without reorganizing the database tables. It is best to use a relational database when you have a well-structured set of data that have a natural relation to each other.

NoSQL databases such as MongoDB are structured to store data in in different ways, the popular ones being as documents, graphs, key-value, and columnar. For our case, we will discuss the data storage model MongoDB uses, documents. Document databases like MongoDB have a powerful query engine and indexing features, making it easy and fast to execute different optimized queries.

NoSQL databases are best used when there is no real relation between the data so that new fields can be added to Schema’s as need be. In our case, we went with MongoDB because although each user is related to their data points we do not explicitly need to model this in our database. Rather we store each data point with a user ID attached. This is best for us because each data point has its own parameters and being able to store them as is without conforming to a set Schema is a huge advantage for us.

<table>
<thead>
<tr>
<th>Relational DB vs. NOSQL DB</th>
<th>RDB (i.e. MySQL)</th>
<th>Document Store (i.e. MongoDB)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DB Schema</strong></td>
<td>Relational Model, Hard for graph model</td>
<td>Complete schema-less</td>
</tr>
<tr>
<td><strong>Performance</strong></td>
<td>Too many join for graph model</td>
<td>High read performance; Potential write performance bottleneck</td>
</tr>
<tr>
<td><strong>Scalability</strong></td>
<td>Difficult to scale-cut (manual sharding)</td>
<td>Auto-sharding on pre-defined shard key</td>
</tr>
<tr>
<td><strong>Query</strong></td>
<td>SQL</td>
<td>Limited query language (no join)</td>
</tr>
<tr>
<td><strong>Consistency</strong></td>
<td>ACID Transactional</td>
<td>Eventual Consistency</td>
</tr>
<tr>
<td><strong>Concurrency Control</strong></td>
<td>Locking or MVCC</td>
<td>node-level locking &amp; atomic operation</td>
</tr>
<tr>
<td><strong>Security</strong></td>
<td>AuthZ &amp; AuthN</td>
<td>Basic security</td>
</tr>
<tr>
<td><strong>Notification Mechanism</strong></td>
<td>Trigger</td>
<td>No build-in notification</td>
</tr>
</tbody>
</table>
JSON

JSON or JavaScript Object Notation is a common syntax for storing and exchanging data, an easy alternative to XML. JSON is more readable and easier to manipulate than other alternatives such as XML. Below is an example of passing the same data using JSON vs XML.

```
{"employees": [  
    {"firstName":"John", "lastName":"Doe"},  
    {"firstName":"Anna", "lastName":"Smith"},  
    {"firstName":"Peter", "lastName":"Jones"}  
]
```

*Figure 43 - JSON Example*

```
<employees>
    <employee>
        <firstName>John</firstName> <lastName>Doe</lastName>
    </employee>
    <employee>
        <firstName>Anna</firstName> <lastName>Smith</lastName>
    </employee>
    <employee>
        <firstName>Peter</firstName> <lastName>Jones</lastName>
    </employee>
</employees>

*Figure 44 - XML Example*
4.4.2 Implementation

We have broken our database into three models: User, Log, and HealthProf.

User
The user model contains all the data that is stored for each user.

```
var userSchema = new Schema({
  user_id: {
    type: String,
    unique: true
  },
  first_name: String,
  last_name: String,
  dob: String,
  profession: String,
  gender: String,
  is_admin: false,
  token: {
    access_token: String,
    expires_in: Number,
    refresh_token: String,
    scope: String,
    token_type: String,
    user_id: String,
    expires_at: String
  }
});
```

*Figure 45 - User Database Schema*

**user_id:**
- The user_id key will be the user Fitbit ID and is guaranteed to be unique.

**first_name:**
- The first_name key will be the user’s first name.

**last_name:**
- The last_name key will be the user’s last name.

**dob:**
- The dob key will be the user’s Date of Birth.

**profession:**
- The profession key will be the user’s profession.

**gender:**
- The gender key will be the user’s gender.

**is_admin:**
- The is_admin flag is set to false as a default. Users can only be marked as admin if approved by a different admin. Admin status is set only for us as a group to use for testing purposes.

**token:**
- The token key is a JSON object returned from Fitbit to keep users authenticated so that we will have access to their data. The token is refreshed after an expiration occurs to persist the users session.
Log

The log model is contains all the data stored for each log (data point).

The user_id key will be the user Fitbit ID that will tie a specific user to this specific log.

The log_type key is a numerical value based on what type data point it is.

The data key is a JSON contains the actual data for each log, which is different for each log type.

The timestamp key is a JSON object with two keys, date and time. Date contains the date the log was collected in the format of YEAR-MONTH-DAY (1234-01-23). Time contains the time the log was collected in the format of HOUR:MINUTE:SECOND (01:01:01).
HealthProf

The HealthProf model contains all the data stored for each Health Professional.

```javascript
var healthProfSchema = new Schema({
  email: {
    type: String,
    unique: true
  },
  first_name: String,
  last_name: String,
  hash_code: String,
  salt: String,
  is_admin: false,
  is_verified: false,
  profession: String
});
```

**email:**
- The email key will be the health professional’s email address and is guaranteed to be unique.

**first_name:**
- The first_name key is the health professional’s first name.

**last_name:**
- The last_name key is the health professional’s last name.

**hash_code:**
- The hash_code key is the health professional’s unique hash code provided from bcrypt. We will use the hash_code key to authenticate the health professional’s during login.

**salt:**
- The salt key is the health professional’s unique salt provided from bcrypt. We will use the salt function to authenticate the health professional’s during login.

**is_admin:**
- The is_admin flag is by default false. Admin’s will only be Julie and Andrew (Our sponsors). Admin’s will have the ability to verify other health professionals.

**is_verified:**
- The is_verified flag is by default false. A health professional can apply to be verified. If the flag is true the health professionals will be able to view the Stress Health data on the web dashboard.

**profession:**
- The profession flag is going to be a description of the health professional’s job. This will be used to help admins verify them.
4.5 Machine Learning Classifier

Our project requires us to not only classify each data set that comes into the system, but also have the system adapt to each individual user. That need for adaptation is a perfect indicator for needing to use some Machine Learning technique in our solution. Our group has little experience with Machine Learning techniques, so we needed to find a library that was powerful, effective, and would work within a Swift project so that we could write our app natively for iOS. To fulfill these requirements, we settled on using Dlib C++, a machine-learning library written in C++, which we could bring into our XCode project as part of an Objective-C header. The algorithms provided in Dlib C++ are numerous, but the library’s website provides a convenient flowchart for choosing the algorithm which best fits any specific use case.

![Dlib C++ Algorithm Selection Flowchart](image)

We are predicting true/false labels (in the form of stressed/unstressed), our data will be labeled for training purposes, and we have fewer than 100 features. This points us nicely to the krr_trainer with a radial_basis_kernel. In order to make this easily usable from our iOS app, we built a simple C++ class that acts as a wrapper around the Dlib C++ library calls, allowing the Swift code in the app to worry only about passing the correct data in.
The `krr_trainer` used in Dlib C++ uses a kernelized ridge regression algorithm to train a decision function. From the Dlib C++ documentation:

“This object represents a tool for performing kernel ridge regression (This basic algorithm is also known by many other names, e.g. regularized least squares or least squares SVM).

The exact definition of what this algorithm does is this:

Find \( w \) and \( b \) that minimizes the following (\( x_i \) are input samples and \( y_i \) are target values):

\[
\lambda \cdot \text{dot}(w, w) + \sum_{i} \left( (f(x_i) - y_i)^2 \right)
\]

where \( f(x) = \text{dot}(x, w) - b \)

except the dot products are replaced by kernel functions. So this algorithm is just regular old least squares regression but with the addition of a regularization term which encourages small \( w \) and the application of the kernel trick.”

Figure 50 - Machine Learning Training Cache

As data is read in by the app it gets labeled by the classifier. These labels may not be accurate, so the app sends the user periodic notifications asking about their stress. Their answers are used to validate the entries for use in training.

Entries are added into the cache as they are validated by the app. These are stored for future training locally on the user’s phone. Once the cache is filled with enough entries (e.g., 1000) the `trainClassifier()` function is called.
This is a broad view of how our classifier will interact with the incoming data, and how it will train on it.

For our trainer to work, we need to make sure we have labeled data. The classifier labels each entry as stressed or unstressed, but these labels may not be correct. Each entry from each moment in time has to be labeled as stressed or unstressed before the trainer can correctly use it to produce a learned function. To solve this problem, we have decided to use user prompts--periodic questions displayed to the user asking about their stress--to validate the labels given by the classifier.

There is also the interesting problem of how to initialize a classifier for new users. Each person responds to stress in different ways, so we need each user to have their own classifier, which learns about them personally. The difficulty comes in creating the first classifier that will initially be part of a new user's app. A general purpose classifier is likely to be inaccurate for most people.

In order to get around this, we have decided that a new user will have an initial evaluation period without a classifier. During this period, the user's data will still be collected by the system as normal, but each entry will be both labeled and validated by the prompts we send to the user. These entries will be cached, and once enough have been validated, a classifier will be trained from the initial cache of data.

In order to develop the classifier module, we started by making a prototype that can be run from a terminal as a proof-of-concept. To simplify some of the data inputs, enums were made for the weather and sleep data. Only four data points were included in each entry: heart rate, temperature, weather, and sleep.

Included below are the prototypes and typedefs from our prototype classifier module. Any additional or more specific data that needs to be given to the classifier can be added to the Entry struct when integrating with Swift.
```c++
#include <stdio.h>
#include <iostream>
#include <string>
#include <dlib/svm.h>

using namespace std;
using namespace dlib;

typedef enum
{
    SUNNY,
    OVERCAST,
    RAIN,
    SNOW,
    STORM
} WEATHER;

typedef enum
{
    BAD,
    POOR,
    GOOD,
    GREAT
} SLEEP;

typedef matrix<double, 4, 1> sample_type;
typedef radial_basis_kernel<sample_type> kernel_type;
typedef decision_function<kernel_type> dec_funct_type;
typedef normalized_function<dec_funct_type> funct_t;

typedef struct _entry Entry;
struct _entry
{
    int heartRate;
    int temperature;

    WEATHER weather;
    SLEEP sleep;
};

class Classifier
{
private:
    funct_type learned_function;

public:
    Classifier();
    Classifier(char* path);
    double classify(Entry* en);
    void trainClassifier(char* cache);
    void saveClassifier(char* path);
    double test(char* path);
};
```
The Classifier class has two constructors, one of which is empty. The non-empty constructor takes a file path, which points to a file containing a saved classifier from a previous training instance. This constructor then loads the learned function from that file. This allows the app to save the classifier between sessions and across phone restarts.

```cpp
Classifier::Classifier () {}
Classifier::Classifier (char* path)
{
    deserialize(path) >> learned_function;
}
```

The classify() function accepts an entry and returns the value generated by the learned function of the classifier. The returned value is a 1 for stressed, and a -1 for not stressed.

```cpp
double Classifier::classify(Entry* en)
{
    sample_type sample;
    sample(0) = (double) en->heartRate;
    sample(1) = (double) en->temperature;
    sample(2) = (double) en->weather;
    sample(3) = (double) en->sleep;

    return learned_function(sample);
}
```

The trainClassifier() function accepts a file path which points to the cache. Each entry from the cache is loaded into the memory and the values are normalized. A krr_trainer is used to train a new classifier, which is then saved to disk.
void Classifier::trainClassifier(char* cache)
{
    //Load cache from file
    ifstream file(cache);
    std::vector<sample_type> samples;
    std::vector<double> labels;

    string line;
    do
    {
        sample_type sample;
        double label;

        getline(file, line);
        stringstream items (line);

        items >> sample(0);
        items >> sample(1);
        items >> sample(2);
        items >> sample(3);

        items >> label;

        samples.push_back(sample);
        labels.push_back(label);
    } while (!file.eof());

    //Create normalized training set
    vector_normalizer<sample_type> normalizer;
    normalizer.train(samples);

    for (int i = 0; i < samples.size(); i++)
    {
        samples[i] = normalizer(samples[i]);
    }

    krr_trainer<kernel_type> trainer;
    trainer.use_classification_loss_for_loo_cv();

    cout << "doing leave-one-out cross-validation" << endl;
    for (double gamma = 0.000001; gamma <= 1; gamma *= 5)
    {
        // tell the trainer the parameters we want to use
        trainer.set_kernel(kernel_type(gamma));

        // loo_values will contain the LOO predictions for each sample. In the case
        // of perfect prediction it will end up being a copy of labels.
        std::vector<double> loo_values;
        trainer.train(samples, labels, loo_values);

        // Print gamma and the fraction of samples correctly classified during LOO cross-validation.
        const double classification_accuracy = mean_sign_agreement(labels, loo_values);
        cout << "gamma: " << gamma << " \ LOO accuracy: " << classification_accuracy << endl;
    }
}
cout << "Training is complete" << endl;

trainer.set_kernel(kernel_type(0.000625));

//Train classifier
learned_function.normalizer = normalizer;
learned_function.function = trainer.train(samples, labels);

serialize("saved_classifier") << learned_function;
}

The saveClassifier() function accepts a path which indicates a file location
to save the classifier’s learned function. This should be the same location that the
function is loaded from in the Classifier class’ constructor.

void Classifier::saveClassifier(char* path)
{
    serialize(path) << learned_function;
}

The test() function accepts a file path to a file containing a set of entries to test on.
Each entry is labeled by the classifier and compared to the label saved in the input
file, and the accuracy of the classifier is displayed after the test run.

double Classifier::test(char* path)
{
    ifstream file(path);
    std::vector<sample_type> samples;
    std::vector<double> labels;

    string line;
    do
    {
        sample_type sample;
        double label;

        getline(file, line);
        stringstream items(line);

        items >> sample(0);
        items >> sample(1);
        items >> sample(2);
        items >> sample(3);
        items >> label;

        samples.push_back(sample);
        labels.push_back(label);
    } while (!file.eof());

    double hits = 0;
int i = 0;
for(i = 0; i < samples.size(); i++)
{
    double result = learned_function(samples[i]);
    int adj_result = (result >= 0) ? 1 : -1;

    if((int) labels[i] == adj_result)
    {
        hits++;
    }

    cout << "Sample: " << samples[i] << "\tResult: " << result << endl;
    cout << "Expected: " << labels[i] << endl;
    //printf("Samples: %f\t Result: %fn", samples[i], learned_function(samples[i]));
}

double accuracy = (hits / samples.size()) * 100;

return accuracy;
}

This prototype module reads in data from plain text files generated by a simple C program which randomly generates data entries that have soft correlations between each of the data points.

#include <stdlib.h>
#include <stdio.h>
#include <time.h>

typedef enum
{
    SUNDAY,
    OVERCAST,
    RAIN,
    SNOW,
    STORM
} WEATHER;

typedef enum
{
    BAD,
    POOR,
    GOOD,
    GREAT
} SLEEP;

void genData(FILE* ofp)
{
    int hr, temp, weather, sleep, stress;

    weather = rand() % 5;
sleep = rand() % 4;
stress = (rand() % 2) * 2 - 1;
switch(weather)
{
    case SUNNY: /*temp between 75 and 110*/
        temp = rand() % 35 + 75;
        if(sleep >= GOOD && temp <= 90)
        {
            hr = rand() % 10;
            hr += (stress == -1) ? 60 : 80;
        }
        else if(sleep < GOOD && temp <= 90)
        {
            hr = rand() % 15;
            hr += (stress == -1) ? 65 : 80;
        }
        else if(sleep >= GOOD && temp > 90)
        {
            hr = rand() % 10;
            hr += (stress == -1) ? 70 : 95;
        }
        else if(sleep < GOOD && temp > 90)
        {
            hr = rand() % 15;
            hr += (stress == -1) ? 75 : 95;
        }
        break;
    case OVERCAST: /*temp between 70 and 80*/
        temp = rand() % 10 + 70;
        if(sleep >= GOOD && temp <= 75)
        {
            hr = rand() % 10;
            hr += (stress == -1) ? 60 : 70;
        }
        else if(sleep < GOOD && temp <= 75)
        {
            hr = rand() % 15;
            hr += (stress == -1) ? 65 : 70;
        }
        else if(sleep >= GOOD && temp > 75)
        {
            hr = rand() % 10;
            hr += (stress == -1) ? 70 : 75;
        }
        else if(sleep < GOOD && temp > 75)
        {
            hr = rand() % 15;
            hr += (stress == -1) ? 75 : 75;
        }
        break;
    case RAIN: /*temp between 40 and 70*/
        temp = rand() % 30 + 40;
        if(sleep >= GOOD && temp <= 60)
        {
            hr = rand() % 10;
            hr += (stress == -1) ? 70 : 80;
        }
        else if(sleep < GOOD && temp <= 60)
        {
            hr = rand() % 15;
            hr += (stress == -1) ? 70 : 80;
        }
        else if(sleep >= GOOD && temp > 60)
        {
            hr = rand() % 10;
            hr += (stress == -1) ? 70 : 80;
        }
        else if(sleep < GOOD && temp > 60)
        {
            hr = rand() % 15;
            hr += (stress == -1) ? 70 : 80;
        }
{
    hr = rand() % 10;
    hr += (stress == -1) ? 65 : 75;
}
else if(sleep < GOOD && temp > 60)
{
    hr = rand() % 15;
    hr += (stress == -1) ? 65 : 80;
}break;
case SNOW: //temp between 0 and 40
    temp = rand() % 40;
    if(sleep >= GOOD && temp <= 30)
    {
        hr = rand() % 10;
        hr += (stress == -1) ? 70 : 80;
    }
else if(sleep < GOOD && temp <= 30)
    {
        hr = rand() % 15;
        hr += (stress == -1) ? 75 : 85;
    }
else if(sleep >= GOOD && temp > 30)
    {
        hr = rand() % 10;
        hr += (stress == -1) ? 70 : 80;
    }
else if(sleep < GOOD && temp > 30)
    {
        hr = rand() % 15;
        hr += (stress == -1) ? 75 : 80;
}break;
case STORM: //temp between 50 and 75
    temp = rand() % 25 + 50;
    if(sleep >= GOOD)
    {
        hr = rand() % 10;
        hr += (stress == -1) ? 75 : 80;
    }
else if(sleep < GOOD)
    {
        hr = rand() % 15;
        hr += (stress == -1) ? 75 : 90;
}break;
}
fprintf(ofp, "%d %d %d %d\n", hr, temp, weather, sleep, stress);
return;
```c
int main(int argc, char** argv)
{
    if(argc < 3)
    {
        printf("Insufficient number of arguments: %d\n", argc);
        return 1;
    }

    srand(time(NULL));

    char* dataFile = argv[1];
    int entries = atoi(argv[2]);

    FILE* ofp = fopen(dataFile, "w");
    if(ofp == NULL)
    {
        printf("Invalid output file: %s\n", dataFile);
        return 1;
    }

    int i = 0;
    for(i = 0; i < entries; i++)
    {
        genData(ofp);
    }

    fclose(ofp);

    return 0;
}
```

The generator accepts two arguments: an output file name, and a number of entries to generate. The entries are generated in a form that the prototype module can recognize and operate on. After training the module on 1000 entries, a test on 100 entries came out at ~91% accuracy.

The final classifier module will support the following functions:

- classify()
- trainClassifier()
- saveClassifier()
- buildEntry()
- cacheEntry()
- clearCache()

The functions classify(), trainClassifier(), and saveClassifier(), will all work in the same way as in the prototype module. In addition to these functions, three additional functions will be defined, buildEntry(), cacheEntry(), and clearCache().

The function buildEntry() accepts a set of arguments that make up an Entry. These values are packaged into a new Entry in memory, which is returned.
The function cacheEntry() accepts an Entry struct and a file path that points to the data cache, and appends it to the cache.

The function clearCache() simply deletes the cache file and acts as a location in code to do any file cleanup which needs to be done then.
5. Testing

5.1 Test Plan

The test cases described in this section will help ensure that we deliver a robust, enjoyable product. We will test all aspects of the system, including the user interface, user interactions through the app, researcher interactions through the dashboard, connectivity between component, and error conditions. The goal is to ensure that all functional requirements are implemented and working as intended. This will then allow us to expand the functionality of the system while also maintaining the requirements of our sponsors.

5.1.1 Description of Test Environment

The app will operate on iPhone devices. Two of our team members have iPhone devices and two of our team members will be using iPhone emulators to test our app functionality. The server and database does not require a specific platform to run on. All members have access to a computer with either a windows, Linux, or OSX operating system. The testers will comprise of each group member, with the potential to also have the sponsors use the software to user test the software.

5.1.2 Stopping Criteria

Our team will operate using functionality and usability requirements. A failure in a functionality requirement requires that a stop in development of the particular component to fix the failing requirement. If the problem is a usability issue, development can continue however, a discussing will be held by stakeholders to determine the next course of action. This course of action will be to decide whether or not the requirement is still valid and if still valid, a timeline to approximate when the requirement can be resolved. Once all test have been passed and no mandatory requirements remain, the product will be declared as finalized.
5.2 Description of Individual Test Cases

Each test case explores the functionality around the requirements. All test cases are to be tested only when the module is implemented.

**Test Objective**: Test the account creation flow with a valid account.

- **Test Description**
  - With a valid Fitbit account, go through the creation flow and check that a unique account is created for the user.
  - Client side, distinctness can be checked by the user id that is provided to the user in the settings screen.
  - Server side, distinctness can be checked by the database entry for the user.

- **Expected Results**
  - The user is taken to the new user flow and at completion; a unique account is created for the user that is persistent on both client and server.

**Test Objective**: Test the account creation flow with an invalid state.

- **Test Description**
  - With an invalid state proceed through and ensure that an account is not created,
  - Invalid States include: no Fitbit account, no Fitbit associated with account, decline of terms of service, account fields are not set.

- **Expected Results**
  - The user is provided reason why an account cannot be created. The user is not allowed to access a screen that require a login state login state.

**Test Objective**: Test the use of a Fitbit account with a non-supported device.

- **Test Description**
  - With a non-supported Fitbit device, attempt to create an account or switch after account has been created.
  - Current supported devices are the Fitbit Charge HR.

- **Expected Results**
  - The user receives an on screen notification indicating that the device that is being used is not supported.
  - The user should not be able to use the app until a supported devices is attached to the account.

**Test Objective**: Test the use of an existing user.

- **Test Description**
After an account has been created, attempt to login after the app has been uninstalled the reinstalled or use another device.

- **Expected Results**
  - The server confirms the existence of an account and send the Machine Learning profile to the app.

**Test Objective:** Test the disabling the location services on phone.

- **Test Description**
  - Location services increases the accuracy of the classification process; it is the used in acquiring environmental data.
  - Turning off the services should not stop the app from running.
- **Expected Results**
  - The user is presented with a message stating that location services are currently off and that result may not be optimal as a result.
  - The classifier should continue to function with a slightly lower accuracy.

**Test Objective:** Test the access of each screen and button of those screens in the app.

- **Test Description**
  - Access each screen in a normal state.
  - Use each interactive button that appears on each screen.
  - Account Not Required
    - Login, Registration, and More Information screens.
  - Account Required
- **Expected Results**
  - The user is able to access each screen that appears in the app.
  - Each screen buttons functions as described.

**Test Objective:** Test how the app and dashboard reacts when there is a server outage or the server cannot be reached.

- **Test Description**
  - Disable the internet on the device and attempt to have the app communicate with server.
  - Take the server taken down for maintenance and attempt to have the app communicate with server.
  - Communication events include account creation, login, and internal request for new data.
- **Expected Results**
The app or dashboard should not crash or enter into an unresponsive state. The user should receive an error message indicating to try again later and continue to function as intended.

**Test Objective**: Test Authentication and Authorization of a Researcher Account.

- **Test Description**
  - Register as a new account on the researcher portal dashboard.
  - Have admin approve and deny accounts.
  - Check registration email from admin about account status.

- **Expected Results**
  - The user can create account but cannot access dashboard home page until authorized.
  - The user is given a notification of either denial or approval in form of an email.
  - If authorized, account must be authenticated on every login.

**Test Objective**: Test dashboard querying functionality.

- **Test Description**
  - From the query page, query for both valid and invalid data.
  - Invalid data includes users that do not exist and date ranges that does not contain information.

- **Expected Results**
  - If the data is valid, the information will be returned to the user.
  - If the data is invalid, no data or an error message appears to the user.

**Test Objective**: Test researcher’s ability to set and modify both contextual questions and ways to resolve stress events.

- **Test Description**
  - Access the contextual questions interface, set and delete contextual questions.
  - Access the stress resolution interface, set and delete contextual questions.
  - User must have access to modify these interfaces.

- **Expected Results**
  - If the user has correct permission, the user can update the both questions and resolutions.
  - If the user does not have permission, the changes do not update or the user cannot access the area.
**Test Objective:** Test researcher’s ability to manipulate the data from the participants.

- **Test Description**
  - From query results, save, export, graph, and create reports.
  - Requires enough information to properly see results.
- **Expected Results**
  - The user is able to save and export the data.
  - The user is able to graph and create reports from the data.

**Test Objective:** Test researcher’s ability to set their default reports on the dashboard.

- **Test Description**
  - After a report has been executed, the user can set it to always appear when the user logs.
- **Expected Results**
  - The main page defaults to the settings set by the user.

**Test Objective:** Test the system’s ability to handle input of erroneous data.

- **Test Description**
  - In the settings screen, enter data in the text or date of birth fields that does not make sense.
  - In the survey area, enter information that does not make sense.
- **Expected Results**
  - The system does not enter into a state of malfunction.

**Test Objective:** Test the system’s ability to handle input of excessive data.

- **Test Description**
  - Force an extended update of information by not allowing the app to talk to the server for a prolonged period, 1 - 2 days.
  - Force the server to request for data that overlaps previously obtained data.
- **Expected Results**
  - The system only accepts only the maximum data length supported.
  - The system does not store duplicate data to the database.
**Test Objective**: Test app, dashboard, and server response when server is under heavy load.

- **Test Description**
  - Reach the max supported users and have them simultaneously make server request.
  - Automation out dummy request are required.
- **Expected Results**
  - Response time for request does not exceed data send/retrieve time + 1 minute.

**Test Objective**: Test speed and fluidity of app during transitions and loading.

- **Test Description**
  - Note how smooth the transition between screens.
  - Note how long it takes to load notification, stress events, and surveys.
- **Expected Results**
  - The app is smooth and there is no noticeable lag

### 6. Administrative Content

#### 6.1 Budget and Financing

6.1.1 Developmental Budget

<table>
<thead>
<tr>
<th><strong>Budget/Financing</strong></th>
<th><strong>Quantity</strong></th>
<th><strong>Rate</strong></th>
<th><strong>Total</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Fitbit Device</td>
<td>5</td>
<td>$150 each</td>
<td>$750</td>
</tr>
<tr>
<td>Apple Developer Account</td>
<td>1</td>
<td>$99 per year</td>
<td>$99</td>
</tr>
<tr>
<td>Github Organization*</td>
<td>12</td>
<td>$25 per month</td>
<td>$300</td>
</tr>
<tr>
<td>AWS Hosting Costs</td>
<td>1***</td>
<td>$30 per month</td>
<td>$360</td>
</tr>
<tr>
<td><strong>Grand Total</strong></td>
<td></td>
<td></td>
<td><strong>$1510</strong></td>
</tr>
</tbody>
</table>

*Github organization account may not be necessary--price included for overall approximation

**Total is an approximation and does not include sales tax or scaling costs

***AWS Hosting costs will increase exponentially with scale
## 6.2 Milestones

<table>
<thead>
<tr>
<th>Date</th>
<th>Milestone</th>
</tr>
</thead>
<tbody>
<tr>
<td>3/5</td>
<td>Have a decision made on what back end technologies we are going to use</td>
</tr>
<tr>
<td>3/19</td>
<td>Have our server running and able to interact with our database</td>
</tr>
<tr>
<td></td>
<td>Have our database schema well defined</td>
</tr>
<tr>
<td>3/26</td>
<td>User registration, authentication, login</td>
</tr>
<tr>
<td></td>
<td>Fitbit authentication</td>
</tr>
<tr>
<td>4/3</td>
<td>Ability to access user’s Fitbit data server side</td>
</tr>
<tr>
<td>4/17</td>
<td>iOS app communicating with our server successfully</td>
</tr>
<tr>
<td>8/29</td>
<td>Collecting continuous stream of data for users</td>
</tr>
<tr>
<td>9/12</td>
<td>Version 1 of our Machine Learning algorithm functional</td>
</tr>
<tr>
<td></td>
<td>iOS push notifications working and sending questionnaire data</td>
</tr>
<tr>
<td>9/19</td>
<td>Have login authentication for web dashboard</td>
</tr>
<tr>
<td>9/26</td>
<td>Version 2 of our Machine Learning algorithm functional</td>
</tr>
<tr>
<td>10/10</td>
<td>Web dashboard can display data retrieved from database</td>
</tr>
<tr>
<td></td>
<td>Machine Learning algorithm can determine when it is appropriate to send push notifications to users</td>
</tr>
<tr>
<td>10/24</td>
<td>Machine Learning algorithm is finalized and final testing begins</td>
</tr>
<tr>
<td></td>
<td><em>(12/6-12/12) FINAL PRESENTATION</em></td>
</tr>
</tbody>
</table>
7. Project Summary

Our project aims to help users manage their stress and research the efficacy of intervention methods. We as a group will have to learn a variety of new technologies and methods in order to effectively deliver this project by the end of Senior Design II. Some of the methods we are required to learn and implement are powerful Machine Learning techniques, which will require a great deal of research and effort in order to properly utilize them.

In addition to learning new technologies and methods, all members of our team will need to refine their existing skillsets in order to deliver a quality product that meets our sponsors’ requirements. Our project is going to be subject to the scrutiny of a panel of senior faculty members, so our number one goal as a team is to present a project we are proud to claim we worked on for 9 months.
8. References

[9] https://www.npmjs.com/package/express-sessions
[19] https://dev.fitbit.com/docs/