PulsePI

Final Report

Alexander Blahnik, Elizabeth Leuenberger, Eonshik Kim

CS 595

Professor Mukul Goyal

May 15, 2020

Table of Contents

INTRODUCTION 3
REQUIREMENTS.......... 3
RELATED WORK 4
DESIGN AND IMPLEMENTATION OVERVIEW.......... 5
IMPLEMENTATION SPECIFICS 13
TESTING 21
CONCLUSION.......... 23
APPENDIX A : API UML DIAGRAMS 24
APPENDIX B : LINKS TO REPOSITORIES.......... 25
APPENDIX C : MEMBER TIMELINES.......... 26
REFERENCES.......... 28

Introduction

The development of PulsePI has progressed seamlessly and is now in its' final stages of production. The software team has worked tirelessly over the past few months to deliver this incredible software product. The successful development and deployment of PulsePI marks a new era in the field of cardiovascular health. PulsePI represents the forefront of medical and technological integration. It flawlessly combines modern technology and human health to provide a tool capable of improving lives through the monitoring of cardiovascular health. Welcome to PulsePI.

Requirements

Several requirements were laid out by the software team at the beginning of the development process. These requirements acted as guidelines to ensure that PulsePI was developed and deployed as a fully functional piece of software. Two main feature categories are reflected in the initial requirements: real-time monitoring and historical analysis. PulsePI provides users with real-time heart rate information and displays it in a meaningful way. Additionally, PulsePI stores real-time data that is collected and performs historical analysis on that data. This analysis is made available to users in a straightforward, efficient manner (charts, tables, etc.). The initial requirements are detailed below:

1. *Real-Time Data Updates*

   The PulsePI application should have the ability to receive real-time updates from the Arduino Pulse Sensor. This goal will entail getting the Pulse Sensor to relay information to the Arduino board and then getting the Arduino board to relay information to the PulsePI application.

2. *Useful Relay of Real-Time Updates*

   The PulsePI should have the ability to relay real-time data to a user in a meaningful way. This includes the development of a user interface that can receive real-time updates and display them in a useful fashion. This should include a numerical value indicating a user's heart rate in beats per minute as well as a word describing what the current heart rate could indicate.

3. *Ability to Store Basic User Information*

   The PulsePI should be able to store basic information so that it is possible to provide users with a customized view of their heart rate. This will include a user profile of some kind that obtains certain information from a user and stores it appropriately and securely. Name, age, height, weight and password information are a few of the currently proposed items to store.

4. *Provide a Useful User Interface*

   The PulsePI should provide a useful, user-friendly and chic user interface that makes it ideal for practical everyday use. This will entail a fair amount of design and possibly some marketing research to develop an interface that will be accepted as useful by the general public.

5. Store Heart Rate Data

The PulsePI should be able to store heart rate data for a specific user. This will include communication between our application and database layers via an API. When heart rate data is received by the application, it needs to be relayed to the database for long term storage.

6. Provide the Means for Users to Specify Activities

The PulsePI application should give users the ability to specify their current activity. For example, a user may want to record heart rate data during exercise in order to track their fitness progress. The PulsePI should let a user indicate that they are exercising, resting, etc. during any interval of heart rate data collection. This will include options in the user interface for selecting these activities as well as some database design in order to determine how to store data collected during a certain activity.

7. The Historical Analysis of Heart Rate Data

The PulsePI should be able to analyze historical heart rate data. Any analysis of past data should be customizable to a specific user. That is, a user should be able to choose what data is analyzed and how that analysis is relayed (chart, table, etc.). The user interface should guide a user in choosing the best analyses and display options. Historical analysis of heart rate data will include data retrieval from a database and some algorithmic procedures that provide a user with a meaningful examination of their data.

Related Work

There are several other applications that strive to accomplish similar goals as PulsePI. Two notable technologies, Fitbit and Apple Watch, are outlined below. The existence of these tools affirms the usefulness of PulsePI as an application, as it proves that heart rate monitoring is a sought-after tool in the field of cardiovascular health and fitness.

Fitbit

Fitbit is a fitness tracker. It has a wireless function that allows you to sync with mobile or desktop apps. You can also monitor your heart rate by using its Pure Pulse technology. Pure Pulse can monitor your heart rate even when you’re asleep. When you wake up, it will show your sleep score based on your heart rate. Fitbit also notifies you when you reach weight goals, manages your stress and optimizes your exercise.

Apple Watch

The Apple Watch is a smartwatch, so you can text, call, use Bluetooth and other wireless functions. You can sync with other IOS devices. It monitors and displays your heart rate during various activities.

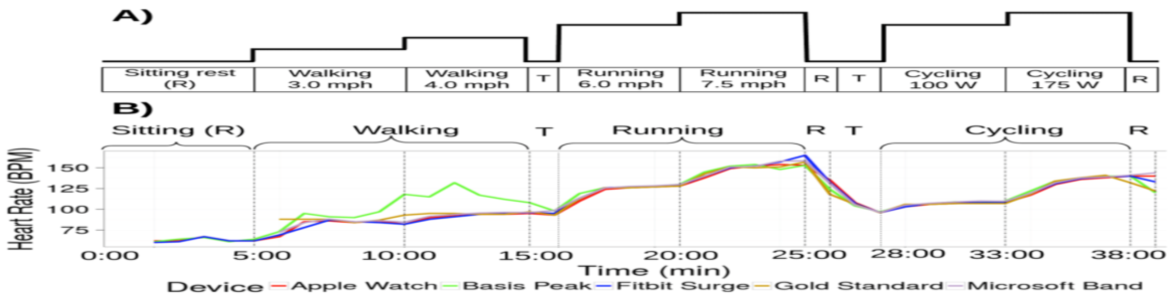


Figure 0: Comparison between Apple Watch, Basic Peak, Fitbit Surge, Microsoft Band, Mio Alpha 2, and Samsung Gear 2.

Other

As mentioned in the caption under Figure 0, there are multiple other tools that do some sort of heart monitoring. These include Basic Peak, Microsoft Band, Mio Alpha 2 and Samsung Gear 2. Garmin also has a tool for heart rate monitoring. As Figure 0 indicates, all these tools provide data at varying levels of accuracy, some better than others. We expect PulsePI to be a leader in all measured fields after its development.

Design and Implementation Overview

The design and implementation of PulsePI can be divided into four main categories: the Arduino-Pulse Sensor, the User Interface, the API, and the database. These individual components were implemented separately and have been integrated to form PulsePI as a whole. Figure 1 below depicts the overall design of the PulsePI software product. The four separate components of PulsePI communicate and work together to deliver quality results. Starting with the Arduino-Pulse Sensor, heart rate data is communicated to the User Interface. The User Interface relays this data to the API, which stores it in the database. Reversing the process, the API retrieves any necessary data from the database and delivers it to the User Interface upon request. The design and implementation of each of the components is reviewed below.

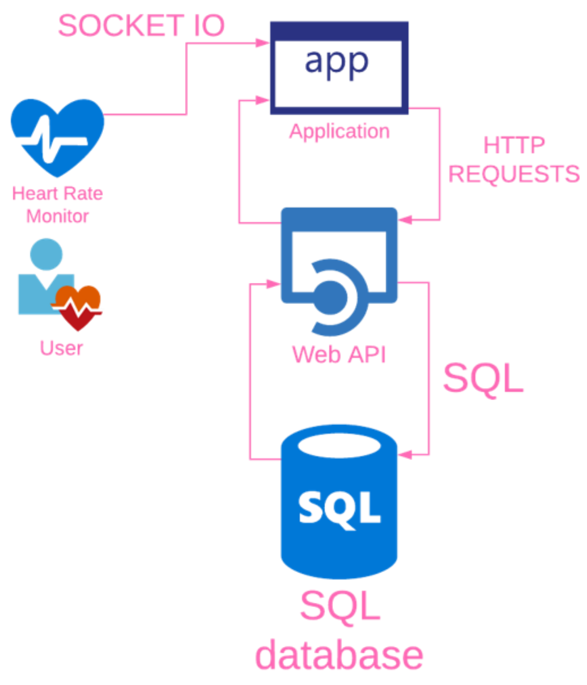


Figure 1: The overall design of PulsePI. The figure depicts multiple components working together to form PulsePI as a whole.

Arduino – Pulse Sensor

The data flow beings with the pulse sensor. The pulse sensor collects the raw data that gets transmitted to the user interface via the Arduino. Figure 2 depicts the data delivery pathway from the pulse sensor to the user interface. The pulse sensor is connected to the Arduino board through 5V and GND digital pins for the power and A0 for the analogue input pin. The Arduino communicates with the user interface through Transmission Control Protocol (TCP), which is established with C++ code (see Figure 3 below). TCP is used rather than the User Datagram Protocol (UDP) to provide reliable data delivery. Since TCP uses the 3-Way Handshake Process, the connection is ensured before any transmission of data. The Arduino sends a SYN (synchronize message) to the UI first to request an open connection. Then the UI responds to the request by sending a SYN-ACK (acknowledgment message). The UI acknowledges the request and asks for an open connection. Lastly, the Arduino sends an ACK (confirmation message) to the UI. Data transmission can then begin.

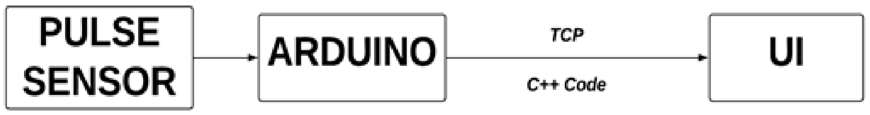


Figure 2: The general design of data input path of the Pulse sensor to the user interface.

The collection of data and establishment of communication with the UI is encoded in C++. Figure 3 shows the overall design of the C++ code. We utilized the Arduino IDE and an existing code base for our implementation. Specifically, we included the "PulseSensorPlayground.h" file that was developed by World Famous Electronics. This library includes several methods for use with the Arduino Pulse Sensor. We retrofitted the "PulseSensor_BPM_Alternative.c" file that was also developed by World Famous Electronics to be more appropriate for our uses. First, our code connects to a WIFI network that is specified by a Service Set Identifier (SSID) and the corresponding password. Next, a server socket is opened. Telnet is used to establish a connection between port number 23 and the TCP. The process then waits for a client to connect, as TCP is a connection-oriented protocol. After a connection is established, any new heart rate data that is registered is written to the socket. A 115200 baud rate is used, so the processing sketch can read the heart rates.

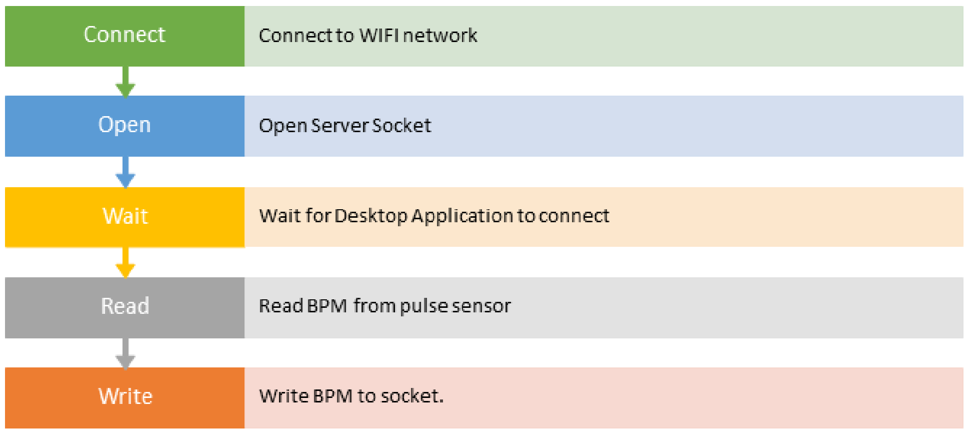


Figure 3: The overall design of the C++ code.

Figure 4 below depicts the flow of data between the data input (pulse sensor) and the UI. The pulse sensor reads in the BPM and sends the information to the Pulse PI desktop application. A user can view his or her real time heart rate data via the desktop application.

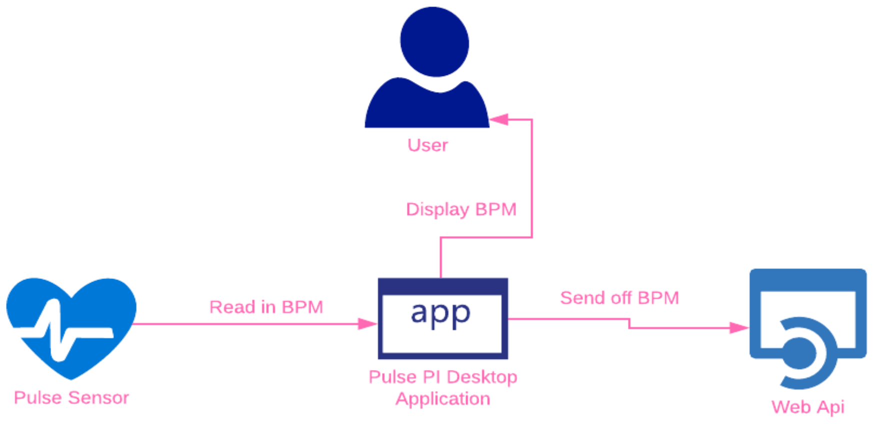


Figure 4: The general design of the desktop application

Pulse PI User Interface Application

The user interface portion of PulsePI consists of a desktop application. Our application needs access to the devices on the local area network in order to find and connect to the pulse sensor. Due to this constraint, the application cannot function as a web app and instead is built using Electron, which uses the Nodejs runtime environment to run JavaScript code. Because we are relying on certain node modules to connect to the pulse sensor, we were unable to use any of the popular front-end frameworks, like Angular, to build the app. Instead, vanilla JavaScript and jQuery are used to handle any and all logic and DOM manipulation. The node modules that we need are used for handling network scanning as well as handling the socket IO connection with the pulse sensor.

In order to connect to the Arduino, the application scans the Local Area Network (LAN) and gets all devices on the same network. Next, it iterates through the devices looking for the Mac address of the Arduino. Once it finds the Mac address, the IP address of the Arduino is recorded, and a client socket is opened. The application can then read in any heart rate data sent by the Arduino.

The user interface itself is designed entirely using twitter Bootstrap. Not only does this allow us to build a very professional looking UI, it also makes the application handle small window and screen sizes due to Bootstrap's mobile first philosophy. It also has the added benefit of cutting down on development time since we do not need to create our own CSS.

The UI layout is very similar to that of other applications built in Electron such as Slack or Discord. Links to the different main functions of the app are presented to the user on a side-nav and links to account and device settings will be on a fixed top-nav. The main content pane of the window will display whichever component the user selects in the side-nav. The options displayed on the side-nav will change depending on if there is a user currently logged in. When there is no account currently logged in, the app users will have the options to log in, check the dashboard, and view their bpm if the device is connected. If a user is logged into an account, they will have access to a profile page, which displays basic account information, as well as a BPM history page where they can view their past BPM records. Within the top-nav, users will have the options to view their notifications, connect to the pulse sensor, and login or logout.

The application must change what is displayed to the user based on two different factors: whether the device is connected and whether an account is logged in. In order to ensure that the application reacts properly to these events, both the Observer and Singleton design patterns are used within the code to represent both the user account and socket connection. With these design patterns properly implemented, all components of the UI that display device or account data can observe their respective class and update accordingly whenever the state of the class changes. Designing the UI in this way greatly reduces the coupling within the code and makes it far more modular and scalable.

Figure 5 depicts our logo. The logo was made by the graphic design tool called Canva. It's an animated gif and a responsive image. Bootstrap was used to make it a responsive image. We called the "img-fluid" class to be able to make it a responsive image. If you look at the bootstrap.css, it's simple to make the responsive image. We just need to set the maximum the width to 100% and height to auto.

Figure 5: The PulsePI logo

Figure 6 shows a sample of the user interface of our desktop application that was discussed above. A user navigates to the various pages in the UI by using the side bar navigation.

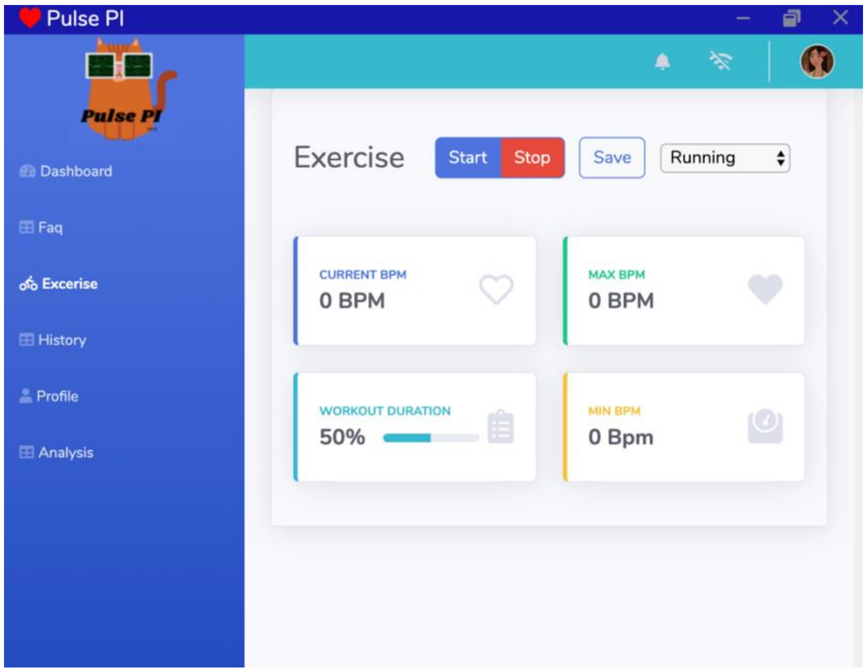


Figure 6: The user interface

PulsePI API

The application programming interface is the link between the user interface and the database. It defines communication protocol used in data exchange between the client facing desktop application and the raw data collected from the Pulse Sensor. Figure 7 depicts the general design of the API. The API consists of three main layers that work together to provide an efficient, decoupled design that is extendible and robust. It is written in C# and uses the ASP.NET Web API framework.

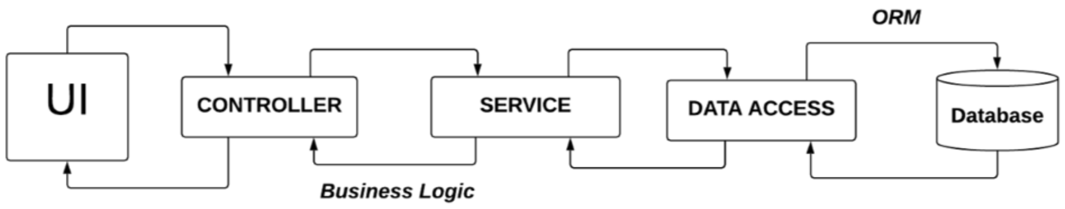


Figure 7: The general design of the API. Depicts the data flow between the various layers of the API

The three layers of the API communicate via data and message contracts. That is, certain models that define what should be received and sent between methods in the various layers are predefined and enforced. This establishes a uniform pattern for data exchange and ensures that incorrect or inappropriate requests and responses do not occur. Full UML diagrams for each of the layers can be found in Appendix A.

The Controller Layer

The controller layer is the client facing layer that exposes the potential actions a client can invoke. It contains the flow control logic, that is, it handles all incoming client browser requests and routes them to the correct service. It also controls the response that is sent back to the client after making a browser request. The basic idea is that each category of operations gets its' own controller class which will contain methods (or "actions") specific to that category of operation. When an HTTP request is generated by the client, it gets routed to the appropriate controller and action via a "route" that is specified in the controller classes.

All of our controller classes extend the abstract controller class that is part of the ASP.NET Web API framework. The abstract controller class provides methods that respond to HTTP requests. The controller class inherits from the abstract class ControllerBase, which represents the base class for all controllers (4). Figure 8 depicts a UML class diagram of one of the controllers that will be included in our API, the AccountController.

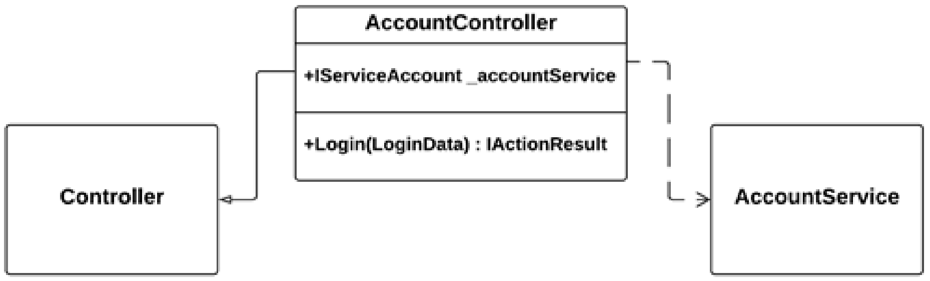

AccountController
+IServiceAccount _accountService
+Login(LoginData) : IActionResult
Controller
AccountService

Figure 8: A UML diagram for the AccountController Class

As seen in Figure 8, the AccountController class is dependent on its corresponding service, the AccountService class. An instance of the service is dependency injected into the controller class. This pattern is followed for all controller classes. Figure 8 also depicts one method in the AccountController class, the Login method (there will of course be many more methods). This method returns an IActionResult, which is essentially a contract that represents the result of an action method (5). All methods in the controller classes will return an IActionResult. This allows us to return a status code along with any message contract that is dictated.

The Service Layer

The service layer is the bridge between the controller layer and the data access layer. The purpose of the service layer is to abstract business logic operations away from the client facing controller layer and/or the data access layer. Having a service layer keeps the controller and data access layers more lightweight (2). Like the controllers, each category of operations has its' own service class. The methods in each separate service class get invoked by the controllers. When invoked, the methods in the service classes call on the data access layer to retrieve whatever data they need. They then perform any necessary business logic on the data received and return the updated data model to the respective controller.

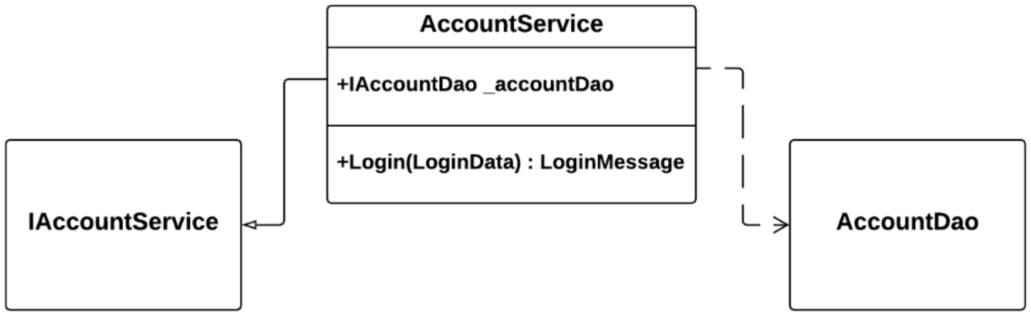


Figure 9: A UML diagram for the AccountService Class

As seen in Figure 9, the AccountService class is dependent on the AccountDao (Account Data Access Object) class. An instance of the AccountDao class is dependency injected into the AccountService class. Each service class is dependency injected with its corresponding data access object class. The AccountService class that can be seen in the figure above also depicts the Login method (again, there will be many more methods). The Login method in the service class, however, returns just a LoginMessage, which is a part of the Message Contracts (the models that dictate what should be returned from any action method). The methods in all the service classes return a message contract to the controller classes.

The Data Access Layer

The data access layer connects the service layer and the data access process. That data access layer is essentially an abstraction of database queries (3). That data access layer performs all operations that extract data from the database. The data access layer implements an ORM (object relational mapping) in order to map database tables into the C# models that exist in our application. Below is a figure depicting one of the data access object classes that is part of the data access layer.

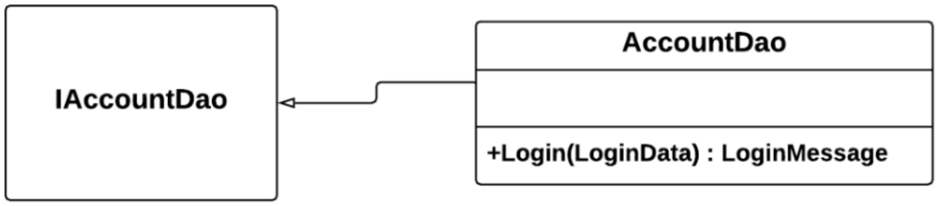


Figure 10: A UML diagram for the AccountDao class

As seen in Figure 10, the AccountDao does not depend on any other classes like the service and controller classes do. The AccountDao only implements the IAccountDao interface. All other data access object classes follow this pattern. Again, the AccountDao class above depicts one method, the Login method. The Login method takes in a LoginData object and returns a LoginMessage object. As mentioned above, these are the objects that dictate what is sent and received between the layers of the API and the client. Each method in a data access object will use LINQ to retrieve the necessary data from the database.

Database

The design and implementation of PulsePI requires the use of a database. We are using a SQL server database that is deployed on the cloud using Azure. This means that the database can conveniently be accessed from anywhere. The data access layer of the API connects to the database using a connection string that contains access credentials. Entity framework is used as the ORM between the C# code in the API and the tables in the database.

Figure 11 depicts the Entity Relationship Diagram for the PulsePI database. The Account table holds all attributes related to an account (username, password, first name, last name, middle name, birth date, email). It has a primary key that is an auto-incremented integer. The Account table relates to every other table in the database.

The Biometric History table holds all attributes relevant to a users' biometrics (height, weight, etc.). This information is necessary when analyzing a users' heart rate data. The date is also included in each Biometric History record, which enables a user to track his/her progress towards certain fitness goals. Each Biometric History record is related to an Account. An Account may have many Biometric History records, while a Biometric History record must have only one Account.

The Device table will hold attributes specific to a device. The device table will enable a user to link a certain device to their PulsePI account. This has the potential to make device connection faster, and the UI more individualized. The Device table is related to the Account table via a many-to-one relationship. An Account does not necessarily have a device associated with it, but it could have many. A Device can exist independently of an Account, but it can have an Account. The Device table holds the foreign key to the Account table.

The Notification table will hold data relevant to any notifications that are generated. This includes a message and whether the particular notification was acknowledged by the user or not. A Notification is related to one Account, but an Account can have many notifications. The Notification record holds a foreign key to the Account that it is related to.

The Heart Rate Record table is perhaps the most important table in the database. It holds all attributes relating to a users' heart rate. After each recorded heart rate session, a Heart Rate Record is created in the database. The record holds the type of activity the user was doing when the rates were recorded, the start time of the activity, the end time of the activity, the minimum BPM recorded, the maximum BPM recorded and the average BPM for the length of the activity. These Heart Rate Records are extracted from the database for analysis. Each Heart Rate Record is related to only one Account, and an Account can have many Heart Rate Records.

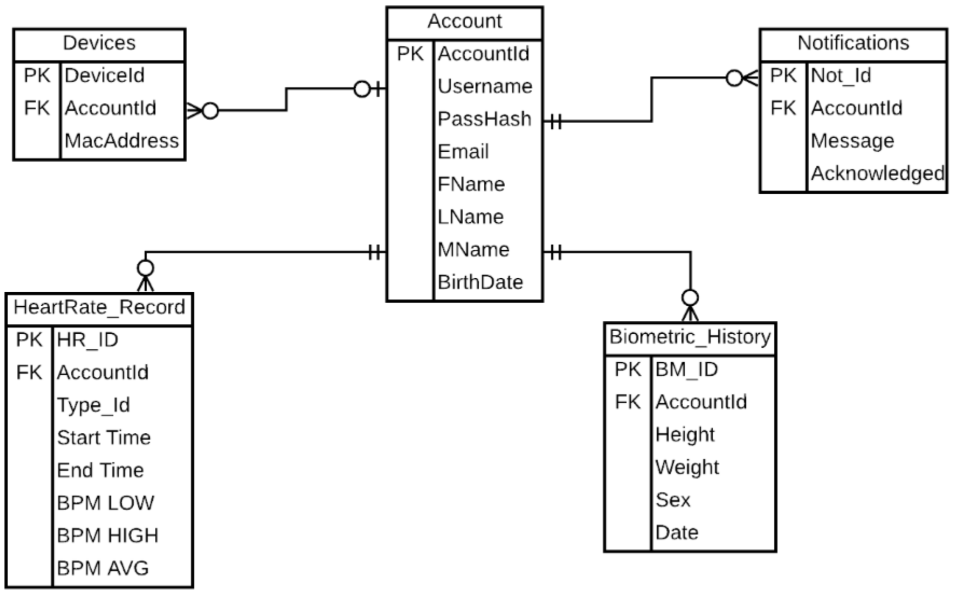


Figure 11: Database ERD

Implementation Specifics

PulsePI includes many components that have been implemented according to the design outlined above. Each of the components involves the frontend and backend working in tandem to produce results. The main components of the PulsePI application include creating an account, logging in, device connection, profile specification, exercise logging, history viewing and heart rate analysis. The implementation specifics of each of these components is explained below.

Create Account

The first component of PulsePI gives users the ability to create an account. Figure 12 depicts the create account screen presented to users. A user begins at this screen and is prompted to enter all necessary information: a username, email address, first name, last name and password. When the user clicks the register account button, the application sends an http request, using AJAX, to the web API with the user entered information. The backend receives the request and routes it to the "CreateAccount" method in the "AccountController" class. When the "CreateAccount" method is invoked, it sends the create account data to the "CreateAccount" method in the "AccountService" class which then sends the data to the "CreateAccount" method in the "AccountDao" class. The "AccountDao" communicates with the database to ensure that the account does not already exist, then creates the account. A successful response message is passed back up to the "AccountController" class and is returned to the UI. Depending on the response the application received from the sever, it will either display an 'Account successfully created' or 'Error Creating Account' message to the user.

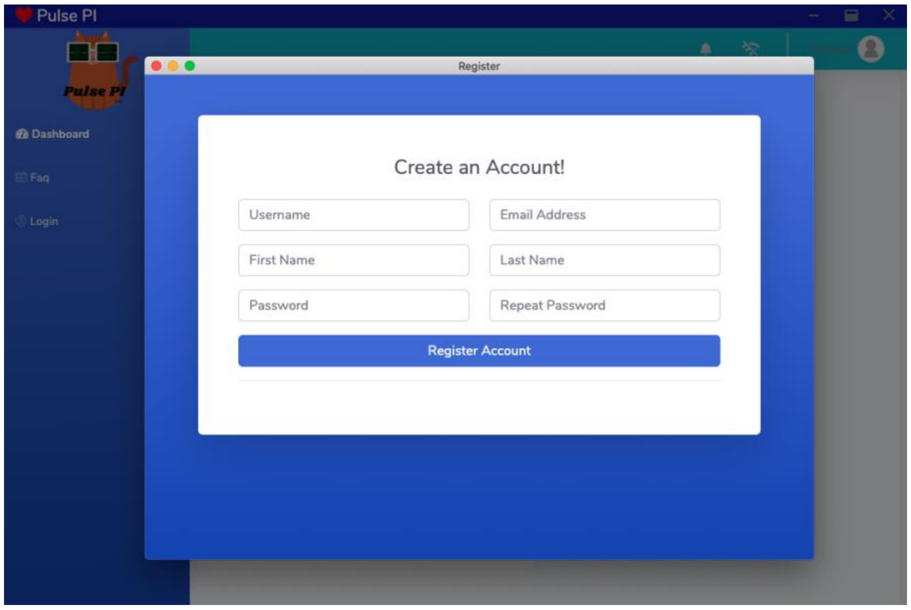


Figure 12: The user-interface create account page

Login

Logging in is the next component of PulsePI. Users can take advantage of accounts they create by logging into those accounts at any time. The login process begins at the login screen, see Figure 13. A user is prompted to enter his/her username and password. When the "Login" button is pushed, the frontend JavaScript sends an http request containing the username and password to the backend. When the backend receives this http request, it is routed to the "Login" method that is part of the "AccountController" class. The controller "Login" method then sends the login data to the service "Login" method which then sends the data to the data access "Login" method. Once the data access "Login" method is invoked, the C# code ensures the account exists and that the password stored matches the password entered. If these things are found to be true, a successful response message is passed back up through the API layers and sent to the frontend. The login response message contains all data relevant to an account: username, first name, last name, middle name, birth data, avatar url and email. The frontend needs to be sent this information to display personalized data to users after logging in.

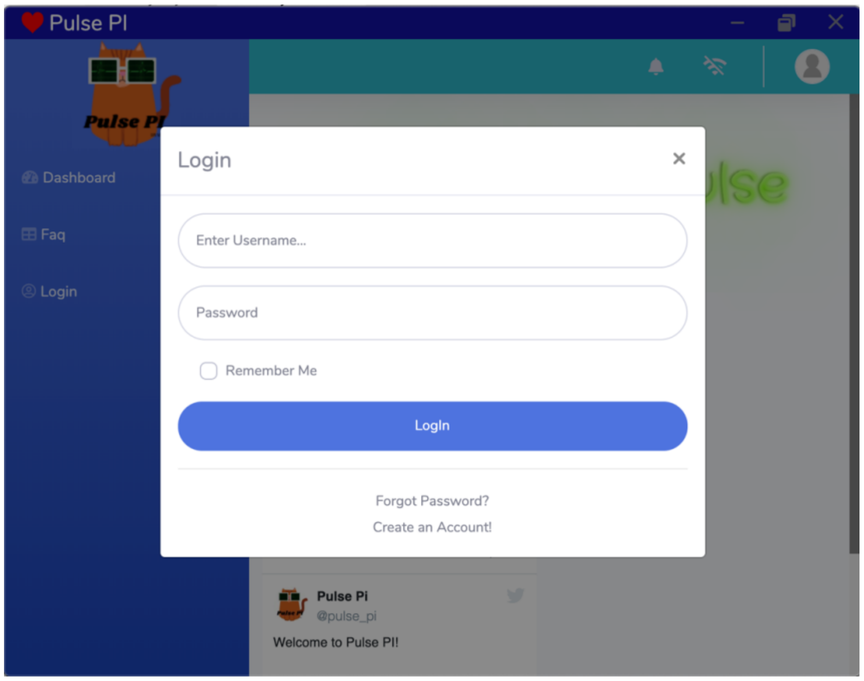
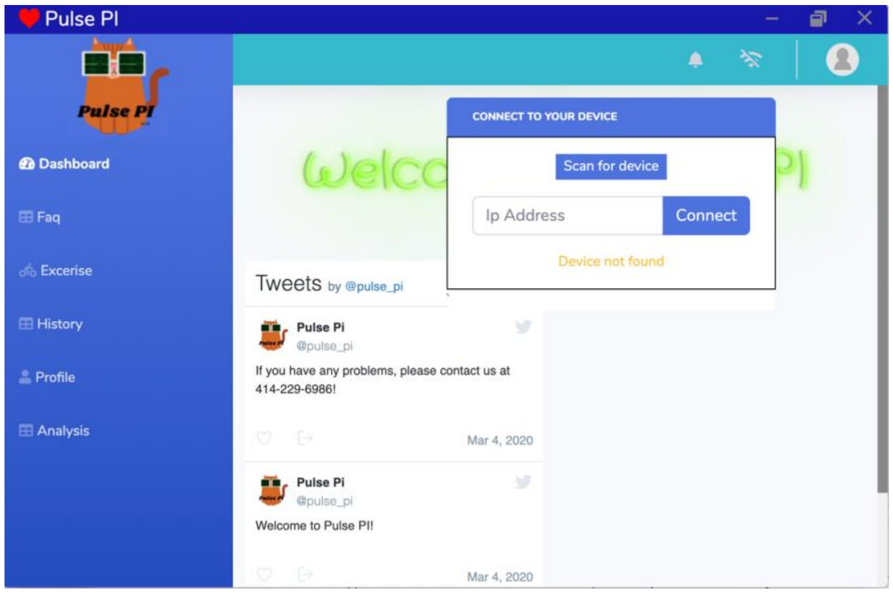


Figure 13: The login screen

Device Connection

The application and the heart rate sensor communicate with each other using socket IO with the heart rate sensor acting as the server and the application acting as the client. The application uses NodeJS' 'net' package for all client socket operations. The application can connect to the heart rate sensor in one of two ways. The first way is to enter in the local IP address of the heart rate monitor. The other way is to 'scan' for the device on the network and then connect to it automatically. The scan works by taking advantage of another NodeJS package called 'local-devices'. What this package does is execute 'arp -a' into the shell which will gather IP and mac addresses of other devices on the local network. Once the devices are found they are returned to the application which then parses through them looking for the mac address of the heart rate sensor. If it is found on the network the application will then connect to it. The scan usually fails in large networks, like UWMs, which is why the option to manually enter the Ip address is there.



Device not found

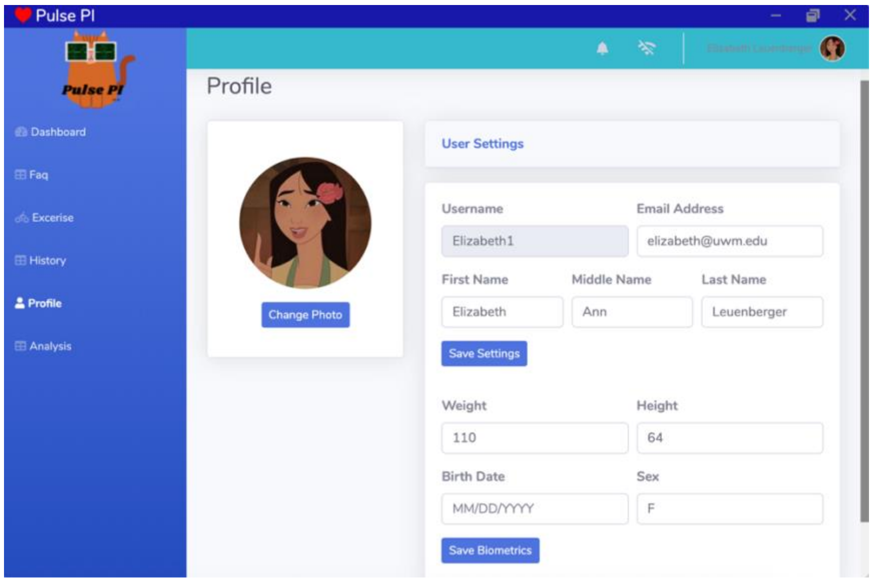
Pulse Pi
@pulse_pi
If you have any problems, please contact us at 414-229-6986!
Mar 4, 2020

Pulse Pi
@pulse_pi
Welcome to Pulse PI!
Mar 4, 2020

Figure 14: The Device Connection modal

Profile Specification

Users can change and personalize their profile information via the "Profile" screen. This screen allows users to both view and edit account information as well as their biometric data. This is also were users can pick their favorite Disney Princess to use as their profile picture. When this page is loaded the application will make a call to the Web API in order to grab user's biometric data. When the user makes any changes to either their biometric data or their account settings, they can save them to the database by clicking the respective save button. When the button is clicked the application will gather the information from the form and send it off to the Web API where it will get saved to the database.



Username	Elizabeth1
Email Address	elizabeth@uwm.edu
First Name	Elizabeth
Middle Name	Ann
Last Name	Leuenberger
Weight	110
Height	64
Birth Date	MM/DD/YYYY
Sex	F

Figure 15: The profile page

Exercise

Figure 16 depicts the exercise page. This page allows users to record exercise/resting sessions. Once a user connects his/her device, heart rates can be recorded. When a user is ready to start the session, the "start" button is pushed. This invokes a JavaScript method that reads in all heart rates sent from the pulse sensor. The method reads in the heart rate, and notes if it is either the lowest heart rate recorded, or the highest heart rate recorded. The previous high/low value is replaced if either of those cases are true. The method also keeps a count of all heart rates recorded and a adds each heart rate to a running sum so that the average bpm can be calculated. The observer pattern is used in the JavaScript code to dynamically update the maximum bpm, minimum bpm and current bpm displayed on the screen. When any of these values changes, the new value is displayed. The page also includes a workout duration, which is a stopwatch that will keep track of how long the user has been exercising. When a user is finished with the session, the "stop" button in pushed. At this point further heart rates are not recorded and the average bpm for the session is calculated. If a user wishes to save the session, he/she can push the "Save" button. When the "Save" button is pushed, the username, start time, end time, low bpm, high bpm, average bpm and exercise type is included as payload to an http request to the backend. This request gets routed to the "RecordHeartRateRecord" endpoint, which saves the received heart rate data to the database.

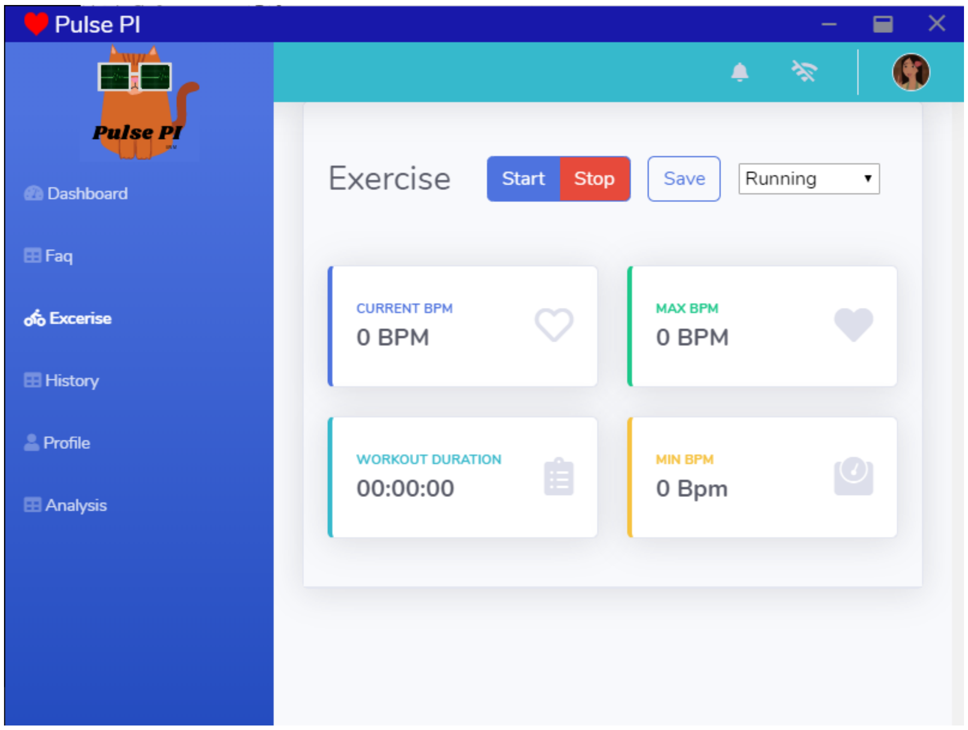
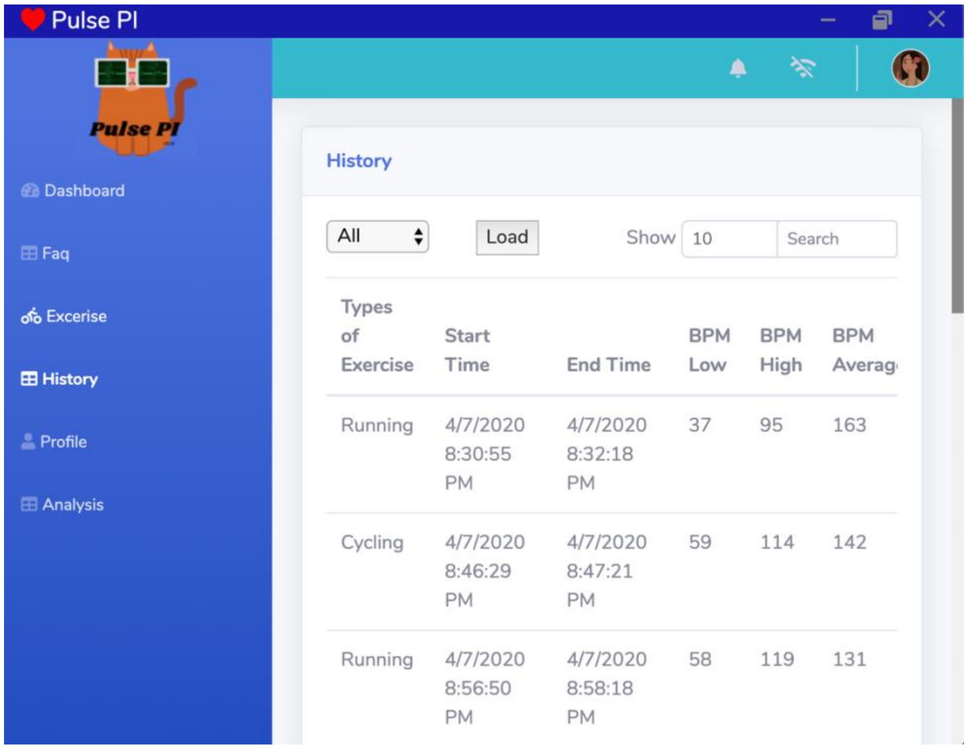


Figure 16: The Exercise page

History

This is the history page, where users can view their heart rate history in table form. A user can select from three options in the upper left hand drop down - "All", "Exercise" or "Resting". When a user pushes "Load", the frontend JavaScript uses AJAX to send a request to one of three endpoints depending on the options selected from the dropdown menu. The username of the user who is logged in is sent as payload for each request. If a user has selected the “All” option, the “GetAllHeartRateHistory” endpoint is invoked. This endpoint will retrieve all the users heart rate records that are stored in the database. If a user has selected the “Exercise” option, the “GetExerciseHeartRateHistory” endpoint is invoked and only a user’s exercise records are returned from the backend. Lastly, if a user has selected the “Resting” option, the “GetRestingHeartRateHistory” endpoint is invoked and a user’s resting heart rate records are returned. Upon receiving a response from the backend, the frontend populates the table seen below in Figure 17. The table changes dynamically after a user pushed “Load” based on what is selected in the dropdown menu.



Types of Exercise	Start Time	End Time	BPM Low	BPM High	BPM Averag
Running	4/7/2020 8:30:55 PM	4/7/2020 8:32:18 PM	37	95	163
Cycling	4/7/2020 8:46:29 PM	4/7/2020 8:47:21 PM	59	114	142
Running	4/7/2020 8:56:50 PM	4/7/2020 8:58:18 PM	58	119	131

Figure 17: The Exercise History Page

Analysis

The analysis of heart rate data recorded during exercise can be viewed by users via the Analysis page. There are two models that are displayed on the Analysis page, the Biometrics modal and the Analysis modal, seen in Figure 18 and Figure 19 respectively.

The Biometrics modal reveals to users certain calculated values based on the biometric data entered on their profile page. The user does not need to take any action on the Analysis page to see this data, it is loaded upon navigation to the page. When a user clicks on the “Analysis” link in the left-hand navigation bar, the frontend JavaScript sends two http requests to the backend to retrieve the data that populates this modal. In this case, the frontend sends only the username as the payload to the backend for both requests.

The first request routes to the “GetHRBounds” method in the “BiometricController” class. This method passes the username received from the frontend down through the layers. In the “GetHRBounds” service layer method, a request is made to the data access to get the users most recent biometric data record. The "GetHRBounds" method in the data access layer uses LINQ to query all biometric data records for the user. It then orders the resulting list by date and returns the most recent record to the service layer.

When the "GetHRBounds" method in the service layer receives the biometric data object from the data access layer, the values that are needed to populate the fields on the frontend are calculated. A resource from the Mayo Clinic was used as a reference on how to determine these values (7). The users maximum heart rate, target heart rate and heart rate reserve are calculated via the formulas listed below:

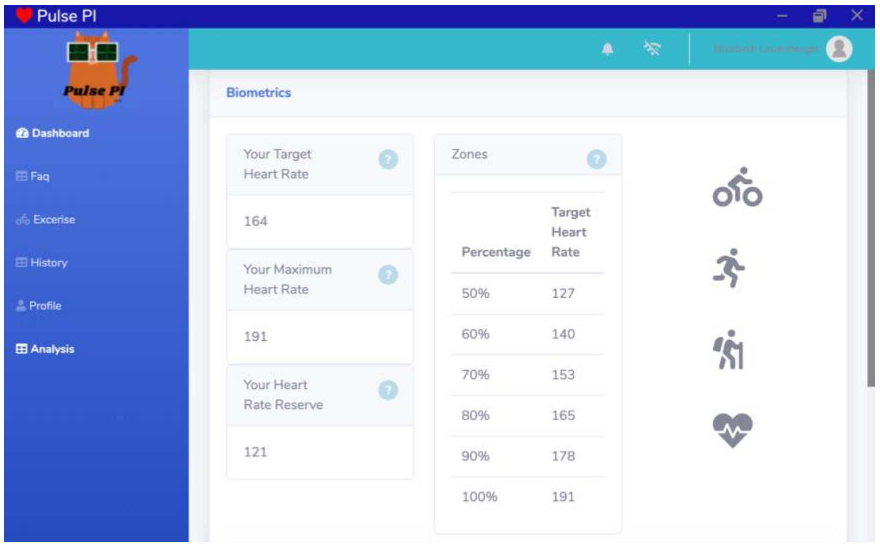
$$\textit{Max heart rate} = 220 - \textit{age}$$
$$\textit{Heart rate reserve} = \textit{max heart rate} - \textit{resting heart rate}$$
$$\textit{Target heart rate} = ((\textit{heart rate reserve} * 0.70) + (\textit{heart rate reserve} * 0.85)/2)$$

After all these values are calculated, they are returned to the "BiometricController" wrapped in a "GetHRBoundsMsg" object and are sent back to the frontend. The frontend receives the data and populates the "Your Target Heart Rate", "Your Maximum Heart Rate" and "Your Heart Rate Reserve" fields seen below in Figure 18.

The second http request is to populate the "Zones" table and routes to the "GetRanges" method in the "BiometricController" class. This method again passes the username received from the frontend down through the API layers. In the "GetRanges" service layer method, the most recent biometric record for the user is again obtained from the "BiometricDao". Upon receiving the biometric data object, the service layer calculates the users maximum heart rate. The target heart rates seen in the chart are then calculated by the equation:

$$\textit{Target Heart rate} = \textit{Maximum heart rate} * \textit{percentage}$$

The Zones chart is useful in determining optimal values for heart rates during exercise. Vigorous exercise is about 70% of one's maximum heart rate (7).



Biometrics

Your Target Heart Rate
164

Your Maximum Heart Rate
191

Your Heart Rate Reserve
121

Zones

Percentage	Target Heart Rate
50%	127
60%	140
70%	153
80%	165
90%	178
100%	191

Figure 18: The Biometrics modal

The Analysis model gives users the option to view trends based on past heart rate data collected during exercise or rest. Unlike the Biometrics modal, the Analysis modal does not load data until a user requests it. A user can choose from two different charts, an "Exercise Efficiency" chart or a "Resting Heart Rate Trends" chart. After selecting the type of chart, a user must push the "Load" button in order to receive data. Chart.js was used to display both charts.

The "Exercise Efficiency" chart displays what percentage of one's target heart rate was achieved during a particular workout. The values displayed in this chart should be compared with the values in the "Zones" table for clarification. When the "Load" button is pushed, the frontend JavaScript sends an http request to the backend. This request gets routed to the "GetIntensities" method in the "BiometricController". This method passes the username sent from the frontend through the API layers. In the "GetIntensities" service layer method, a user's exercise history and most recent biometric record are requested from the data access layer. After receiving this information from the "BiometricDao", the service layer method does many calculations to determine the percentages displayed in the chart.

First, a user's personal heart rate values for each percentage is calculated. There is a private helper method in the service layer to do this. This method calculates a user's maximum heart rate, then multiplies it by each percentage decimal to determine the target heart rate for that percentage. The percentages as well as their corresponding heart rates are returned in a list to the calling method. Next, it must be determined which percentage interval each workout falls into. This is done in another private method. For each exercise record, the average bpm value is compared with the data contained in the list containing the user's various intensities. The list is iterated until the closest heart rate to the average heart rate is found. The associated percentage is noted. The date of workout and associated percentage for each exercise record is then returned to the frontend.

Upon receiving this data, the frontend uses Chart.js to build the "Exercise Efficiency" chart. It uses the dates returned from the backend as the x-axis and the percentages values as the y-axis.

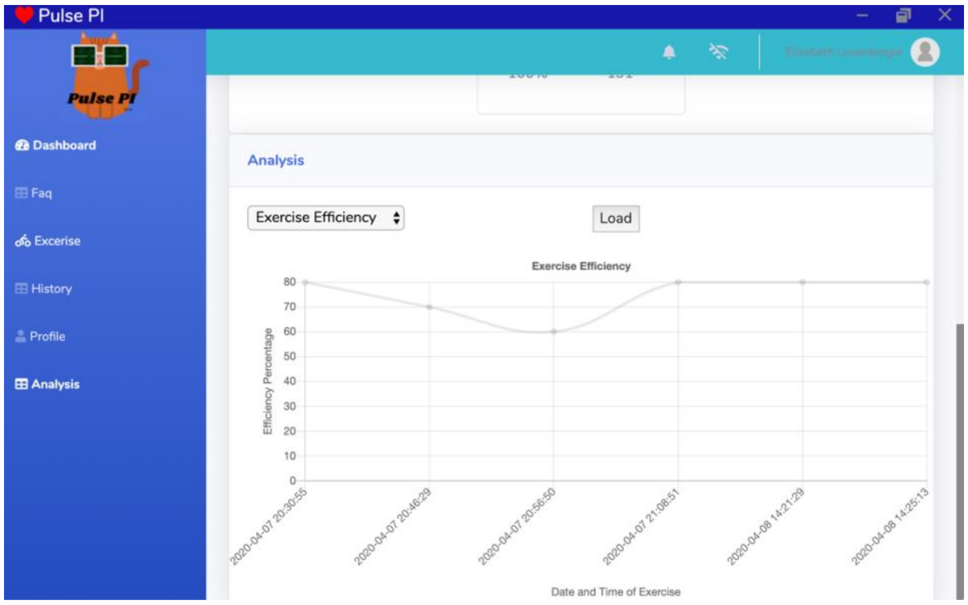


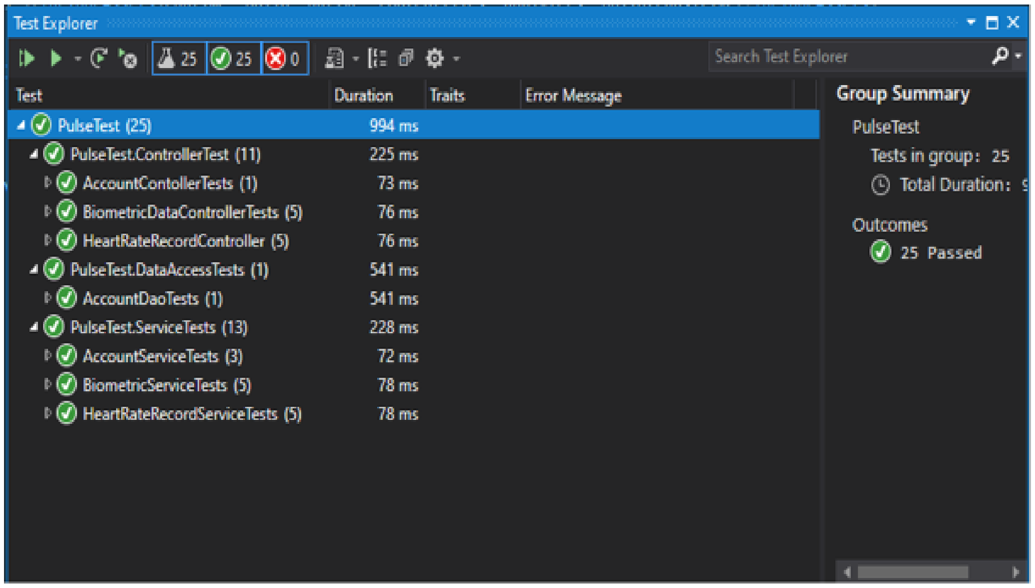
Figure 19: The Heart Rate Analysis modal

The second chart, “Resting Heart Rate Trends” is simpler than the “Exercise Efficiency” chart. When a user selects the “Resting Heart Rate Trends” option and clicks “Load” the frontend JavaScript sends an http request to the backend with the username as payload. When the request is received by the backend, it is routed to the “GetRestingHeartRateHistory” method in the “HeartRateRecord” controller. This method passes the username down through the API layers. The service layer method simply requests the user’s resting heart rate history from the data access layer. The records returned from the data access layer to the service layer are compiled in a list and sent back to the frontend via the controller layer. Upon receiving this data, the frontend uses Chart.js to generate a chart. It uses the time of each heart rate recording as the x-axis and the average bpm for each recording as the y-axis.

Testing

Unit Tests

We used MOQ with XUnit for the unit testing. We decided to use the Moq framework because it made the unit testing easier. For example, suppose that we need to use a database for one class test. In this case, inserting and removing information directly into the database can be a significant burden. Using the Moq framework made testing easier by mimicking the behavior of the database as well as certain methods that require making changes to the database. In order to achieve this, however, it was necessary to design a program that operates only through interfaces. As such, the controller and service unit tests were completed using Moq and XUnit, while the data access unit tests were completed using an in-memory database. From Figure 20, we can see that all 25 tests passed.

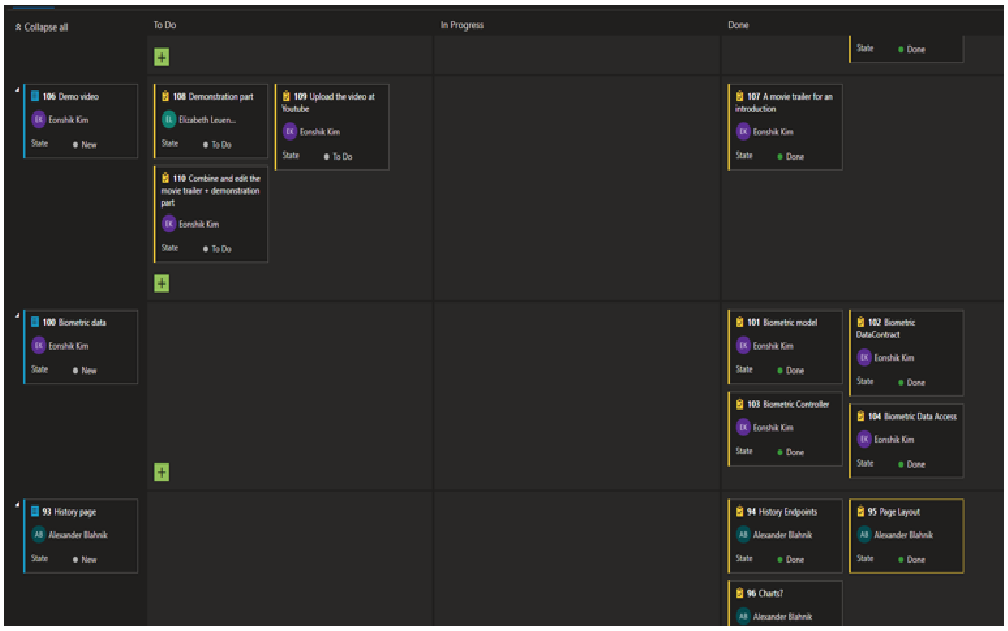


```
Test Explorer
25  25  0                                              Search Test Explorer
Test                                  Duration  Traits  Error Message      Group Summary
PulseTest (25)                        994 ms                                 PulseTest
  PulseTest.ControllerTest (11)       225 ms                                   Tests in group: 25
    AccountContollerTests (1)          73 ms                                   Total Duration: 9
    BiometricDataControllerTests (5)   76 ms                                 Outcomes
    HeartRateRecordController (5)      76 ms                                   25 Passed
  PulseTest.DataAccessTests (1)       541 ms
    AccountDaoTests (1)               541 ms
  PulseTest.ServiceTests (13)         228 ms
    AccountServiceTests (3)            72 ms
    BiometricServiceTests (5)          78 ms
    HeartRateRecordServiceTests (5)    78 ms
```

Figure 20: Test Results

Azure DevOps

From Azure DevOps, we used the Azure boards and Azure pipeline. Azure boards are useful for agile planning and work item tracking. Since we practiced Scrum Agile methodology, we decided to divide the tasks into sprint 1 and sprint 2. From Figure 21, it is clear that the tasks are divided into 3 separated conditions. It was easier to manage our tasks this way because we could see what the most pressing to-do items were, what items were in progress, and what items were completed. This also helped us to avoid duplicated tasks. From Figure 22, we also used the Azure pipeline in Azure DevOps to Automate our builds and deployments.



State Done

106 Demo video
Eonshik Kim
State New

108 Demonstration part
Elizabeth Leuen...
State To Do

109 Upload the video at Youtube
Eonshik Kim
State To Do

110 Combine and edit the movie trailer + demonstration part
Eonshik Kim
State To Do

107 A movie trailer for an introduction
Eonshik Kim
State Done

100 Biometric data
Eonshik Kim
State New

101 Biometric model
Eonshik Kim
State Done

102 Biometric DataContract
Eonshik Kim
State Done

103 Biometric Controller
Eonshik Kim
State Done

104 Biometric Data Access
Eonshik Kim
State Done

93 History page
Alexander Blahnik
State New

94 History Endpoints
Alexander Blahnik
State Done

95 Page Layout
Alexander Blahnik
State Done

96 Charts?
Alexander Blahnik

Figure 21: Azure boards

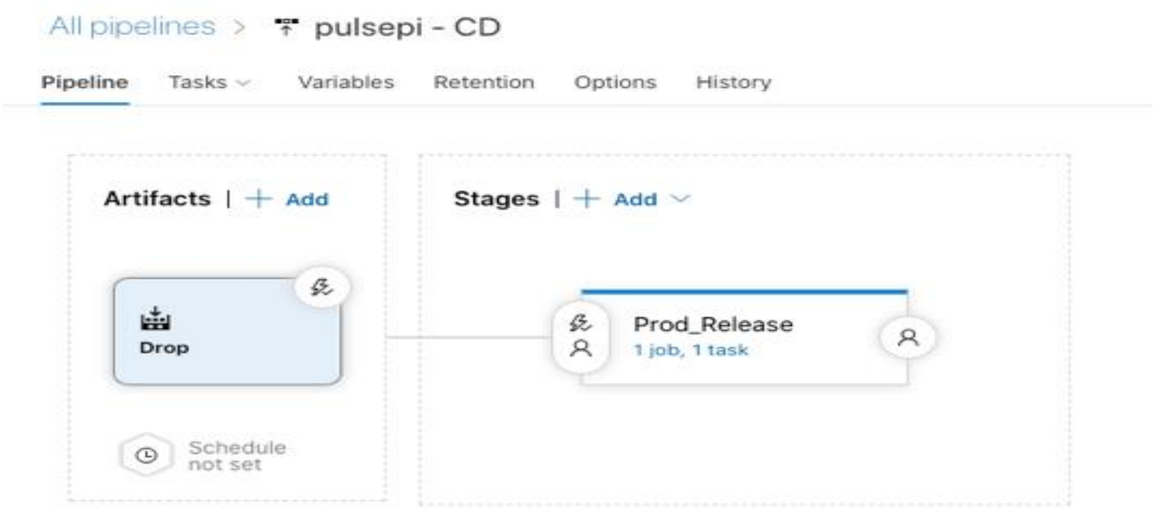


Figure 22: Azure Pipeline

Leftover Work

Notifications

The team wished to implement the ability to send users notifications. This functionality has not been completed, however. The notifications would include things like inspirational quotes to instill our users with motivation and up to date news about health-related issues.

Linking devices to profiles

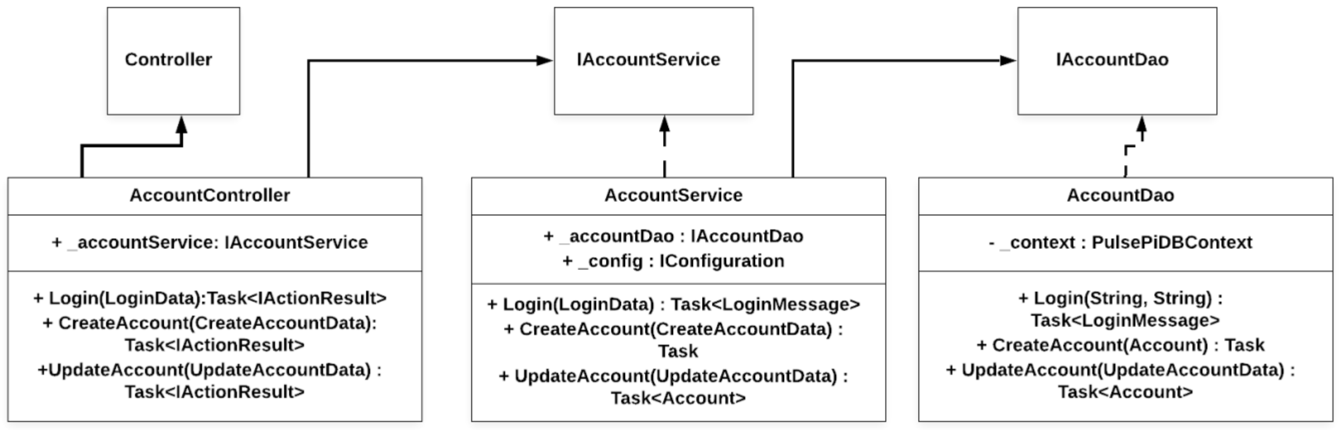
The software team discussed making a device specific to a certain user's profile, that is, a user would be able to “register” his/her device to his/her profile. This would have included storing the mac address of the device as well as a device name in the database. This functionality was not implemented.

Conclusion

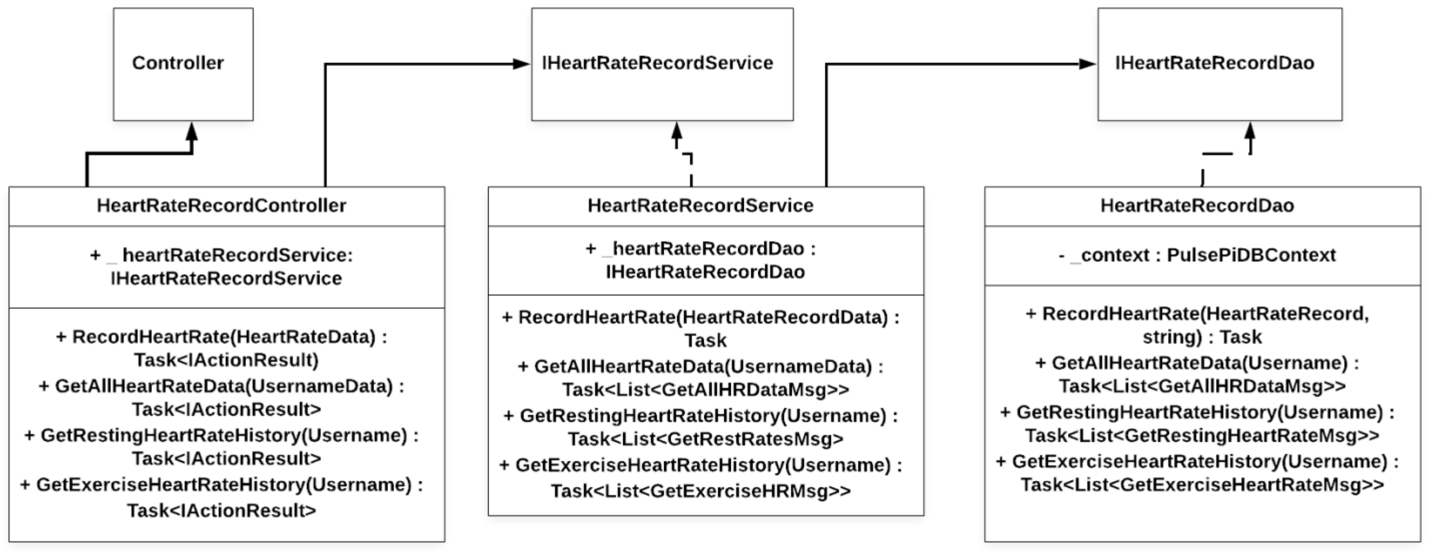
PulsePI has been developed into a robust application that accomplishes what it set out to do. PulsePI can store basic user information, which allows clients to create accounts and login, as well as customize their profiles with personal biometric data and an avatar of their choice. PulsePI enables users to connect their pulse sensor devices and monitor their heart rates during exercise and rest. It lets users specify a type of activity and save heart rate data from that activity. PulsePI uses a client's personal biometric data to calculate heart rate values to assist in their quest for a more efficient cardiovascular system. It generates charts that communicate helpful trends of heart rates during exercise and rest that give user’s important insight into their heart’s efficiency. PulsePI is no doubt the next great tool in the realm of cardiovascular monitoring technology. It has been developed with the future in mind using the most up to data technologies and best practices. PulsePI will help improve the health and well-being of those who use it.

Appendix A : API UML Diagrams

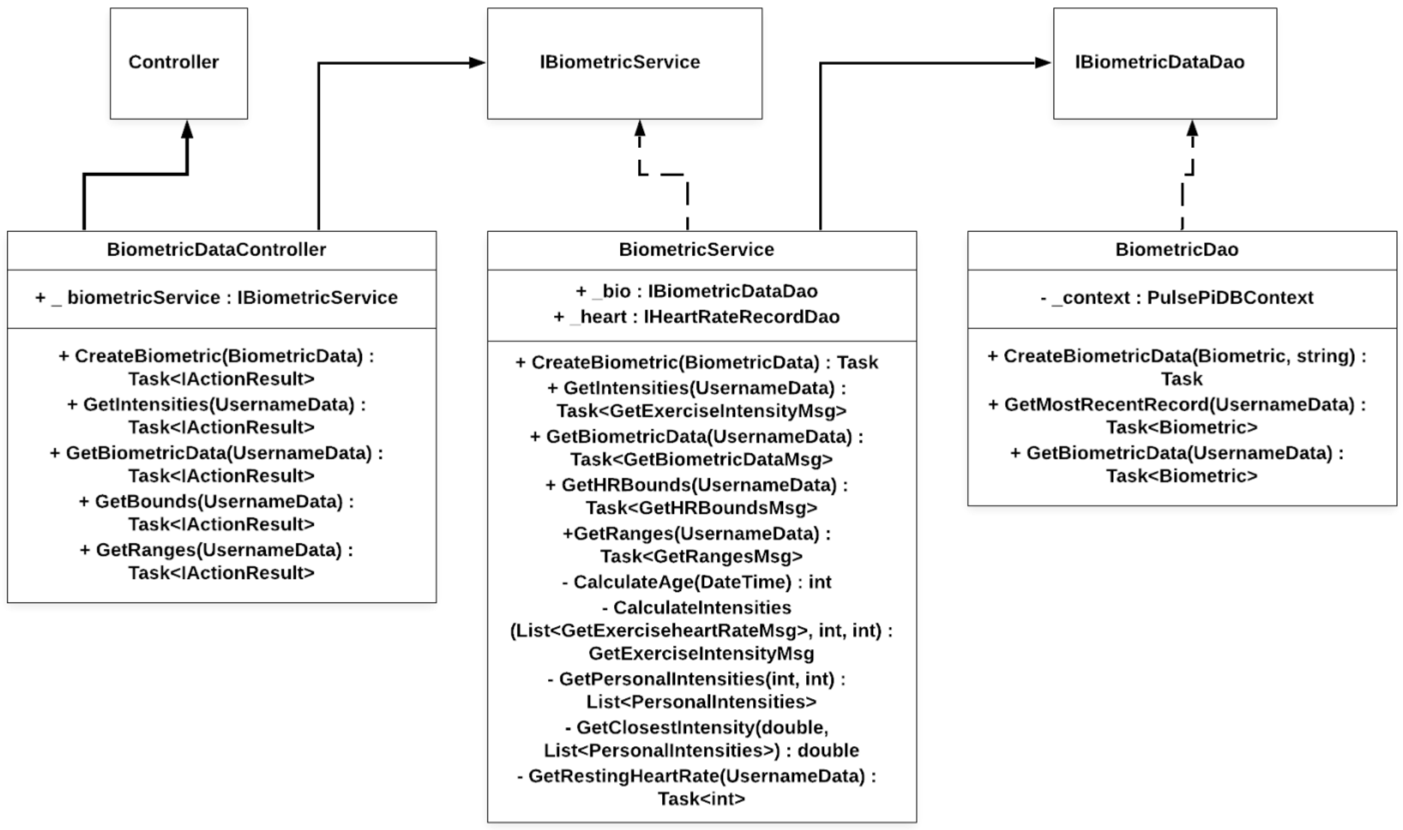
Account UML Diagram


Controller
IAccountService
IAccountDao
AccountController
+ _accountService: IAccountService
+ Login(LoginData):Task<IActionResult>
+ CreateAccount(CreateAccountData): Task<IActionResult>
+UpdateAccount(UpdateAccountData) : Task<IActionResult>
AccountService
+ _accountDao : IAccountDao
+ _config : IConfiguration
+ Login(LoginData) : Task<LoginMessage>
+ CreateAccount(CreateAccountData) : Task
+ UpdateAccount(UpdateAccountData) : Task<Account>
AccountDao
- _context : PulsePiDBContext
+ Login(String, String) : Task<LoginMessage>
+ CreateAccount(Account) : Task
+ UpdateAccount(UpdateAccountData) : Task<Account>

Heart Rate UML Diagram


Controller
IHeartRateRecordService
IHeartRateRecordDao
HeartRateRecordController
+ _ heartRateRecordService: IHeartRateRecordService
+ RecordHeartRate(HeartRateData) : Task<IActionResult)
+ GetAllHeartRateData(UsernameData) : Task<IActionResult>
+ GetRestingHeartRateHistory(Username) : Task<IActionResult>
+ GetExerciseHeartRateHistory(Username) : Task<IActionResult>
HeartRateRecordService
+ _heartRateRecordDao : IHeartRateRecordDao
+ RecordHeartRate(HeartRateRecordData) : Task
+ GetAllHeartRateData(UsernameData) : Task<List<GetAllHRDataMsg>>
+ GetRestingHeartRateHistory(Username) : Task<List<GetRestRatesMsg>
+ GetExerciseHeartRateHistory(Username) : Task<List<GetExerciseHRMsg>>
HeartRateRecordDao
- _context : PulsePiDBContext
+ RecordHeartRate(HeartRateRecord, string) : Task
+ GetAllHeartRateData(Username) : Task<List<GetAllHRDataMsg>>
+ GetRestingHeartRateHistory(Username) : Task<List<GetRestingHeartRateMsg>>
+ GetExerciseHeartRateHistory(Username) : Task<List<GetExerciseHeartRateMsg>>

Biometric UML Diagram


Controller
IBiometricService
IBiometricDataDao
BiometricDataController
+ _ biometricService : IBiometricService
+ CreateBiometric(BiometricData) : Task<IActionResult>
+ GetIntensities(UsernameData) : Task<IActionResult>
+ GetBiometricData(UsernameData) : Task<IActionResult>
+ GetBounds(UsernameData) : Task<IActionResult>
+ GetRanges(UsernameData) : Task<IActionResult>
BiometricService
+ _bio : IBiometricDataDao
+ _heart : IHeartRateRecordDao
+ CreateBiometric(BiometricData) : Task
+ GetIntensities(UsernameData) : Task<GetExerciseIntensityMsg>
+ GetBiometricData(UsernameData) : Task<GetBiometricDataMsg>
+ GetHRBounds(UsernameData) : Task<GetHRBoundsMsg>
+GetRanges(UsernameData) : Task<GetRangesMsg>
- CalculateAge(DateTime) : int
- CalculateIntensities (List<GetExerciseheartRateMsg>, int, int) : GetExerciseIntensityMsg
- GetPersonalIntensities(int, int) : List<PersonalIntensities>
- GetClosestIntensity(double, List<PersonalIntensities>) : double
- GetRestingHeartRate(UsernameData) : Task<int>
BiometricDao
- _context : PulsePiDBContext
+ CreateBiometricData(Biometric, string) : Task
+ GetMostRecentRecord(UsernameData) : Task<Biometric>
+ GetBiometricData(UsernameData) : Task<Biometric>

Appendix B : Links to Repositories

API Repo: https://github.com/mblahnik/PulsePI

Desktop Application Repo: https://github.com/mblahnik/PulsePI_APP

Arduino/Heart Rate Monitor: https://github.com/elileuenberger/PulseSensor_BPM

Appendix C : Member Timelines

Group:

Before any major individual was done our group worked together to get the heart rate monitor to work. This took a bit of time due to compatibility issue with the Arduino and the pulse sensor as well as the difficulty of finding help online. It took a lot of group effort, but we were eventually able to get it to read in a bpm value and write it to a socket. After the pulse sensor was set up, we worked on starting up each part of the project. This included making GitHub repos for each component and acquiring a database from Azure. Once we had all the boilerplate code in for each project we went ahead and started to work on different parts of the project individually.

Alex Blahnik

The first thing that I worked on was coming up with a way to get the desktop application to connect to the Arduino. This involved experimenting with different node modules in order to find one that I could easily integrate into our project. Once I was able to find a good client socket module, I created the connect to device dropdown to the UI and added all the logic. At this point in the project there was enough of the Web API built to justify creating a deployment pipeline on Azure. Once we had the web API deployed on the cloud, I created pop up dialogs for logging in and creating accounts and added them to the desktop application along with all the necessary logic. By this point Eonshik had created the layout for most of the pages for the desktop application. I went ahead and added logic to the profile page, the exercise page, as well as the side nav. Also, at some point early on I created the awesome custom title bar.

Elizabeth Leuenberger

Throughout the project I mostly worked on the Web API. Initially, I worked collaboratively with other group members to determine the technologies and design that would make up the API. Once the design was established, I created the various classes and infrastructure to support our design. As Alex and Eonshik progressed with frontend development, I worked in tandem with them to code the necessary endpoints and infrastructure on the backend. This included writing methods for each endpoint in the controller, service and data access layers. I contributed to the frontend work when it came time to complete the Analysis Page – I completed both the frontend and backend portions for the Analysis Page. This required a fair amount of research into Chart.js and heart rate analysis equations, etc. Additionally, I helped with the development of some tests using the moq framework. Finally, I contributed greatly to all presentations and reports that were due throughout the semester.

Eonshik Kim

At the beginning of the project, I worked collaboratively with my team in order to conduct research to most suitable device to connect with the heart rate sensor. After some research, I decided to buy an Arduino Rev 2. I played around with its built-in functions in order to get familiar with the device. After Alex bought the heart rate sensor, I assembled the components of the sensor and connected it to Arduino and tested the sensor with Arduino to see whether both were working. At the midway point of our project, I transitioned towards mostly working on the page layouts and design. I created the layout and designed most of the pages using HTML/CSS and Bootstrap. Alex and I worked together to design the side and top navbars. I added the Twitter API on the dashboard page. I also created our animated logo using Canva and placed it inside of the side navbar. On the exercise page, I created an animated cat for the loading screen and programmed it using JavaScript to work only when the user clicked the start button. Towards the end of the project, I helped primarily with backend development. Elizabeth and I implemented a Biometric data endpoint and wrote unit testing using Moq and XUnit. Finally, I made a movie trailer for our demonstration video using iMovie and edited it alongside our demonstration using Adobe Premiere.

References

1. https://docs.microsoft.com/en-us/dotnet/api/system.web.mvc.controller?view=aspnet-mvc-5.2
2. https://codereview.stackexchange.com/questions/33109/repository-service-design-pattern
3. Zhelev, Dimitar. *C# Layer Interaction.* https://mentormate.com/blog/service-and-repository-layer-interaction-in-c/
4. Microsoft Documentation. *Controller Class*. https://docs.microsoft.com/en-us/dotnet/api/system.web.mvc.controller?view=aspnet-mvc-5.2
5. Jones, Matthew. *IActionResult and ActionResult.* https://exceptionnotfound.net/asp-net-core-demystified-action-results/
6. "How to Build a Responsive Bootstrap Website." SitePoint, 4 May 2018, www.sitepoint.com/build-responsive-bootstrap-website/
7. https://www.mayoclinic.org/healthy-lifestyle/fitness/in-depth/exercise-intensity/art-20046887
8. Hein, Buster. "Study Finds Apple Watch Is Most Accurate Wearable at Measuring Heart Rate." Cult of Mac, 26 May 2017, www.cultofmac.com/483234/study-finds-apple-watch-accurate-wearable-measuring-heart-rate/.
9. Kushwaha, Dinesh. "Moq Mocking Framework With XUnit.net Unit Test In C#." *C# Corner*, www.c-sharpcorner.com/article/moq-mocking-framework-with-xunit-net-testing-fr/.
10. "Azure Pipelines." Microsoft Azure, azure.microsoft.com/en-us/services/devops/pipelines/.