HUNK3

FYP Proposal

Chatbots as Personal Caregivers for Diabetics

by

Shivang GUPTA and Allen Kin Wai FOK

HUNK3

Advised by

Prof. Sung KIM

Submitted in partial fulfillment

of the requirements for COMP 4981

in the

Department of Computer Science

The Hong Kong University of Science and Technology

2016-2017

Date of submission: April 20, 2017

Abstract

From Siri to Google Home, conversational interfaces are gaining popularity day by day. The motivation behind this project was to test the potential of solving complex problems with existing natural language processing and messaging technology. Bo, a personal diabetes management chatbot was created to demonstrate this potential. Users can chat with Bo like any other friend on Facebook and Bo helps users manage blood glucose values, meal data, and even reminds them to take their medicines on time. Through the creation of Bo we determined that current technology is capable of creating simple single-purpose bots, but vast improvement is needed both in natural language understanding and messaging platform interfaces before conversational interfaces can replace traditional graphical user interfaces.

Table of Contents

1 Introduction

1.1 Overview

Chatter Bots or ‘Chatbots’ are a type of conversational interface for humans to interact with computers through natural language. In essence, chatbots are computer programs that can simulate human conversation. Chatbots have been around since the 1960s when researchers at the Massachusetts Institute of Technology presented ELIZA, the first chatbot, which primarily relied on identifying keywords and replying to them with pre-set responses. [1] ELIZA was followed by A.L.I.C.E., a chatbot that was first seen around 1995 and had the additional feature of using care based reasoning (CBR) to better understand the context of a user message. A.L.I.C.E. also included tools to normalize the user input through simple methods like spell checking. [2]

While the idea of a conversational interface has been around for decades, chatbots have witnessed a recent rise in popularity due to advances in computing power and artificial intelligence. Using advanced Natural Language Processing (NLP) and Machine Learning (ML), chatbots are able to understand context from the inherently ambiguous language used by humans. They are also more capable of varying their replies and the language of present day chatbots is far more similar to humans than past attempts like A.L.I.C.E and ELIZA.

The initial motivation of this project was to test the capabilities of these new chatbot platforms and existing artificial intelligence technology through the implementation of one complete use case. Diabetes was chosen for this purpose.

Diabetes is a chronic disease that affects nearly one-tenth of the world’s adult population. [12] In order to manage the symptoms and prevent life threatening complications, patients must monitor their daily blood glucose levels to keep them within a target range, while maintaining a healthy diet and in many cases taking medication. [13]

Traditionally some diabetics used diaries to keep tabs on their daily values, however, with advances in digital technology many mobile applications and online services have gained popularity for the same task. Generally, these services have shown that they can better help patients manage their health over time. However, they are yet to be widely adopted and have been found to be difficult to use due to complex user interfaces and the lack of personalized feedback. [3]

This projects aims to create a chatbot that can act as a virtual caregiver to diabetics by helping them keep track of their blood glucose values, recording meals, and reminding them to take their medication and have their meals on time. Virtual assistants like Apple's Siri have become widely adopted into daily lives and in this project we adopt similar technology to create a tool specifically for diabetics. Our main motivation was the belief that by acting like a human, a chatbot can provide the personalized touch that regular applications lack.

Given the nascent nature of modern chatbot technology there are few examples of complex chatbots and in this project we hope to test the true capabilities of current chatbot technology.

1.2 Objectives

The primary objective of this project is to implement an easily available and easy to use chatbot that can make diabetes management easier. Managing diabetes is a multifaceted endeavor that requires patients to track values as well as manage their lifestyle. We consulted with diabetes experts and diabetic patients who advised us that the most important factors are monitoring meals, blood glucose levels and medication.

To address the key issues of diabetes management and to best utilize the unique nature of conversational interfaces, the primary objective can be broken down into the following chatbot features:

1. The user should be able to store and view reports about blood glucose values.
2. The user should be able to store and view meal related information.

3. The chatbot should remind users to take their medication and have their meals on time.

While a chatbot could achieve some subset of the above features through a simple rule-based system or through the use of buttons, our chatbot goes further as it understands user intention and conversation context and replies using natural language. This is the secondary objective of our project: to create an easy to use system that utilizes a combination of natural language and traditional user interface elements, to deliver a user friendly experience.

Through this project we also aim to get a deeper understanding of the current state of NLP technology and Chatbot interface technology.

One of the key challenges of implementing such a complex system is the fact that NLP technology is constantly evolving and as such there are very few resources such as documentation and no standardized best practices to follow for training. We took an experimental approach in the development phase to tackle this issue.

Testing a personalized system is another challenge that requires an unorthodox methodology, as the number of possible paths the program could follow is far more than traditional software. We chose to rely on automated testing as far as possible with integration testing done manually following a custom script.

1.3 Literature Survey

The literature survey was conducted over the summer of 2016 and focused on current and upcoming technologies as while chatbots have been around for decades, they have only recently integrated AI to achieve a truly useful purpose.

To our best knowledge, there are no existing chatbots on Facebook Messenger that are designed specifically for helping users manage diabetes, as such we divided our literature survey into four parts:

1. Existing apps and other digital solutions for diabetes management.

To better understand the current solutions to the same problem that use different technology.

2. Existing messaging platforms that can be leveraged for hosting the chatbot.
   The chatbot needs to be present on a messaging platform like Facebook, we analyzed the different options to decide which would be best for our use case.
3. Artificial Intelligence engines that could be used to power the brain of our bot.
   There are many different engines that use natural language processing (NLP) to understand user-input, we analyzed a few before choosing the most appropriate one for our use case.
4. Existing chatbots on the Facebook Messenger Platform
   To better understand the current technology, we analyzed how it is applied to different problems.

1.3.1 Existing Services

1.3.1.1 HealthTap

HealthTap is a medical question-answer and teleconsultation service that was one of the first available bots on the Messenger Platform. Users simply message the bot a question and it finds an answer from a database of collected entries from physicians. If it cannot find the answer it refers to a real life doctor and replies to the user in a few hours. HealthTap is used by thousands of patients on Facebook; however, it does not feature any personalized features for diabetics and is more actively used through their mobile application.[4] Their chatbot is just an intermediary to handle contacting between two humans and does not use any AI to process the questions as of now.

1.3.1.2 mySugr

mySugr is a service that provides different apps for diabetic patients. These apps are similar to many others on the app stores and provide simple services such as a logbook to store blood glucose values, an educational app to get information and a quiz app. These apps have a common drawback, the user would have to install and get used to multiple different interfaces and these apps lack personalized feedback and a human touch. [5]

1.3.1.3 Omada Healthcare

Omada is a ‘digital therapeutics’ provider that combines a mobile app and connected fitness equipment with a personal lifestyle coach who gives the patient insight. Their flagship Prevent program has shown successful results in trials and patients have shown an affinity to the personalized feedback they provide. The major drawback of Omada’s services is that they offer personalized feedback through humans, which is not scalable in a global market. This project aims to replicate Omada’s results but with AI playing the roles of caregivers. [6]

1.3.2 Messaging Platforms

The original pioneer of the conversational interface in this re-emergence has been WeChat, which has proven very successful in China. While WeChat is widely used to do everything from pay bills to order food, the interface is clunky and is little more than a web browser embedded into a chat application. There is much room for improvement. Eyeing this success, many messaging platforms have introduced support for building chatbots in the last few months. These include Skype, Kik, Slack, Telegram and Facebook Messenger.

Skype and Kik are relatively new and geographically limited platforms as they are not used widely for messaging over the world. Slack is primarily based on enterprise communication and as such the bots on Slack are oriented around productivity in the office space. Telegram is slowly gaining popularity but despite a fantastic bot service, it is yet to become a global platform.

Facebook Messenger offers the best of both worlds. Over 900 million people all over the world use it and on April 12 at the F8 conference, Facebook launched the Messenger Platform, which allows developers to build chatbots on their platform. Given the vast existing membership of the Messenger platform and the availability of engine-agnostic APIs for making bots (meaning we can use any AI engine we want), Facebook Messenger is the ideal choice for our project.

1.3.3 Artificial Intelligence Engines

Facebook Messenger uses the Wit.ai engine by default. This engine uses a story and role based approach to training the bot and has a simple GUI that can be used to declare Entities and Intents. However, this engine is under rapid development and as such it's features and methods are always changing. A stable version has recently been released and with the new conditional flows Wit.ai has become even more powerful at NLU and NLP.

Api.ai is another popular choice of AI engine that is primarily used for building chatbots. As api.ai got acquired by Google on the 19th of September, their service is likely to undergo a lot of change in the near future. [7]

IBM's Watson supercomputer first became popular for winning 'Jeopardy' against the human champions in 2011.[8] Since then, IBM has released a public API for people to use Watson's AI features. With separate algorithms and systems for Natural Language Classification, Dialog, Conversation, Emotion Recognition and more, Watson is extremely powerful but is expensive for small scale research projects such as this one as the user is charged per message which can accumulate very fast during research and development.

1.3.4 Existing Chatbots on the Messenger Platform

Uber is an online transportation network company. It launched its Facebook Messenger Chatbot in December 2015. It uses Facebook Messenger to interact with the user. The user can order the car by interacting with Uber's chatbot in Facebook Messenger. According to the video demo, the user can click on the address and press the "Request a ride" item with the car icon, and Messenger will provide the order information, including the map, contacts for the driver and payment options. We can see that it mainly relies on the GUI to interact. For example, it uses a button to order a car and pay for it. It seems not smart. It just matches the address in the conversation and mainly use the button to interact. Although it can report current traffic information in English, they just look like generated by some templates. In addition, the demo video does not show its ability to respond to user's message. [9]

Poncho is a chatbot used to report the weather. It can tell the user weather and set the weather alert. The user can tell it the location and time of the alert. Normally the user can choose to interact with the button instead of the text message. It can only understand simple sentences, so using buttons is usually more efficient. We found the update feature of Poncho which alerts you about bad weather to be useful as it was similar to the Reminder feature we want to implement in our chatbot. [10]

1-800-Flowers.com is an online flower shop. On its Facebook page, users can use its Facebook messenger chatbot to order flowers. This chatbot is similar to other 2 chatbots. It can only understand simple sentences. Similar to the Uber's chatbot, users can pay their orders in Messenger. First, it requires order information, including address, phone number, number and type of flowers and notes. If the user does not send a clear text, it will give them some examples or standard input format and ask them to re-text. When the user needs to select the type of flower, it will display some options with images. Finally, the user can use the GUI to pay for the order. [11]

Overall, 3 chatbots can only recognize simple natural language, but they can help user through the GUI and button to complete some specific tasks. Uber and 1-800-Flowers.com even have a payment function in the messenger to help them order quickly.

2 Design

We chose ease of use as our primary directive in the design process. To use this chatbot, the user can log on to Facebook Messenger and chat with the bot like any other friend. The user can ask the chatbot to save or retrieve stored blood glucose, send pictures to save meals and go through a tutorial process to understand how to use this new form of human computer interaction.

The figure below shows an overview of the different features of the chatbot:

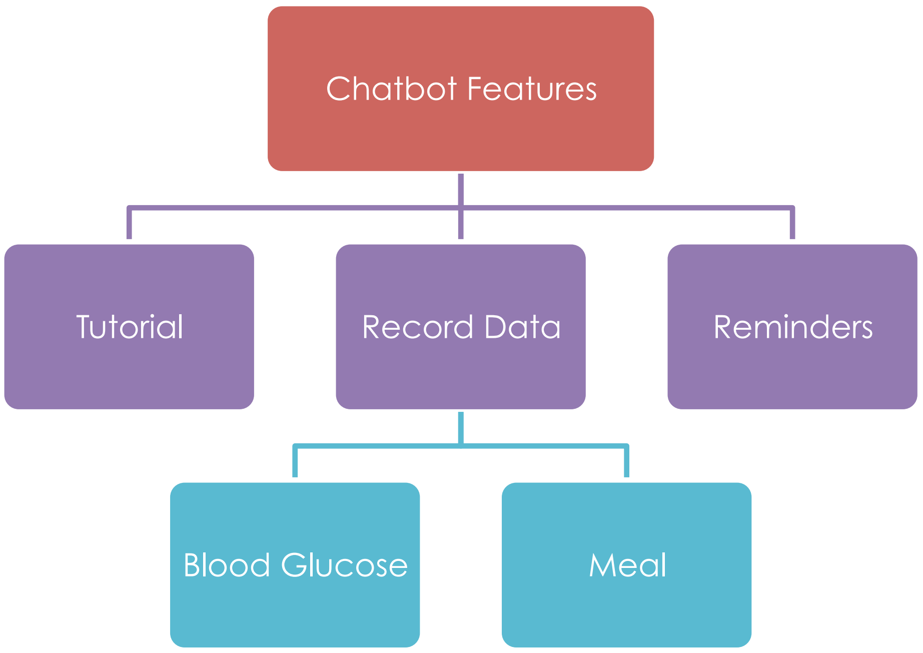


Figure 2.1: Feature Overview

In detail the design of the chatbot system as a whole can be broken down into the following parts:

2.1 System Architecture

On a technical level, chatbots are just web applications with a natural language processing component and a conversational interface in place of the regular click-and-type user interface. To keep with this existing pattern, we used the Model View Controller (MVC) design pattern so that different components like the nlp engine could be replaced if necessary in a modular manner.

The diagram below shows the three main components of the system and their connections:

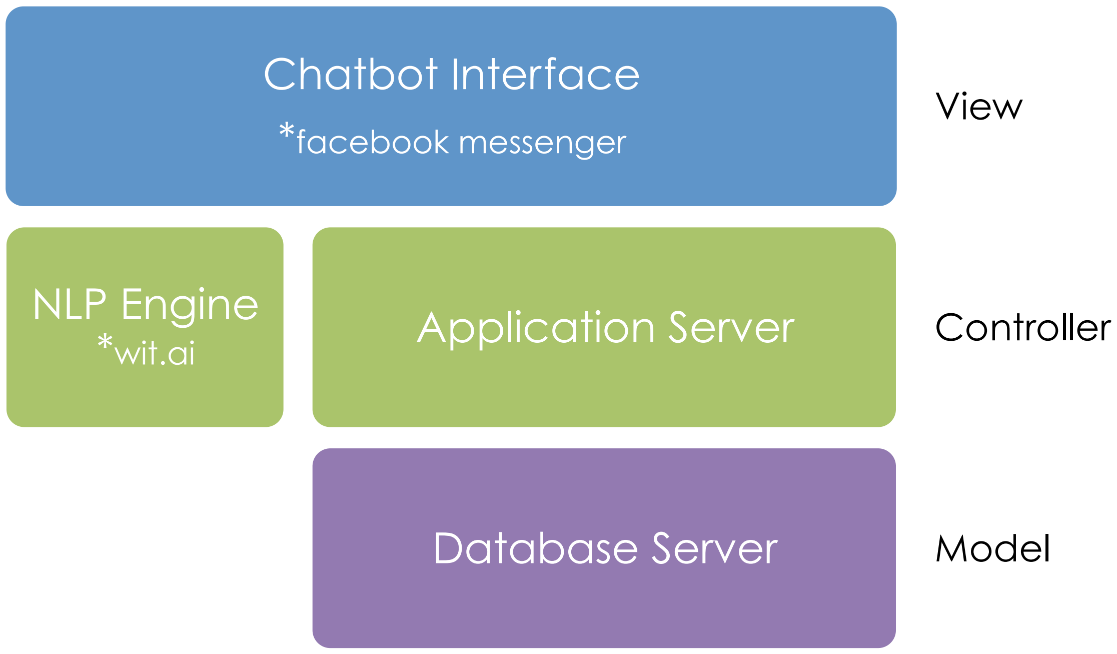


Figure 2.2: System Component High Level Overview

The chatbot interface is hosted using the Facebook Messenger Platform; this acts as the View in our MVC architecture. The Control is shared between our application server and the NLP engine; which interprets the user input. The data is stored in a traditional database server; which is the Model in our MVC architecture.

As per the MVC pattern, the user only ever interacts with the View with the other two layers hidden and the flow starts from the top to the bottom and ends at the top again in response-request cycles.

2.2 Conversation Control Flow and System Logic

Unlike traditional controllers, our chatbots will split control between the application server and the NLP engine in order to interpret natural language commands. This makes the flow challenging to map as well as increasing the complexity of development, for example: if the user is halfway through the process of registering a blood glucose value and decides to record a meal instead, due to the lack of a 'Back' button the chatbot needs to automatically understand and switch between contexts. The bot also needs to remember context for when the user resumes the original task.

Our chatbot is designed to handle control on the application server as little as possible, handing over as much control to the NLP engine as possible. This ensures that the user has freedom of input rather than a fixed set of commands. The activity diagram below, showing one instance of a conversation, can help better understand the user's interaction with the system and the system logic. Processes marked in green are actions that are executed on the application server, blue processes are undertaken by the wit.ai NLP engine.

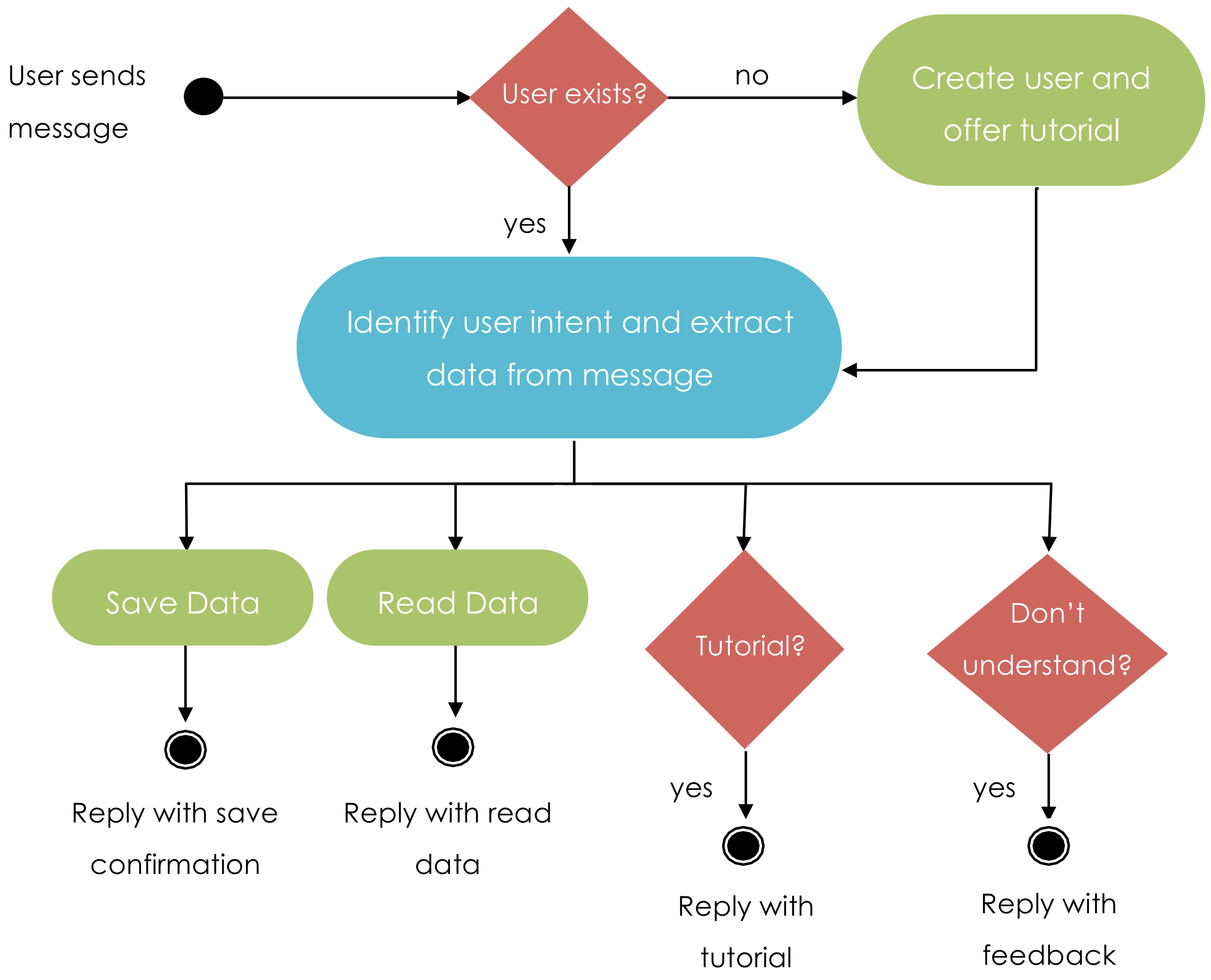


Figure 2.3: Activity Diagram Showing Control Flow

2.3 Database Models

After consulting with medical experts, we identified blood glucose levels as the key sequential value that needs to be recorded as well as the various parameters that would be useful for medical purposes. We also decided to store images for meals to provide an easier interface to users, just like taking pictures of food for friends.

The diagram below shows the design of the database and tables as an overview:

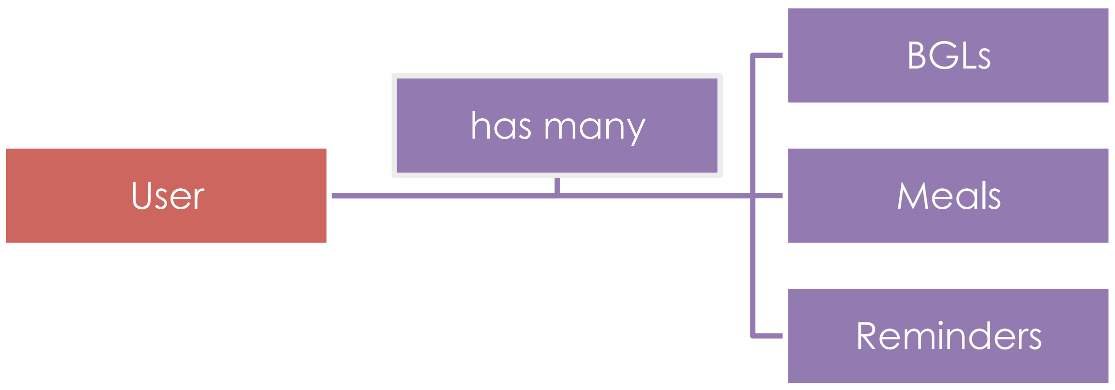


Figure 2.4: Entity Relationship Diagram

As seen above, the main User model is associated to the blood glucose values, meals and reminders models through one-is-to-many relationships. We chose not to model these as weak entity relationships, but rather as strong entities for each to allow us to have greater control over them individually. This is important as the chatbot is not always likely to follow our fixed flow and having free control over the individual models through their own controllers allowed us to implement modular functions.

The User record stores name, weight, height and age data for all users. The BGLs are stored with blood glucose value and record time while meals are stored as images. Reminders are stored as reminder type and date. A more detailed Entity-Relationship diagram can be found in the appendix.

2.4 Chatbot User Interface

Chatbots are different from conventional mobile and web applications due to the limitations on the number of ways they can receive input from or output data to a user. In order to keep the user engaged it is vital to not come across as a machine with robotic replies. While we are limited by the user interface elements provided by Facebook Messenger Platform, we have control over our chatbot's personality and look.

Our chatbot is called 'Bo' a friendly name for Bot. We found that the global range of Facebook Messenger and the spread of Diabetes makes it difficult to identify a target demographic to model the bot's personality for. To cater to the widest possible audience we decided to choose a unisex name that is not tied to a particular region of the world. We also designed Bo to stand out from other Bots by using more natural language to express personality rather than the approaches page design and design of UI elements used by bots like Poncho and Uber.

Figure 2.5: Bo Facebook Profile Picture[1]

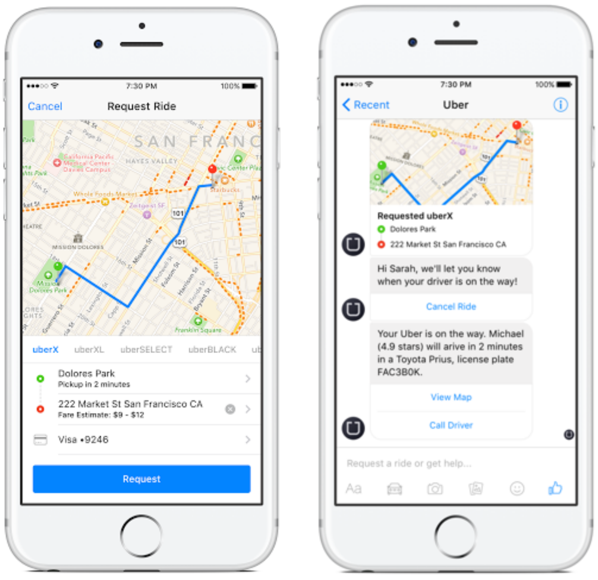
We chose to make Bo a friendly but unobtrusive personality, as users will have to engage with it frequently over the course of the day, we wanted to ensure that Bo does not intrude upon their lifestyle by taking too much time. This does not leave much room for jokes and other side conversations so Bo steers the conversation back to the main flow in a natural manner.

In addition to the conversational elements Bo has been designed to use different Facebook elements such as the typing sign to indicate longer waiting times (for example: when preparing reports) and can also send attachments and images to help

[1] Icon made by Roundicons from www.flaticon.com

make the conversation more vibrant. Some examples of these UI elements can be seen below:

Figure 2.6: Bo Sending Typing Message



7:30 PM

Cancel Request Ride

uberX uberXL uberSELECT uberBLACK

Dolores Park
Pickup in 2 minutes

222 Market St San Francisco CA
Fare Estimate: $9 - $12

Visa •9246

Request

Recent Uber

Requested uberX
Dolores Park
222 Market St San Francisco CA

Hi Sarah, we'll let you know when your driver is on the way!

Cancel Ride

Your Uber is on the way. Michael (4.9 stars) will arive in 2 minutes in a Toyota Prius, license plate FAC3B0K.

View Map

Call Driver

Request a ride or get help...

Figure 2.7: Examples of Facebook Messenger UI Elements on Uber Bot[2]

2.5 Designing a System for AI Training

To easily sync with the Facebook Messenger Platform, we chose to use Wit.ai as our natural language processing engine. Wit allows us to train an instance of a bot on their website and then interface this bot with our application server using an API over HTTP.

Training on Wit is done in the form of 'stories', each of which is one conversation from start to finish. The bot learns these stories and adapts to similar stories automatically. The bot must also be trained to execute certain 'actions' at the right time that allow it to control data on the application and database servers.

The most important part of Wit training is 'entities'. These are the pieces of data or valuable information that we want to extract from the user input. Wit has many

[2] https://techcrunch.com/2016/01/05/facebook-messenger-bots/

in-built entities such as ‘time’, ‘quantity’, etc. that can extract that data from the user input. Additionally, wit allows us to use our own entities that we can define for individual use cases. These entities are also used to capture user ‘intent’, which is an indication of what the user wants to do.

As there are few examples of training a chatbot AI to perform many related and un-related tasks, we designed an experimental approach where we would individually train each feature on a separate wit instance before integrating them onto a single bot. This idea soon failed as the bot failed to work successfully after integration due to conflicts between the *stories* and *entities* that were trained separately.

Our next design solution was to take an incremental approach to training the bot. In this approach, we designed a flow beginning by training the simple features such as the yes/no questions, then slot-based features (user fills in the blanks) and then move on to the more complex branched flows where there are many free-text possibilities. This design philosophy allows us to ensure that simpler training never overwrites complex training and continuous regression testing can ensure that the newer more complex stories don’t break the functioning of the older stories.

Over winter 2016 we interviewed some diabetic patients, who gave us the insight that a majority of the Save/Read tasks that the system needs to perform can be encapsulated in a few keywords such as ‘What’, ‘How much’ and ‘Show’, ‘View’. Based on these interviews, we categorized user intentions into the following:

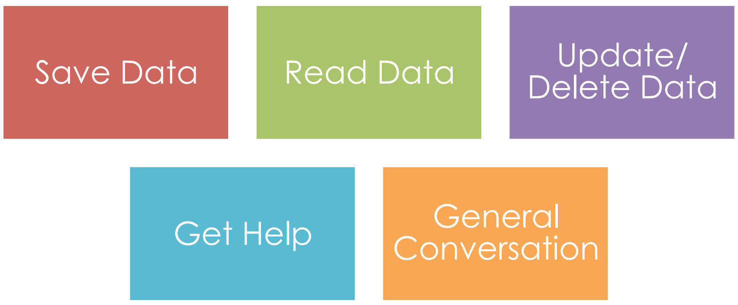


Figure 2.7: Examples of Facebook Messenger UI Elements on Uber Bot

With the wit.ai engine we can capture these user intentions as *stories* and *intents* in our implementation.

3 Implementation

As seen in the system architecture overview above, the chatbot software can be divided into three main components: the user interface and Facebook Messenger side code (equivalent to a view in the MVC pattern), the wit engine and application server (equivalent to the controller in MVC pattern), and the database server (the model in MVC pattern). We chose Ruby on Rails as our development environment as it is based on the MVC pattern and gives us the ability to rapidly develop the web server and spend more time on developing the chatbot interface and NLP capabilities.

First the application server was set up, as it is the heart of the system architecture. Then the database server and relevant models were created. The web server was then connected to the Facebook Messenger platform, followed by connection to the wit engine. Detailed explanation of these steps is given below:

3.1 Build and Deploy the Web Server Environment

The web server consists of two parts, the application server and the database server. For the application server we chose to use a Ruby on Rails (RoR) as our main development framework. We chose this setup as RoR is designed with the MVC pattern and allowed us to rapidly develop the web server thanks to its scaffolding and generator features. RoR also ships with a dedicated test suite, using the ‘Minitest’ gem, which will allow us to perform unit tests easily in a separate environment.

Most importantly, Rails comes with ActiveRecord, a database agnostic helper, which provides easy access to the databases and comes bundled with a suite of helper functions that allow easy conversion from natural language into numbers/dates, for example: the time_in_words helper function allows us to state the datetime extracted from the database in easy words instead of complex numbers. (Ex: 2 days ago instead of UTC formatted time)

Ruby is a multi-paradigm language, which is important as the wit.ai actions have to be implemented as lambda functions in order to be called automatically. It also comes with supported libraries (gems) for both wit.ai and Facebook Messenger. One vital gem during our development process was the 'dotenv' gem. This gem creates a secure environment file to store all the API keys we need such as wit.ai and Facebook Messenger API keys. A complete list of all gems we use is available in the appendix.

For local development and testing we are using *ngrok* and we are using the Heroku cloud platform as a live deployment environment. We need to use *ngrok* as Facebook Messenger requires all connections to be made over secure HTTPS protocol and *ngrok* forwards localhost to a secure address to allow local development. We chose *Heroku* due to two main reasons: it is free to host a simple service such as a chatbot during the production testing phase and provides free SSL certification, a requirement for interaction with Facebook webhooks. The downside of using Heroku is that task scheduling, a server feature we will require to implement reminders, is highly limited for free users. However, this should not affect our development and testing plans as we can use Linux *crontab* to test scheduling locally. This issue can easily be fixed in deployment by buying a Heroku premium account for actual real-life use cases.

The image below shows the detailed system architecture and the different technologies we have used to implement the modules:

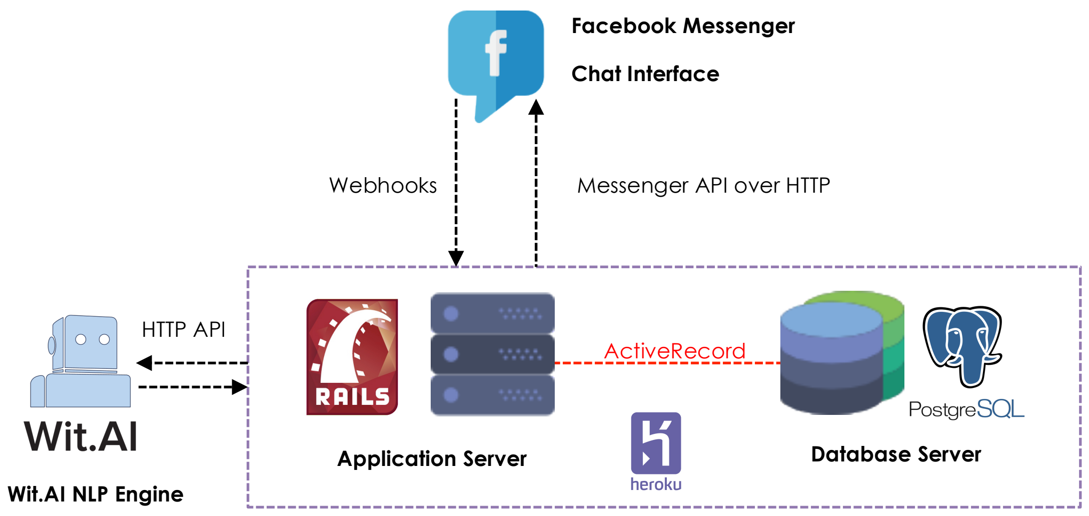


Figure 3.1: System Architecture

3.2 Implement the Server-side Control Logic

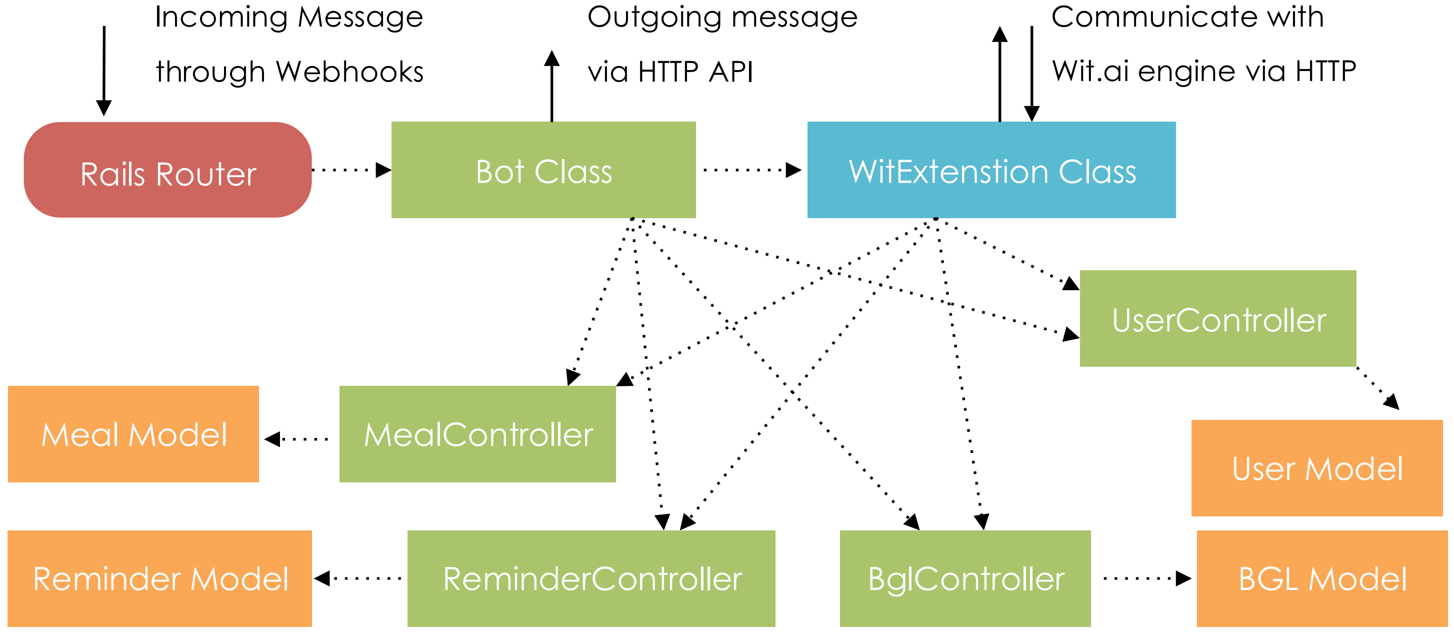


Figure 3.2: Internal Ruby on Rails Server Architecture (dotted line represents internal process)

Figure 3.2 above, shows the internal architecture of our Web Server, which comprises of the application server (ROR) and the database server (ActiveRecord + PostgreSQL). Green is used to represent the Ruby on Rails classes for Controllers, Blue is used for the WitExtension Class that manages the communication with the NLP engine and, Yellow is used to represent the Rails Models; which connect to the database through ActiveRecord. Note that there are no views, as our application does not use traditional views (replaced by conversational interface).

As shown in the diagram above, the internal architecture of our system is similar to the typical MVC architecture in a Rails application. The Rails *router* forwards incoming messages from users to the *Bot Class* through Webhooks. The *Bot Class* is an extension of the *facebook-messenger* Ruby gem. This class acts as a wrapper for the Messenger HTTP API. After checking for urgent errors (such as no message body), the *Bot Class* forwards the message to the Wit.ai engine through the *WitExtension Class*. In addition to code that handles messages, *Bot Class* also contains the setup code necessary to validate Webhooks when it receives validation requests from Facebook, as well as code to manage post-backs when user clicks buttons or performs other actions on the UI. The *Bot Class* can also directly access the controllers and is designed to take certain actions automatically without sharing control with wit. For ex: If a user sends the Bot a picture, there is only one possible intention: to save it to the database. As the user doesn't have to be asked for any further information the *Bot Class* directly saves the data and sends the user an appropriate reply.

Wit Extension is a custom class that is built on top of the official *wit-ruby sdk*. The *WitExtenstion* class contains the *actions* necessary to manage the data (CRUD) as well as the *send* function. These actions are implemented as lambda functions and they are passed on to the Wit.ai engine. After extracting the user's intention and entities from the message, the wit.ai bot calls these functions as necessary (based on training) and sends the correct reply to the Application Server to forward back to the user. *WitExtension* calls the other controllers as necessary which in turn manage the respective models in the database and it never directly modifies the Models.

In addition to storing and retrieving data, the chatbot must be able to set reminders for the user, to achieve this we are using *Heroku Scheduler/Crontab* to periodically release a task from the *Rails Rake* tasks queue that might be pending. These tasks are populated from the entries in the *Reminders* model. This set up is very similar to *cronjobs* in other web application frameworks, and is the only part of our application that breaks the request-response cycle. When a user sets a reminder, it is added to the Reminder model like any normal data, the *Crontab/Scheduler* runs through the database every hour and activates all reminders that belong to that hour. At the moment we have chosen hourly reminders as it is a limitation of the free tier for Heroku, the code can easily be modified to check for reminders more often, allowing users to specify the exact minute they want to be reminded.

During development we realized that communicating with Facebook Messenger over *ngrok* is very different from communication over an actual deployed server, as *ngrok* does not account for the lag time of the server process booting. Sometimes this lag time can cause Facebook to send the message more than once and this recursive calling has to be handled on the server side. There is also a higher down-time on *ngrok* than on a live server cause requests to queue up at times. We handled this issue by running tests on *ngrok* in a structured manner with one test only beginning after the past has been completed.

3.3 Build the Database

Rails and ActiveRecord are database server agnostic, meaning they can be used with any database technology. For it's fast access and our previous experience with it, we chose PostgreSQL as the database server. PostgreSQL was also preferred over the Rails default Sqlite3 for development as Heroku does not allow Sqlite3 to be used on their platform and to ensure concurrency between data, we set up PostgreSQL as the database system across all our environments.

We used Rails generators to create models and the relations that corresponded to our ER diagram. As the user is free to input any amount of information due to the free entry nature of chatbots, we implemented strict constraints on the database to ensure functionality even if server side validation fails. The data types for all dates were strictly enforces as *datetime* and in lieu of *Big Integer, String* was used as the data type for all Facebook ids to protect against future changes in length of ids which are already too big to be stored as integers.

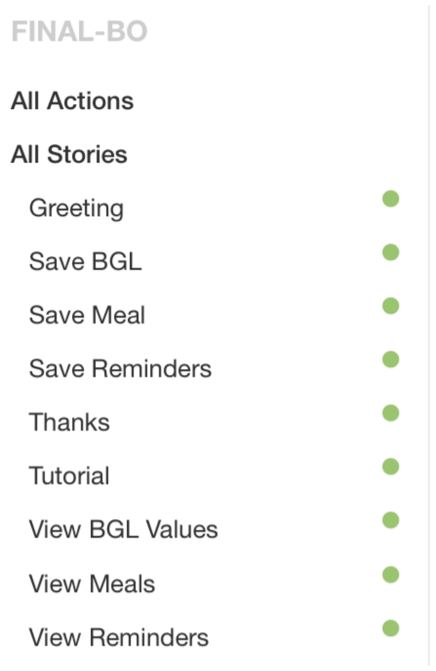
One major challenge that we encountered was that there were several different components in our system that relied on time, the Wit.ai engine and Facebook Messenger have their own clocks while the database server has it's own. Wit interprets time in US time while Messenger uses the client's local time and the server uses it's own local time (HKT during development). To overcome this issue and ensure concurrency, store all datetime information in the database in UTC and convert it to local time as necessary when sending it to the view.

3.4 Train the Artificial Intelligence Engine

The most vital part of the implementation is the training of the artificial intelligence engine. The engine is the driving force behind the chatbot and is responsible for extracting *context*, *intent* and *entities* (valuable data) from the users message. The engine also manages control of the next action the chatbot takes such as choosing the right flow to follow and the appropriate response.

In order to accomplish this, the engine needs to be 'trained' or taught to know what words are keywords, which words signify intent and what actions to call under which situation. Wit takes this training and applies its own NLP algorithms to automatically generalize understanding from our supplied instances. After the initial training, Wit uses a reinforcement learning approach to NLP by logging all messages that the chatbot receives and allowing the developer to correct the low confidence guesses manually.

In Wit.ai, training is performed on a cloud-based console where the user creates '*stories'* and tells the bot the correct answer in a form of supervised learning. A '*story'* is a general path that the conversation follows for a particular purpose. *Stories* tell the bot how to decipher a message and extract important information from it in the form of *entities*. As our chatbot accomplishes many different and unrelated features, we had to design several *stories* to manage our flow, beginning with 12 stories distributed across multiple bots. Our final bot uses nine stories that are listed in the image below:


FINAL-BO

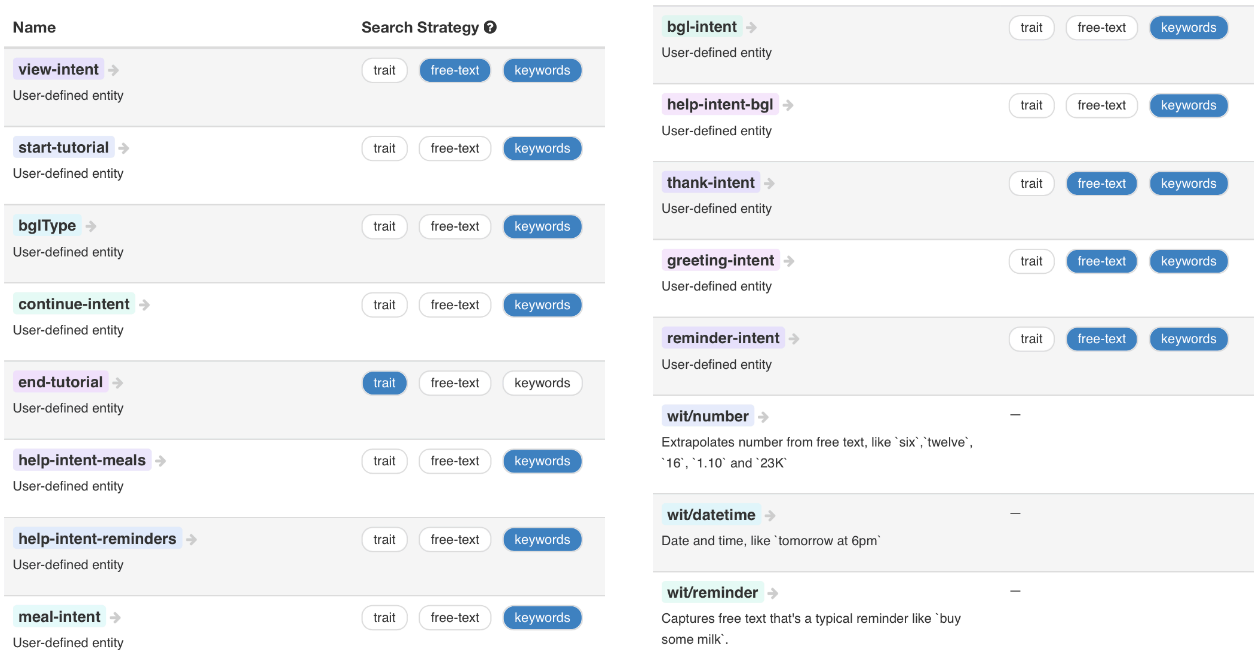
All Actions

All Stories

- Greeting ●
- Save BGL ●
- Save Meal ●
- Save Reminders ●
- Thanks ●
- Tutorial ●
- View BGL Values ●
- View Meals ●
- View Reminders ●

Figure 3.3: Stories used in our bot

In wit.ai you can both use common in-built *entities* (such as time, location, etc.) and you can also define our own *entities* to fit custom use cases. *Entities* can use one of three search strategies to infer content: ‘trait’, ‘free-text’ and ‘keywords’. Keywords uses pattern matching to look for particular words and is often combined with free text which uses pattern matching to understand content based on it’s location in the sentence. ‘Trait’ is the least reliable as it tries to infer meaning based on the entire sentence as a whole and is highly context dependent. We chose to focus on a well defined structure of keywords and free-text in our bot to ensure high confidence level. For our chatbot, we started with more than 30 *entities* before distilling it down to 16 *entities* in our final bot, 13 of which are self-defined and three of them are in-built. A complete list of all entities used can be found below:



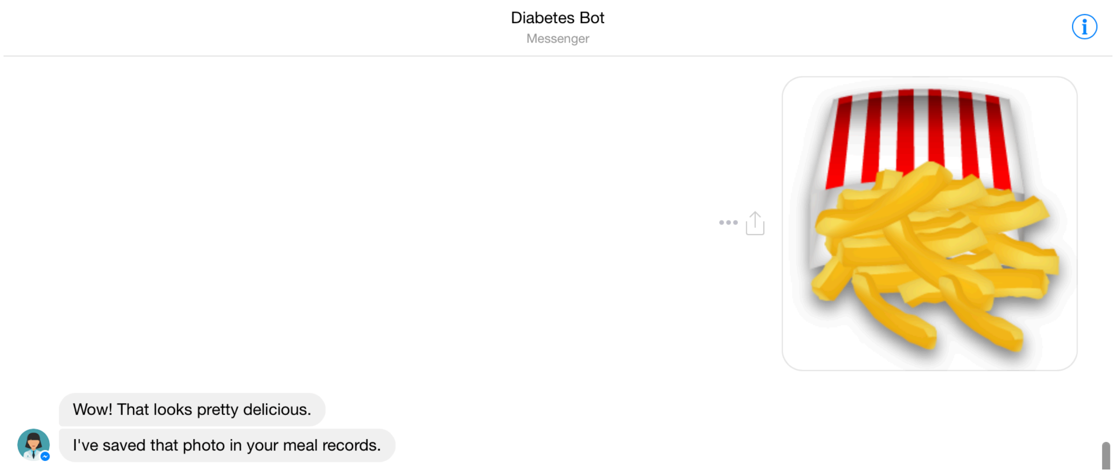
Name	Search Strategy
view-intent User-defined entity	trait [✓ free-text] [✓ keywords]
start-tutorial User-defined entity	trait free-text [✓ keywords]
bglType User-defined entity	trait free-text [✓ keywords]
continue-intent User-defined entity	trait free-text [✓ keywords]
end-tutorial User-defined entity	[✓ trait] free-text keywords
help-intent-meals User-defined entity	trait free-text [✓ keywords]
help-intent-reminders User-defined entity	trait free-text [✓ keywords]
meal-intent User-defined entity	trait free-text [✓ keywords]
bgl-intent User-defined entity	trait free-text [✓ keywords]
help-intent-bgl User-defined entity	trait free-text [✓ keywords]
thank-intent User-defined entity	trait [✓ free-text] [✓ keywords]
greeting-intent User-defined entity	trait [✓ free-text] [✓ keywords]
reminder-intent User-defined entity	trait [✓ free-text] [✓ keywords]
wit/number Extrapolates number from free text, like `six`,`twelve`, `16`, `1.10` and `23K`	–
wit/datetime Date and time, like `tomorrow at 6pm`	–
wit/reminder Captures free text that's a typical reminder like `buy some milk`.	–

Figure 3.4: Entities used in our bot (in-built entities are prefixed with ‘wit/’)

As seen in Figure 2.7 in the Design section above, we classified our user’s actions into five broad categories: save data, read data, delete data, get help and general conversation (includes greetings and goodbye). Note that data here means one of three things: meal images, blood sugar values or reminders. As such the categories can further be clustered as: save/read/delete meals, save/read/delete Blood Glucose Level (BGL) values, save/read/delete reminders, get help and general conversation. Any message that the user sends to the bot should be able to fit into one of these five categories. Below we have provided a description of how we mapped each of these categories to its own set of *stories* and *entities* on the wit platform.

3.4.1 Training the Save/Read/Delete Meal Category

To save the three different types of data the user uses three very different approaches. For meals as we are storing only images, the *Bot Class* on the server directly handles the saving of data as described above in Section 3.2. We used a *story:* 'Save Meal', that is called after an image is stored in order to reply to the user with confirmation that the story was saved. The figure below shows how the bot handles images it receives:



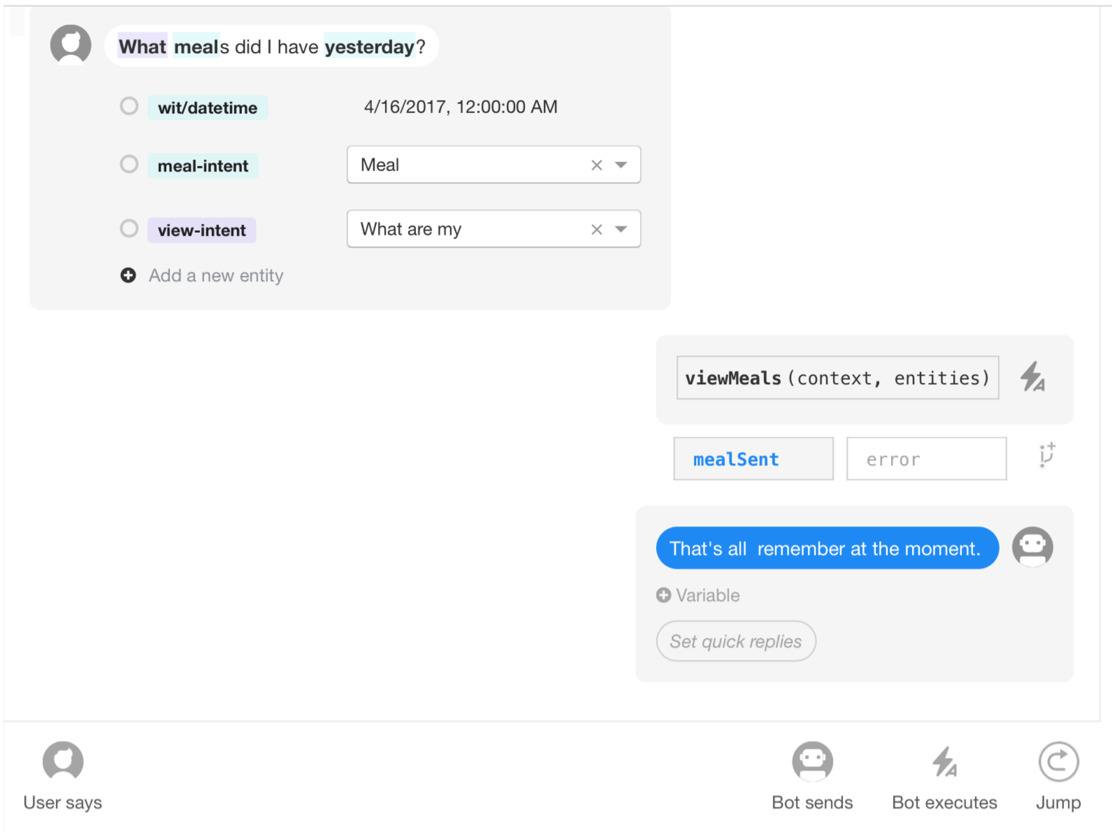
Wow! That looks pretty delicious.

I've saved that photo in your meal records.

Figure 3.5: Screenshot of bot saving meal image and replying

To read meals, first, the *'meal-intent'* was created that identifies keywords such as 'meal', 'eat', 'etc.' to understand that the user is talking about meals. Next the *'view-intent'* was made which uses keywords such as 'view', 'show' and 'what did I' to infer that the user is trying to read data. The View Meals story states that if both *meal-intent* and *view-intent* are found then the bot calls the lambda function associated with the *viewMeals* action before replying with the content found or an error if it can't find the requested data. This story also uses the wit in-built *entity wit/datetime* to infer the time period the user wants to view meals for.

Updating data is very tricky with a natural language interface, as such we chose a work around where the user can delete data and re-create it in place of updating data directly. In order to delete data the user needs to first read it and then select the desired data to be deleted. To avoid ambiguity in identifying which data to delete we chose not to use natural language as the interface for deleting, rather we took advantage of the Facebook UI elements to circumvent the need for delete data stories. This is further elaborated upon in Section 3.5. The image below show's an example of the wit interface for training stories:


What meals did I have yesterday?

wit/datetime 4/16/2017, 12:00:00 AM

meal-intent Meal

view-intent What are my

Add a new entity

viewMeals(context, entities)

mealSent

error

That's all remember at the moment.

Figure 3.6: Screenshot of wit story training interface

3.4.2 Training the Save/Read/Delete BGL Values Category

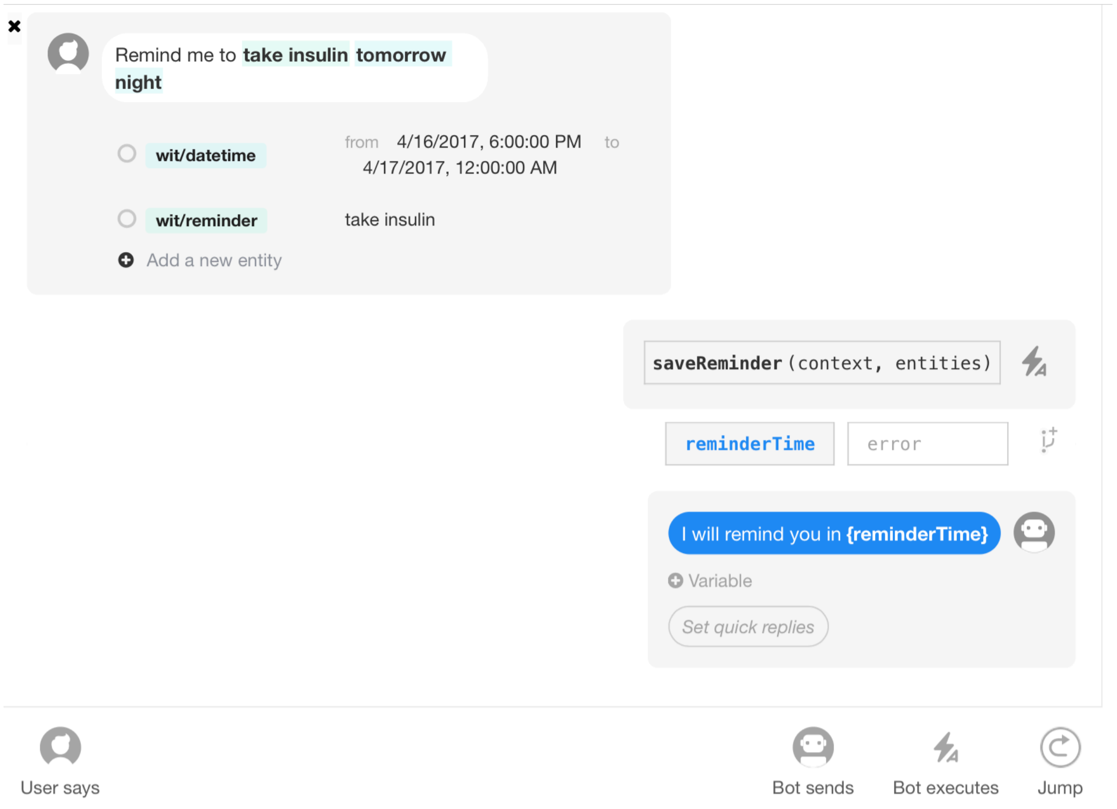
Similar to the Meals, we first created the *'bgl-intent',* which identifies that the user wants to manage BGL data through keywords such as 'bgl', 'sugar' and 'glucose'. The previously created *'view-intent'* is combined with the *'bgl-intent'* and *'wit/datetime'* to create the View BGL Values story which allows users to retrieve records based on date similar to the View Meals story.

To save BGL values the user must present the time, value and type of the reading. When wit extracts *'wit/datetime'* (used for time of record), *'bglType'* (possible values: fasting, random, post-meal), *'wit/number'* (used to extract bgl value) as well as 'bgl-intent', the Save BGL calls the *saveBglValue* function to insert these extracted values into the database.

Like meals, deleting of BGL values also takes place without any input from the wit.ai engine.

3.4.3 Training the Save/Read/Delete Reminders Category

Reminders are different from meals and BGL values in a few ways as they are more free-form and there are no fixed keywords attached to identify reminders. We first started capturing this intent by creating *'reminder-intent'* which uses free-form text as well as keywords to extract reminders from the readers' messages using the *'wit/reminder'* in-built entity. The View Reminder and Save reminder stories then follow a similar patter to the Save and Read BGL Values already made as shown in the figure below:



Remind me to take insulin tomorrow night

wit/datetime — from 4/16/2017, 6:00:00 PM to 4/17/2017, 12:00:00 AM

wit/reminder — take insulin

```
saveReminder(context, entities)
reminderTime    error
```

I will remind you in {reminderTime}

Figure 3.7: Part of the Save Reminder story, showing similar pattern to View Meal

3.4.4 Training the Get Help Category

The entire Get Help category was simplified into one story called 'Tutorial' that is invoked for all first time users, and can be called manually at any time by simply messaging the bot the word 'Tutorial' or 'Help'.

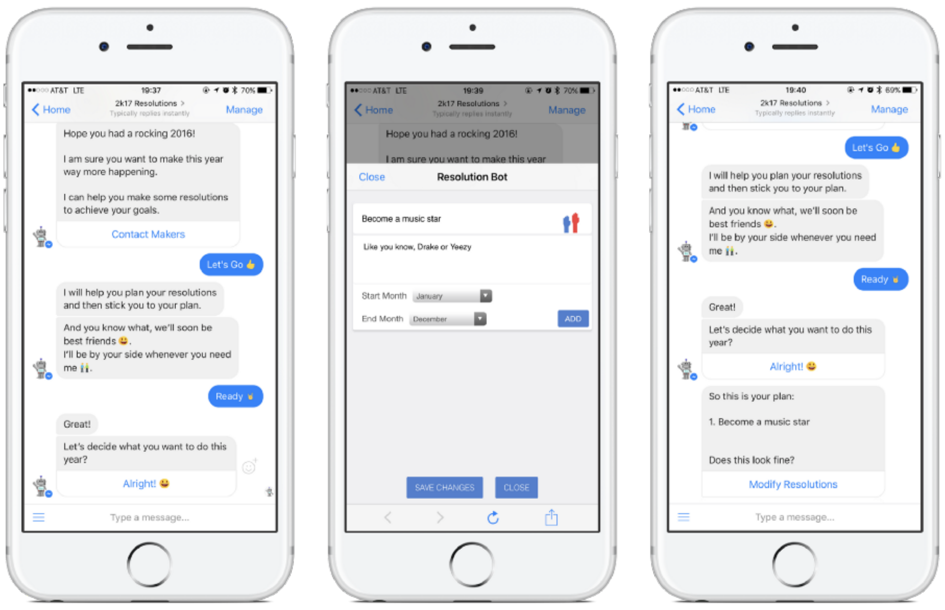
This story guides the user through all the functions and uses the *'start-tutorial' 'continue-intent', 'end-tutorial', 'help-intent-reminders', 'help-intent-meals', and 'help-intent-bgl' entities* to decipher the users chosen path through the tutorial. To make the tutorial easier it also incorporates the ‘quick replies’ feature where wit.ai suggests replies to the user (as buttons) so that they can progress to the next part of the story smoothly and quickly. This story also uses *branches, jumps and bookmarks,* three features of wit.ai that allow us to weave conditional logic into the chatbot’s process.

3.4.5 Training the General Conversation Category

The general conversation beyond the above intentions is limited to greetings and thank you messages that end the conversation. ‘*Greeting-intent’* and *‘thank-intent’* and the ‘Greeting’ and ‘Thank You’ story summarize the bots replies to these messages. Any other general conversation message is not catered to in order to streamline the bot’s efficiency.

3.5 Implement the Conversational User Interface

The chatbot interface is largely constrained by the Facebook Messenger platforms limitations, there are three main ways of information exchange: text, audio and image attachments. Messenger also provides WebViews and List and Button Templates that contain snippets of pre-formatted HTML, sent to the user within the message window. Messenger extended these features to the desktop website, enabling them to be more user friendly.



Hope you had a rocking 2016!

I am sure you want to make this year way more happening.

I can help you make some resolutions to achieve your goals.

I will help you plan your resolutions and then stick you to your plan.

And you know what, we'll soon be best friends 😃.
I'll be by your side whenever you need me 👬.

Great!

Let's decide what you want to do this year?

Become a music star

Like you know, Drake or Yeezy

Start Month January

End Month December

So this is your plan:

1. Become a music star

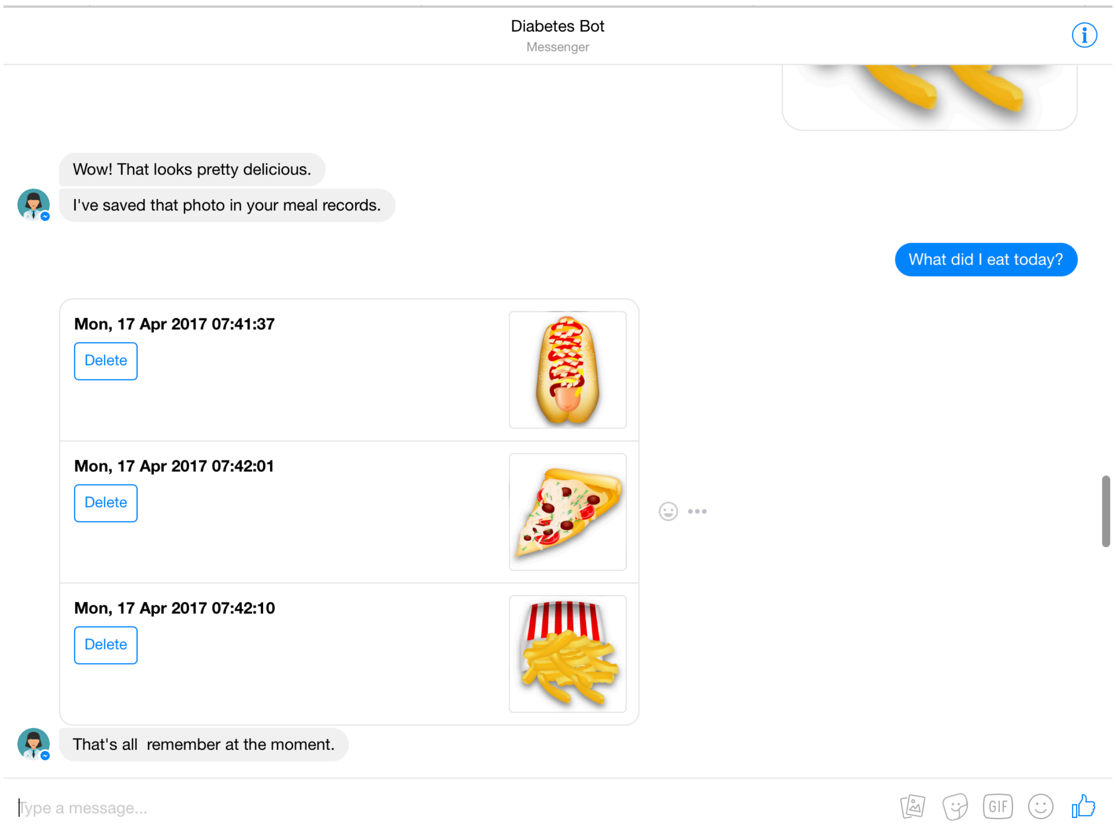
Does this look fine?

Figure 3.8: Examples of WebView, lists and Button elements from Messenger UI

In addition to the message window itself, the Facebook Page that is linked to the bot plays an important part in the user interface design as the profile picture for the bot is the same as the page and users find the bot through the page.

To implement Bo's user-friendly design, we chose to use casual natural language when Bo replies to user messages. For ex: when the user stores a new meal image Bo replies with a comment about how it looks delicious or how it's hungry.

One of the key challenges of creating an application that uses a natural language interface is creating an interface for simple CRUD operations. While create and read operations are covered by the NLP engine, the delete operation requires precise input from the user. To simplify this issue, we leveraged Facebook Messenger's List template that allows us to list all previous records. An example of this is shown below:


Diabetes Bot
Messenger

Wow! That looks pretty delicious.
I've saved that photo in your meal records.

What did I eat today?

Mon, 17 Apr 2017 07:41:37
Delete

Mon, 17 Apr 2017 07:42:01
Delete

Mon, 17 Apr 2017 07:42:10
Delete

That's all remember at the moment.

Figure 3.9: Example of Bo using List Template to show meal records

Each records has a button attached which returns the record id to the server when the user clicks it, letting the server know exactly which record to delete. The downside of this approach is that the user needs to perform a read operation to locate the record before deleting it; this is not as oriented towards natural language as the rest of our interface.

4 Testing

We developed one feature at a time and as such unit testing was performed according to the features: Meals, BGL values, Reminders and then Tutorial and Help. Each feature's save function was checked first followed by read and delete with careful attention paid to multiple use cases from the user as detailed below.

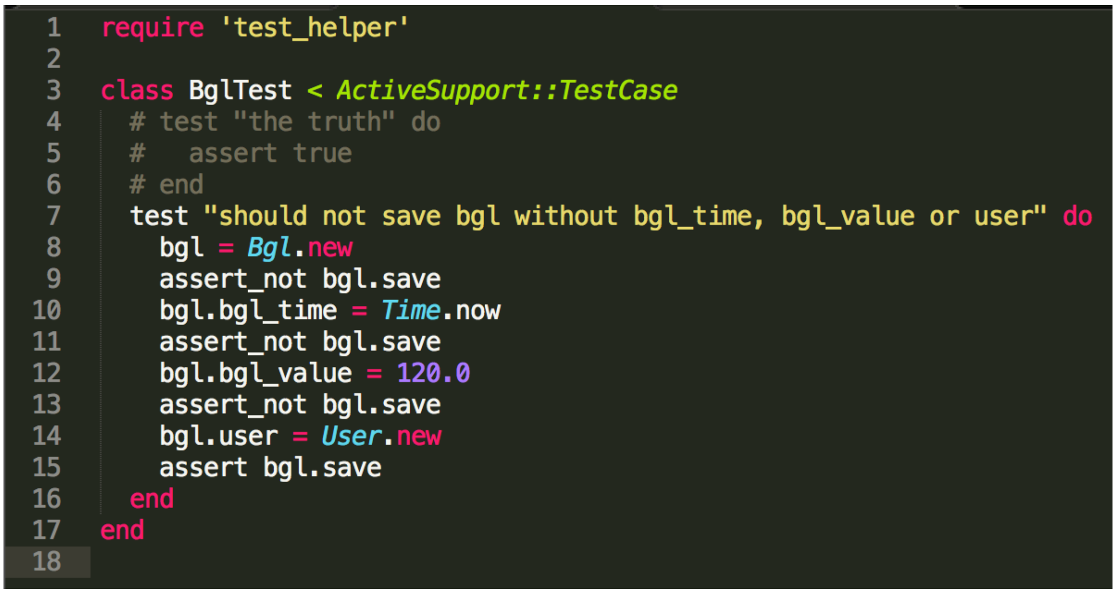
During the development phase we used the setup with *ngrok* to continuously perform unit testing on each component of the system as we built it. We used the *web_console* gem for debugging during development. The unit testing was first performed on the application server and the database server to ensure that the system functions correctly internally. Next, unit testing was performed on wit.ai and finally; Facebook Messenger unit testing was performed simultaneously with the integration testing as the Messenger platform cannot function independent of the server. The Linux Crontab was used to perform testing on the reminder feature.

As mentioned in the Design section before, to perform training efficiently we combined the agile development process with our own method of training in the order of simplest to most complex feature. During testing, this allowed us to easily perform regression testing on each feature as we performed unit testing.

Details of the different testing phases are given below:

4.1 Web Server Unit Testing

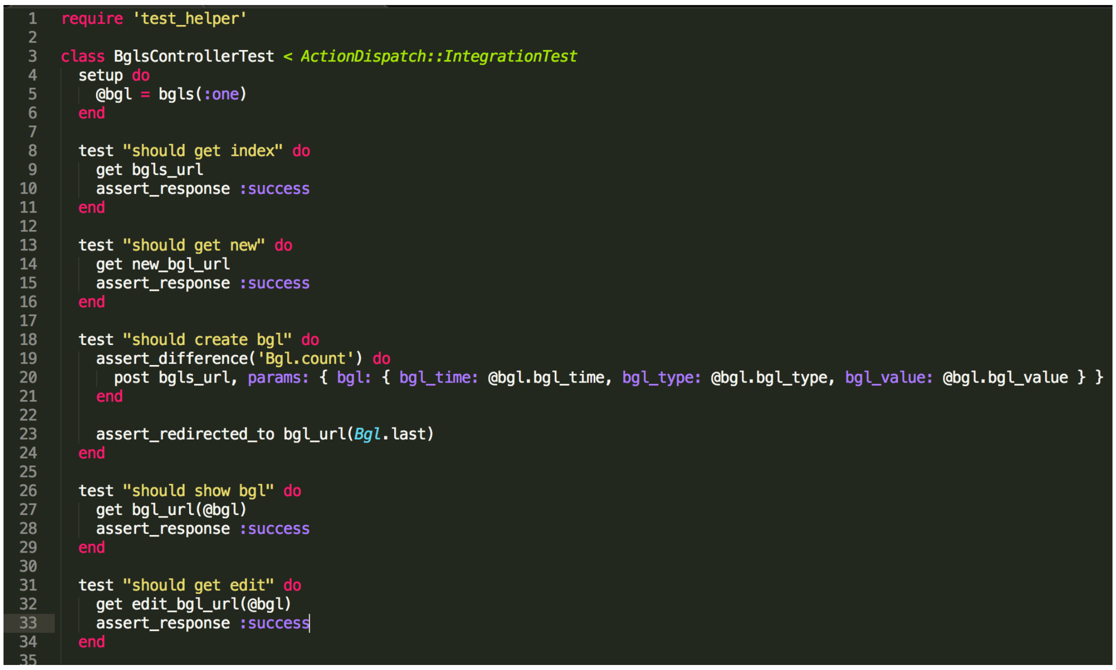
We used the rails 'minitest' gem to create a test suite for the models and controllers in our rails application. The minitest gem allows us to easily create assertion test cases for model validations as demonstrated in the figure below, which shows the validation test for the BGL model. In the image the test-helper is part of the default minitest gem.



```ruby
require 'test_helper'

class BglTest < ActiveSupport::TestCase
  # test "the truth" do
  #   assert true
  # end
  test "should not save bgl without bgl_time, bgl_value or user" do
    bgl = Bgl.new
    assert_not bgl.save
    bgl.bgl_time = Time.now
    assert_not bgl.save
    bgl.bgl_value = 120.0
    assert_not bgl.save
    bgl.user = User.new
    assert bgl.save
  end
end
```

Figure 4.1: Test case for BGL Model

The ‘minitest’ gem also comes with pre-defined tests for controllers. These were used in the default manner for our app as the control is normally handled on the wit end rather than the application server. An example of these tests is shown below:

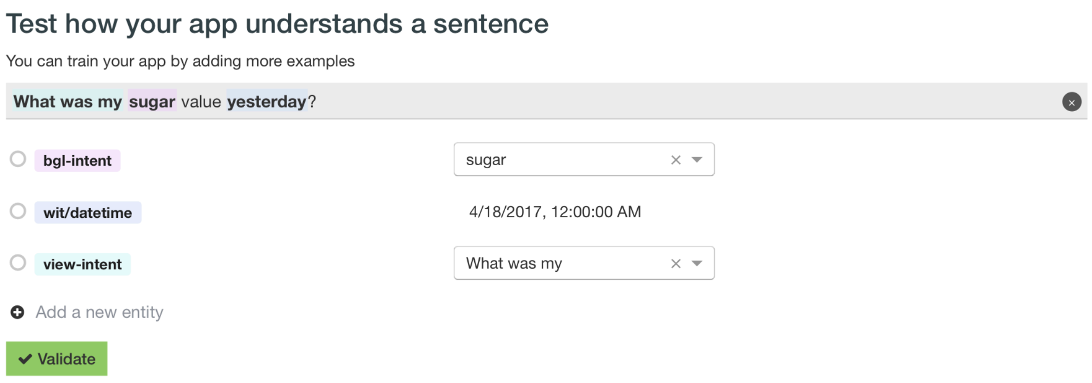


```ruby
require 'test_helper'

class BglsControllerTest < ActionDispatch::IntegrationTest
  setup do
    @bgl = bgls(:one)
  end

  test "should get index" do
    get bgls_url
    assert_response :success
  end

  test "should get new" do
    get new_bgl_url
    assert_response :success
  end

  test "should create bgl" do
    assert_difference('Bgl.count') do
      post bgls_url, params: { bgl: { bgl_time: @bgl.bgl_time, bgl_type: @bgl.bgl_type, bgl_value: @bgl.bgl_value } }
    end

    assert_redirected_to bgl_url(Bgl.last)
  end

  test "should show bgl" do
    get bgl_url(@bgl)
    assert_response :success
  end

  test "should get edit" do
    get edit_bgl_url(@bgl)
    assert_response :success
  end
```

Figure 4.2: Some test cases for BGL Controller

4.2 Wit.ai Unit Testing

Wit.ai comes with an in-built status check that let's you know if something is wrong with the training of the bot such as conflicting stories. Additionally there are two other unit testing features included on the wit platform. An online chat interface to test the correct functioning of stories and correct training of the bot and an understanding section where a bot's entities can be trained and entity recognition can tested at the same time. Examples of the three testing situations for unit testing the wit.ai bot are given below:

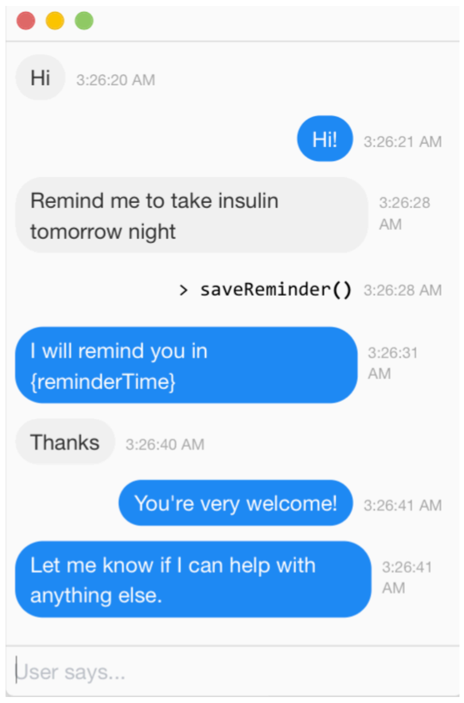

Test how your app understands a sentence

You can train your app by adding more examples

What was my sugar value yesterday?

bgl-intent	sugar
wit/datetime	4/18/2017, 12:00:00 AM
view-intent	What was my

Figure 4.3: Understanding section of wit GUI


Hi 3:26:20 AM

Hi! 3:26:21 AM

Remind me to take insulin tomorrow night 3:26:28 AM

> saveReminder() 3:26:28 AM

I will remind you in {reminderTime} 3:26:31 AM

Thanks 3:26:40 AM

You're very welcome! 3:26:41 AM

Let me know if I can help with anything else. 3:26:41 AM

Figure 4.4: Wit chat test interface

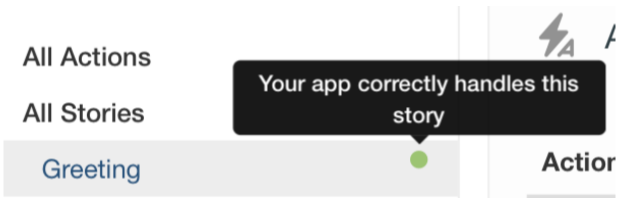


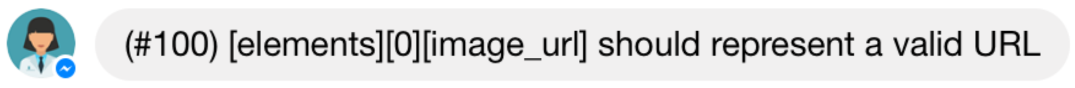
Figure 4.5: Wit gives each story a status color

4.3 Facebook Messenger Unit Testing and Integration Testing

As the server setup was our first step, we connected Facebook Messenger to our local *ngrok* server and then performed end testing by training a feature and then testing a variety of possible inputs from the user for that feature to ensure that the bot can handle edge cases as well as standard cases.

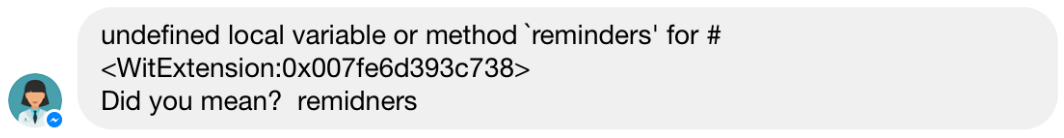
To debug test the functioning of the external APIs we used the Ruby *begin.. end* functions which allow us to try running an action and catch all exceptions. These exceptions were then passed on to the bot allowing us to debug while performing integration testing. Integration testing was performed in April and showed that the project is capable of delivering high accuracy in interpreting the user input and performs the correct action with near-perfect accuracy with the few errors that exist being attributed to the server lag rather than errors in the code or training.

Some examples of the bot sending errors during development are shown below:



```
(#100) [elements][0][image_url] should represent a valid URL
```

Figure 4.6: Errors in the Messenger API are pre-fixed by (#100)



```
undefined local variable or method `reminders' for #
<WitExtension:0x007fe6d393c738>
Did you mean?  remidners
```

Figure 4.7: Ruby on Rails errors such as syntax messages are also debugged the same way

5 Evaluation

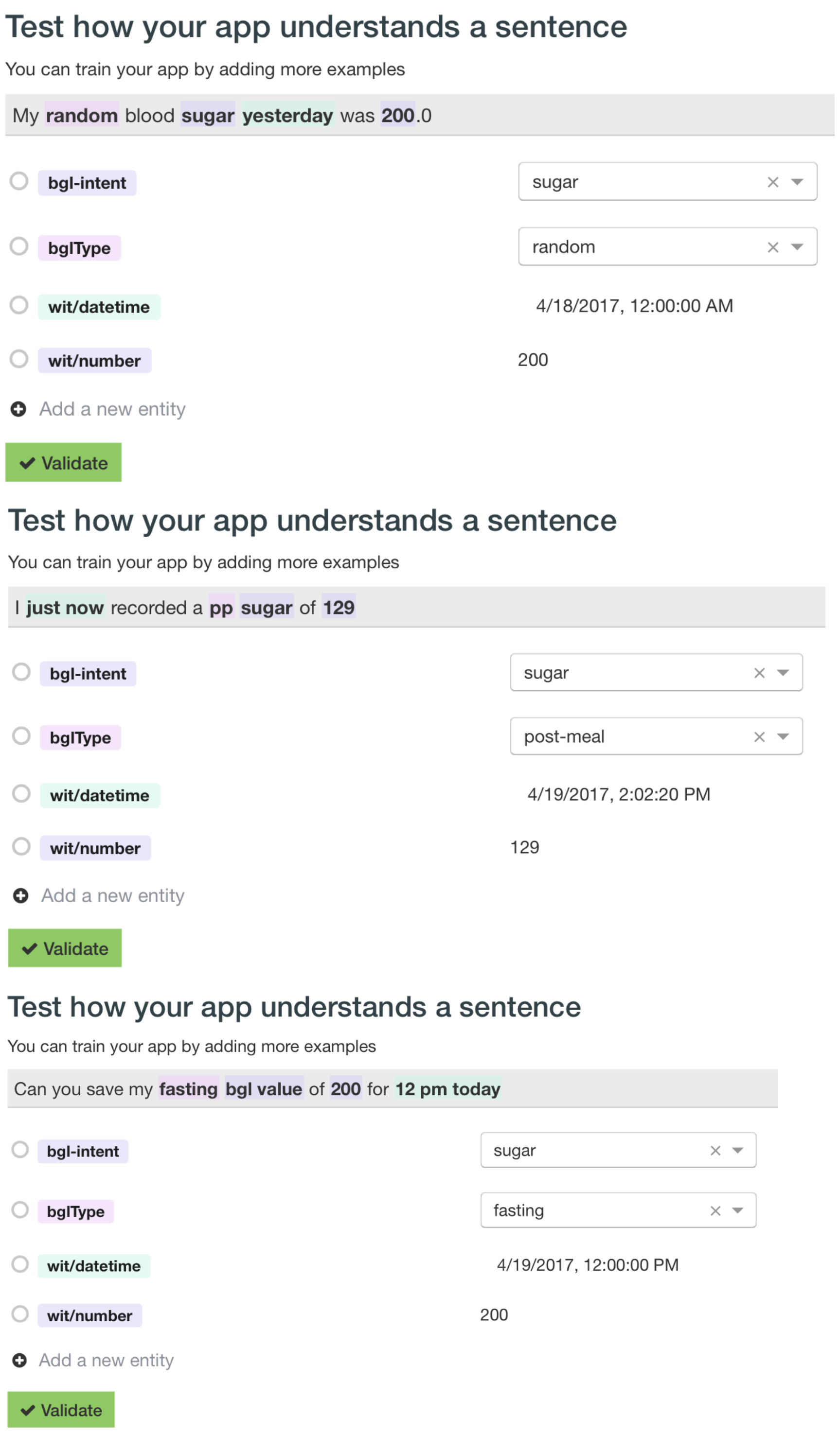
Our primary objective was to create a chatbot with which users could manage blood glucose data, meal data, and set reminders for medication and that could intuitively offer help. The evaluation phase included evaluation of the performance of our system with regards to the primary objectives.

Our secondary aim was to understand the current state of messaging platforms and NLP engine technology with respect to creating a multi purpose chatbot; this aim is fulfilled in the discussion section with the evaluation phase limited to the primary objectives. Detailed evaluation of each objective follows:

5.1 Evaluating Management of Blood Glucose Data

The first aim of our chatbot was to help users manage their blood glucose data. Existing mobile application that use the traditional GUI provide charting abilities and reports to help users visualize data, but a majority of them are unable to fully serve the needs of the user as they are dependent on the user to input their blood glucose values and it is difficult to motivate users to input their data regularly. By using a conversational interface we hoped to simplify the experience of inputting BGL values for the user.

In terms of saving values, are project satisfies the requirements completely as it allows the user to enter free form data as a message with no restraints on the format and no need to follow a sequence of steps. Below are some examples of how the bot is capable of handling different input situations to save blood glucose data:



Test how your app understands a sentence

You can train your app by adding more examples

My **random** blood **sugar** **yesterday** was **200**.0

bgl-intent	sugar
bglType	random
wit/datetime	4/18/2017, 12:00:00 AM
wit/number	200

Test how your app understands a sentence

You can train your app by adding more examples

I **just now** recorded a **pp** **sugar** of **129**

bgl-intent	sugar
bglType	post-meal
wit/datetime	4/19/2017, 2:02:20 PM
wit/number	129

Test how your app understands a sentence

You can train your app by adding more examples

Can you save my **fasting** **bgl value** of **200** for **12 pm today**

bgl-intent	sugar
bglType	fasting
wit/datetime	4/19/2017, 12:00:00 PM
wit/number	200

Figure 5.1: Examples of how the bot can handle varying inputs

For reading values, the bot uses Facebook List Templates which provide an efficient way of displaying structured data within the messenger interface, however, the key drawback of this feature is that a list can only support 2-4 items at a time and does not support any pagination by default. This is not a major issue for this use case as a person is not likely to record more than 3 BGL values per day. However, due to this the bot is only partially capable of fulfilling the requirements for managing BGL data.

5.2 Evaluating Management of Meals

Managing meals and diet are an important part of every diabetic's life. To simplify this task to a one step process we implemented a system where the user only needs to take a picture to save the meal for future reference. Saving the meals is thus very easy, and reviewing them is also easy as Messenger Lists have in-built support for images.

The utility of simply saving meals is low, however, with advances in AI image recognition technologies such as Google Vision this conversational interface for pictures could easily be converted into an automatic calorie counter and recorder. This feature has great potential for expansion but our overall observation was that in their current form, natural language processing interfaces are not very capable of handling image based data and are more suited for text based data.

5.3 Evaluating Reminders Feature

The reminders feature is a unique element of our application as it is the only part that breaks the request-response cycle. Our app is able to quickly and easily store any reminders for the user and is capable of reminding users on time. However, due to the complexity of training the NLP engine to recognize recurring reminders, our bot's current version is only capable of handling one-time reminders. All reminders are saved in memory only until they have been released and due to limitations on the free tier of Heroku, the test bot can only record reminders at a resolution of one hour. The code however (as tested on crontab) can easily be run at a higher frequency to achieve more precise reminders. Overall, out of all our features, reminders are the most suited to the conversational interface due to the high variability in possible inputs.

5.4 Evaluating the Overall Performance

While the chatbot is well trained and is capable of replying to almost all inputs with the desired output, our conclusion is that end users would still find it hard to adapt to this conversational interface as the Messenger platform has a lot of quirks which cause requests to be repeated, especially during the development phase. While most use cases would not have an issue with an extra message being sent, a data based use case like ours suffers due to this as a repeated request might try to save data twice.

One key issue with wit.ai is that context is maintained on our server and not on wit, what this means is that we need to keep track of session ids for each conversation as well as the context variable which needs to be updated entirely on our side. This adds a lot of complexity to multi-step events, for ex: if the user does not mention a date while entering the BGL values, he will have to re-enter the entire message. This is not a fatal issue as the conversation interface does not require too many words, and only a few edge cases miss these validations, however, this is a key disadvantage of existing CI technology when opposed with traditional GUI.

Our system manages to successfully fulfill the primary objectives, however the overall performance of the system when compared to similar traditional solutions indicates that conversational interfaces are not at a stage currently to be on par with traditional point and click GUI.

6 Discussion

In this discussion, the primary objectives and extent to which they were fulfilled are discussed; followed by a discussion of the current state of technologies for making chatbots:

6.1 Discussion of Blood Glucose Data Management Objective

The objective for managing blood glucose data was partially fulfilled by this project as it provides a new and intuitive interface for self-reporting of blood glucose values, however, due to limitations in the NLP engine it is not as user friendly as some traditional GUI apps.

Previously GUI apps discouraged users from self-reporting their values as the UI is cumbersome and many users are not comfortable with downloading or using an app regularly to record their values. Due to this, apps have not yet been able to overtake diaries and auto-logging as the preferred methods for BGL value reporting. With our project we can see that Conversational Interfaces simplify the process of inputting data into one atomic statement that the user does not even have to memorize as they can vary it daily as they would in natural conversation. This was the biggest achievement of our project with regard to this objective.

These limitations are mostly due to the difficulty in maintaining context or state between requests. We tried to mitigate this issue by using a single Wit Instance (using the Singleton design pattern) and generating unique ids for each conversation session. However, when combined with the propensity of the server to accidentally replay requests, our solutions were not enough and only through further development of the NLP engine can conversational interfaces realistically handle all CRUD operations.

6.2 Discussion of Meal Data Management Objective

As we had already chosen a regular CRUD process for the BGL feature, we chose to store meal data purely through images. This makes it easier for users to report their meals with minimum effort, but it also gave us the added ability to explore the use of images and attachments in conversational interfaces.

Our bot allows the user to easily store images and even automatically extracts time of image stored for future reference. Chatbot interfaces lower the self-reporting barriers for meals in the same way as BGL and this can be built upon in future projects to connect to AI image recognition and provide complete dietary information about the foods photographed. As such, this is a good step forward in dietary management for diabetics through digital diaries.

6.3 Discussion of Reminders Objective

As discussed before in the Implementation, the reminders function is capable of recording any kind of reminder and then using Rails tasks and Scheduler, it is triggered at the right hour to deliver the reminder to the user. Our bot is capable of handling only one-time reminders at the moment and this is again a limitation of the NLP engine.

This feature is commonly implemented in other voice companions like Siri and Facebook M and as such there is more support and documentation for it than the other features, which are unique to our use case. However, reminders are free form and extremely widespread in possibility and due to this the Reminder stories crashed with the General Conversation and Get Help stories repeatedly. We had to implement several checks on the wit engine and add new conditions to each action in our system in order to ensure that the reminders feature always follows the required path.

6.4 Discussion of Current State of Chatbot Technology

This section is broken down into three parts, each discussing one of the three components of our system:

6.4.1 Discussion of Application and Database Server

Existing web server frameworks like Ruby on Rails and Python Flask are highly capable of creating API-only apps that substitute the regular HTML views with HTTP requests. As such, application servers are ready to deliver the performance necessary to create chatbots.

Database solutions like PostgreSQL, similarly, provide high-uptime, concurrent access to multiple users and all the basic features necessary to set up an application for a chatbot.

Both of these technologies are in a ready state for production of chatbots with a number of different options for frameworks and languages for developers to choose from.

6.4.2 Discussion of Chatbot User Interface

The conversational user interface is the key of this project. It is the novelty of being able to talk to computers the same way we talk to humans that makes CIs interesting and exciting to work on, but at the same time the ambiguity in human natural language is what makes them really hard to design.

We chose Facebook Messenger as our platform for the CI, this platform is widely used and as such is intuitive for most users for regular conversation, however we found the interface to still be very clunky in certain regards. For example, it is not suitable to handle large amounts of data at once, and is also limited to text and certain attachments such as images at the moment. While these limitations are not deal-breakers and do not hinder the potential development of a successful chatbot, the biggest drawback of existing platforms for conversational interfaces is that they do not provide any support for UI customization. All bots on a platform must draw from the same limited set of UI elements and this means that despite designers' best efforts, all bots on a platform ultimately look very similar.

The key onus on differentiating your bot from others then becomes on personality development and the choice of language that your bot uses, but again there are not many ways to both make a bot stand out and still be approachable for a wide audience. For example: a bot which uses localized slang to differentiate itself would limit it's audience to those who understand that slang.

6.4.3 Discussion of Natural Language Processing Engines

We used Wit.ai as our natural language processing engine. Wit was a good choice for many reasons as it makes training the bot extremely simple and also gives us many in-built entities to work with as well as a suite of tools for unit testing the bot within the wit environment. The NLP engine also surpassed our expectations for how well it would extract entities and meanings from varied data. However, if we had to pick one limiting factor, as the reason chatbots are not already very popular, it would be the NLP engines.

These engines have great parsing and logical power, but they are currently very nascent in their development due to which there are many bugs and the interface is constantly changing. For example: wit stories are still in beta mode and the training method and entity types changed more than four times throughout the length of our project. There is also a severe lack of documentation, tutorials and best practices as many of the tutorials which do come out become outdated almost immediately due to the ever changing nature of wit.

Wit and other NLP engines function almost entirely in a black box manner with the developer having little to no control over how their data is handled or processed and pre-processing limited to what you can do on your server. Wit is also entirely dependent on the response-request cycle and does not play well with repeated requests, which is a common feature of HTTP APIs.

While wit is currently great for making single-purpose bots that use only a couple of stories, all of these limitations together make wit a challenging platform to build anything more complex than a single-purpose bot. For multi-purpose bot use cases, a custom NLP model trained on a deep learning platform like TensorFlow combined with the Facebook Messenger API would be a better choice than Wit.

Initially we wanted to expand the features of the bot to include advise and motivation for the user as well as personalized and customized feedback, however, wit.ai provided no easy way to train models for these features and in the limited time we had, we could not complete the entirety of these features, so we chose to exclude them from the scope of this project. With future additions, this bot could be made even more efficient for diabetic users.

7 Conclusion

Conversational Interfaces and chatbots have recently gained a rise in popularity due to advancements in Artificial Intelligence technology. In this project we explored the current state of these technologies by implementing a chatbot as a personal caregiver for diabetics that could help them record and store their daily values and remind them to take their medication on time. We designed and implemented a chatbot to accomplish these objectives on the Facebook Messenger platform with help from the Wit.ai NLP engine, two of the most popular technologies in the market. We used a Ruby on Rails Model-View-Controller architecture to combine all of these components. Thanks to the multi-faceted nature of this project, in addition to the web application development and HTTP request-response cycles, we also learnt a great deal about training artificial intelligence models and implementing real life use cases.

Though our project successfully achieves all of the primary objectives to varying extents, there were several challenges and problems that we experienced along the way. One of the major issues was choosing an NLP engine and after conducting are surveys and choosing wit.ai we realized that it might not have been the best choice as it is under constant development. When combined with other advancements in AI, such as image recognition, our project could be further extended to include automatic calorie calculation and other features to assist diabetics even further beyond simple record keeping.

We were also able to partially achieve our secondary objective. It was not fulfilled as, though we experimented with several NLP engines, we only studied the Facebook Messenger platform in depth and as such did not grasp the current state of other CI tools like Slack or Telegram completely. Future studies should explore multiple messaging platforms, which are relevant to various use cases. Our ultimate conclusion regarding the current state of these technologies is that while the messaging platforms are already ready, further improvements on the NLP technology are needed for chatbots to become capable of replacing traditional apps for complex use cases like diabetes management.

8 References

[1] Weizenbaum, J., "ELIZA- A Computer Program For the Study of Natural Language Communication Between Man And Machine," *Communication of the ACM*, Vol. 9, No. 1, 1966.

[2] Huma Shah, "A.L.I.C.E.: an ACE in Digitaland, " *tripleC*, 4(2): 284-292, 2006.

[3] El-Gayar O, Timsina P, Nawar N, Eid W. Mobile Applications for Diabetes Self-Management: Status and Potential. *Journal of Diabetes Science and Technology*. 7(1):247-262, 2013.

[4] Unknown. *HealthTap* [Online]. Available: http://www.healthtap.com Accessed: 2016, September 10.

[5] Unknown. *mySugr* [Online]. Available: http://www.mysugr.com Accessed: 2016, September 10.

[6] Sepah SC, Jiang L, Peters AL, "Long-Term Outcomes of a Web-Based Diabetes Prevention Program: 2-Year Results of a Single-Arm Longitudinal Study", J Med Internet Res 2015;17(4):e92, Available: http://www.jmir.org/2015/4/e92 Accessed: 2016, September 10.

[7] Unknown. *"api.ai Joins google!"* Available: https://api.ai/blog/2016/09/19/api-ai-joining-google/ Accessed: 2016, September 10.

[8] Guizzo, E. (2011). IBM's Watson Jeopardy computer shuts down humans in final game. *IEEE Spectrum*, *17*.

[9] "Uber On Messenger | Request a Ride While in Messenger," 16 Dec 2015. [Online]. Available: https://newsroom.uber.com/messengerlaunch/ Accessed: 2016, January 10.

[10] "Hi Poncho," [Online]. Available: https://www.facebook.com/hiponcho/ Accessed: 2016, January 10.

[11] "1-800-Flowers.com," [Online]. Available: https://www.facebook.com/1800flowers Accessed: 2016, January 10.

[12] "WHO Statistics on World Diabetes". Available: http://www.who.int/mediacentre/factsheets/fs312/en/ Accessed: 2016, January 10.

[13] American Diabetes Association. "Standards of Medical Care in Diabetes--2012. Diabetes Care." 2012;35(Suppl 1):S11–63.

Appendix A: Meeting Minutes

Note: The team was only formed in September the initial literature survey was done by Shivang over the summer. As such, meetings so far were held in September

8.1 Minutes of the 1st Project Meeting

Date:	September 11, 2016
Time:	3:00 pm
Place:	Room 3536
Present:	Shivang, Allen, Prof. Sung Kim
Absent:	None
Recorder:	Shivang

1. Approval of minutes

As this was the first group meeting, there were no minutes to approve.

2. Report on progress
   - 2.1 The team members are aware of all the Final Year Project instructions.
   - 2.2 Shivang has conducted research on the topic and relevant technologies and will share with the team

3. Discussion items
   - 3.1 The goal of the project is to make a chatbot with Facebook messenger as the platform. The further aims were also discussed.
   - 3.2 The AI engine used by the project needs to be decided as well as the software stack for development.
   - 3.3 An application or use case needs to be decided for the technology. We are currently debating between healthcare and education based uses.
   - 3.4 We discussed the possibility of using Deep Learning and NLP for the processing of data.

4. Goals for the coming week

4.1 Allen will catch up on the materials he has missed.

4.2 Shivang will research deep learning technologies we could use and AI engines.

4.3 All group members will think of applications and use cases for the technology.

4.4 All group members will think about ways to develop an efficient and user friendly system.

5. Meeting adjournment and next meeting

The meeting was adjourned at 4:00 pm.

The next meeting will be at 4:00 on September 21st at room 3536.

8.2 Minutes of the 2nd Project Meeting

Date:	September 21, 2016
Time:	3:30 pm
Place:	Science Commons, Room 3536 (at 4pm)
Present:	Shivang, Allen, Prof. Kim (at 4pm)
Absent:	None
Recorder:	Shivang

1. Approval of minutes

The minutes of last meeting were approved without amendment.

2. Report on progress

2.3 All group members have studied the messenger platform.

2.4 Shivang shared the sample app he set up with Prof. Kim's instructions and discussed its working.

2.5 It was determined the IBM Watson would be a superior told to use due to its stability.

2.6 Allen studied the materials from the summer to understand the project better.

2.7 All group members came up with different ideas for applications.

2.8 The meeting was shifted to Prof. Kim's office at 4pm

3. Discussion items
   3.1 The team discussed creating a healthcare chatbot vs. an education chatbot, ultimately it was decided the healthcare chatbot would be made.
   3.2 The team confirmed the software stack and development tools to be used in the project.
   3.3 Prof. Kim approved both the topics and suggested we start building soon to get a better understanding as this is cutting edge technology.

4. Goals for the coming week
   4.1 Finish the proposal report.
   4.2 Set up a test server to experiment with different AI engines.
   4.3 The team will work on the schedule and choose a regular meeting time.

5. Meeting adjournment and next meeting

The meeting was adjourned at 5:00 pm.

The date and time of the next meeting will be set later by e-mail.

8.3 Minutes of the 3rd Project Meeting

Date:	October 21, 2016
Time:	2:30 pm
Place:	Learning Commons
Present:	Shivang, Allen
Absent:	None
Recorder:	Allen

1. Approval of minutes

The minutes of last meeting were approved without amendment.

2. Report on progress
   2.1 Finished the proposal report.
   2.2 Set up the test server to experiment with different AI engines.
3. Discussion items

3.1 The team discussed design of the Reminder feature. Ultimately the team decided to set a cronjob to remind user after Reminder was saved.

3.2 The team assigned a GANTT chart for ork distribution over the exam period and holidays.

4. Goals for the coming week

4.1 Shivang implement the Reminder feature on the test server.

4.2 Allen will research on chatbot UI to get better understanding of materials.

5. Meeting adjournment and next meeting

5.1 The meeting was adjourned at 3:30 pm.

5.2 Next meeting will be at Oct. 30th, time will be decided later

8.4 Minutes of the 4th Project Meeting

Date:	October 30, 2016
Time:	6:30 pm
Place:	Learning Commons
Present:	Shivang, Allen
Absent:	None
Recorder:	Shivang

1. Approval of minutes

The minutes of last meeting were approved without amendment.

2. Report on progress

2.1 Implemented the Reminder feature on test server with Heroku Scheduler.

2.2 Allen tried out some tutorials to better understand the system design.

3. Discussion items

3.1 The team discussed how to test the Reminder feature more efficiently.

3.2 The Heroku Scheduler is limited in free tier and as such the tasks have to be chosen accordingly.

3.3 We discussed the possibility of entering the President's Cup

4. Goals for the coming week

4.1 Allen will test the Reminder feature further.

4.2 All team members think of solutions for different types of reminders.

4.3 Create and submit President's cup proposal

5. Meeting adjournment and next meeting

5.1 The meeting was adjourned at 7:00 pm.

5.2 The date and time of the next meeting will be set later by text.

8.5 Minutes of the 5th Project Meeting

Date: November 21, 2016

Time: 12:10 pm

Place: Learning Commons

Present: Shivang, Allen

Absent: None

Recorder: Allen

1. Approval of minutes

The minutes of last meeting were approved without amendment.

2. Report on progress

2.1 The Reminder feature was tested.

2.2 The president's cup proposal was completed and submitted

3. Discussion items

3.1 The team discussed design of the Advise feature. Ultimately the team decided to provide two kinds of advice: Bot related and Diabetes related advice.

3.2 We put the implementation on hold to better understand the design

4. Goals for the coming week

4.1 Shivang design the system architecture and flow

4.2 Allen design ER model

5. Meeting adjournment and next meeting

5.1 The meeting was adjourned at 1:00 pm.

5.2 The date and time of the next meeting will be set later by e-mail.

8.6 Minutes of the 6th Project Meeting

Date: November 30, 2016

Time: 2:30 pm

Place: Science Commons

Present: Shivang, Allen (Late)

Absent: None

Recorder: Shivang

1. Approval of minutes

The minutes of last meeting were approved without amendment.

2. Report on progress
   - 2.1 Designed ER model
   - 2.2 Designed System Architecture
   - 2.3 Discussed possible inputs with medical experts over phone
3. Discussion items
   - 3.1 The team discussed how to implement the Advise feature more efficiently.
   - 3.2 The recording of data needs complicated CRUD actions, it was decided to skip the Update action entirely
   - 3.3 Rails and PostgreSQL were finalized as the tools\
   - 3.4 Shivang met with Prof. Kim to discuss Presidents' cup and monthly reports

4. Goals for the coming week
   - 4.1 Allen develop part of the Advise feature.
   - 4.2 All team members think of BGL Recording feature and other CRUD actions

5. Meeting adjournment and next meeting
   - 5.1 The meeting was adjourned at 3:30 pm.
   - 5.2 The date and time of the next meeting will be set later by text.

8.7 Minutes of the 7th Project Meeting

Date: December 14, 2016

Time: 10:30 am

Place: Science Commons

Present: Shivang, Allen

Absent: None

Recorder: Shivang

1. Approval of minutes

The minutes of last meeting were approved without amendment.

2. Report on progress
   2.1 Some design work, but not much progress during exams.
3. Discussion items
   3.1 The team discussed design of the BGL Recording feature. Ultimately the team decided to save BGL value, time and type of recording to database and use only images for meals.
   3.2 A future goal was set to perform visual analysis on these images automatically.
4. Goals for the coming week
   4.1 Shivang implement the BGL Recording feature.
   4.2 Finish all pending goals from last week
5. Meeting adjournment and next meeting
   5.1 The meeting was adjourned at 11:00 am.
   5.2 The date and time of the next meeting will be set later by e-mail.

8.8 Minutes of the 8nth Project Meeting

Date: December 21, 2016

Time: 12:30 pm

Place: Science Commons

Present: Shivang, Allen

Absent: None

Recorder: Allen

1. Approval of minutes

The minutes of last meeting were approved without amendment.

2. Report on progress
   2.1 Implemented part of the BGL Recording feature.
3. Discussion items
   3.1 The team planned out how to work over the winter holidays.
4. Goals for the coming week
   4.1 Allen will implement the BGL Recording feature further
   4.2 All team members think of Meal Recording feature implementation.

5. Meeting adjournment and next meeting
   5.1 The meeting was adjourned at 5:30 pm.
   5.2 The date and time of the next meeting will be set later by e-mail.

8.9 Minutes of the 9th Project Meeting

Date:	January 24, 2017
Time:	2:50 pm
Place:	Hall 5 Common Room
Present:	Shivang, Allen
Absent:	None
Recorder:	Shivang

1. Approval of minutes

The minutes of last meeting were approved without amendment.

2. Report on progress
   2.1 The controllers and models were set up to implement the CRUD functions

2.2 Training was performed in isolation and a new training system was developed. The Slot based CRUD operations will now go first.

2.3 The system setup is complete and a testing suite is also being developed to test on the live deployment.

2.4 Some acceptance test users were found by Shivang in India through doctors

3. Discussion items

3.1 The team discussed the progress over the winter and decided a way to move forward with a modified timeline to account for some small delays.

4. Goals for the coming week

4.1 Finish all designing tasks and complete recording feature implementation.

4.2 Finish new version of reminder feature.

5. Meeting adjournment and next meeting

5.1 The meeting was adjourned at 4:00 pm.

5.2 The date and time of the next meeting will be set later by e-mail.

8.10 Minutes of the 10th Project Meeting

Date:	February 14, 2017
Time:	1:00 pm
Place:	The BASE
Present:	Shivang, Allen
Absent:	Prof. Kim
Recorder:	Shivang

1. Approval of minutes

The minutes of last meeting were approved without amendment.

2. Report on progress

2.1 Implemented the Meal Recording feature and set up new tasks for reminder feature.

2.2 Block in reminder feature CRUD.

3. Discussion items

3.1 The team discussed how to debug the reminder feature while waiting for Prof. Kim.

3.2 We finished debugging the glitch and are back on track.

4. Goals for the coming week

4.1 Allen test the Meal Recording feature.

4.2 All team members think of integrating all features and how to perform integration testing.

5. Meeting adjournment and next meeting

5.1 The meeting was adjourned at 2:00 pm.

5.2 The date and time of the next meeting will be set later by e-mail.

8.11 Minutes of the 11th Project Meeting

Date:	February 28, 2017
Time:	2:00 pm
Place:	Skype
Present:	Shivang, Allen
Absent:	
Recorder:	Allen

1. Approval of minutes

The minutes of last meeting were approved without amendment.

2. Report on progress

2.1 Allen tested the Meal Recording feature.

2.2 The reminder feature works but has some small glitches

3. Discussion items

3.1 The team discussed how to integrate all features and how to perform integration testing.

3.2 We finished debugging the Meal Recording feature.

3.3 Development strategy was changed to feature wise instead of component wise

4. Goals for the coming week

4.1 Shivang integrate features and Allen test the integration of new features.

4.2 All team members think of eliminating conflict between different features.

5. Meeting adjournment and next meeting

5.1 The meeting was adjourned at 4:00 pm.

5.2 The date and time of the next meeting will be set later by e-mail.

8.12 Minutes of the 12th Project Meeting

Date:	March 7, 2017
Time:	3:00 pm
Place:	Learning Commons
Present:	Shivang, Allen
Absent:	
Recorder:	Allen

1. Approval of minutes

The minutes of last meeting were approved without amendment.

2. Report on progress

2.1 Integrated BGL recording feature and Meal recording feature.

2.2 This integration was tested.

3. Discussion items

3.1 The team discussed how to integrate with other features.

3.2 We finished debugging the integration.

4. Goals for the coming week

4.1 Shivang will continue to making read features and Allen test the meal and bgl feature.

4.2 All team members think of eliminating conflict between different features.

5. Meeting adjournment and next meeting

5.1 The meeting was adjourned at 4:30 pm.

5.2 The date and time of the next meeting will be set later by whatsapp.

8.13 Minutes of the 13th Project Meeting

Date:	March 21, 2017
Time:	1:30 pm
Place:	Skype
Present:	Shivang, Allen
Absent:	
Recorder:	Shivang

1. Approval of minutes

The minutes of last meeting were approved without amendment.

2. Report on progress

2.1 Integrated Tutorial and general conversation feature

2.2 Old features were re-tested

3. Discussion items

3.1 The team discussed how to integrate with other features.

3.2 We finished debugging the integration.

4. Goals for the coming week

4.1 Shivang is in India so will work on meal read feature as he gets time.

4.2 Allen will test the wit engine to fix bugs.

4.3 All team members think of eliminating conflict between different features.

5. Meeting adjournment and next meeting

5.1 The meeting was adjourned at 3:00 pm.

5.2 The date and time of the next meeting will be set later by whatsapp.

8.14 Minutes of the 14th Project Meeting

Date: April 7, 2017

Time: 2:30 pm

Place: The BASE

Present: Shivang, Allen

Absent:

Recorder: Allen

1. Approval of minutes

The minutes of last meeting were approved without amendment.

2. Report on progress

   2.1 Integrated recording features and all save/read feature.

   2.2 This delete feature was tested.

3. Discussion items

   3.1 The team discussed how to fix bugs in code

   3.2 We finished debugging the integration.

4. Goals for the coming week

   4.1 Finish any remaining code and begin work on new sections of Final report

5. Meeting adjournment and next meeting

   5.1 The meeting was adjourned at 4:30 pm.

   5.2 The date and time of the next meeting will be set later by whatsapp.

8.15 Minutes of the 15th Project Meeting

Date: April 14, 2017

Time: 3:00 pm

Place: Seating area Outside LT-A

Present: Shivang, Allen

Absent:

Recorder: Shivang

1. Approval of minutes

The minutes of last meeting were approved without amendment.

2. Report on progress

   2.1 Integrated most features, bug fixing

   2.2 Report work is almost complete, poster needs to be started.

3. Discussion items

   3.1 The team discussed poster design and presentation practice

   3.2 Discussed how to finish the report and prepare presentation demo

4. Goals for the coming week

   4.1 Finish writing the report and making the poster and submit all reports to fyp system.

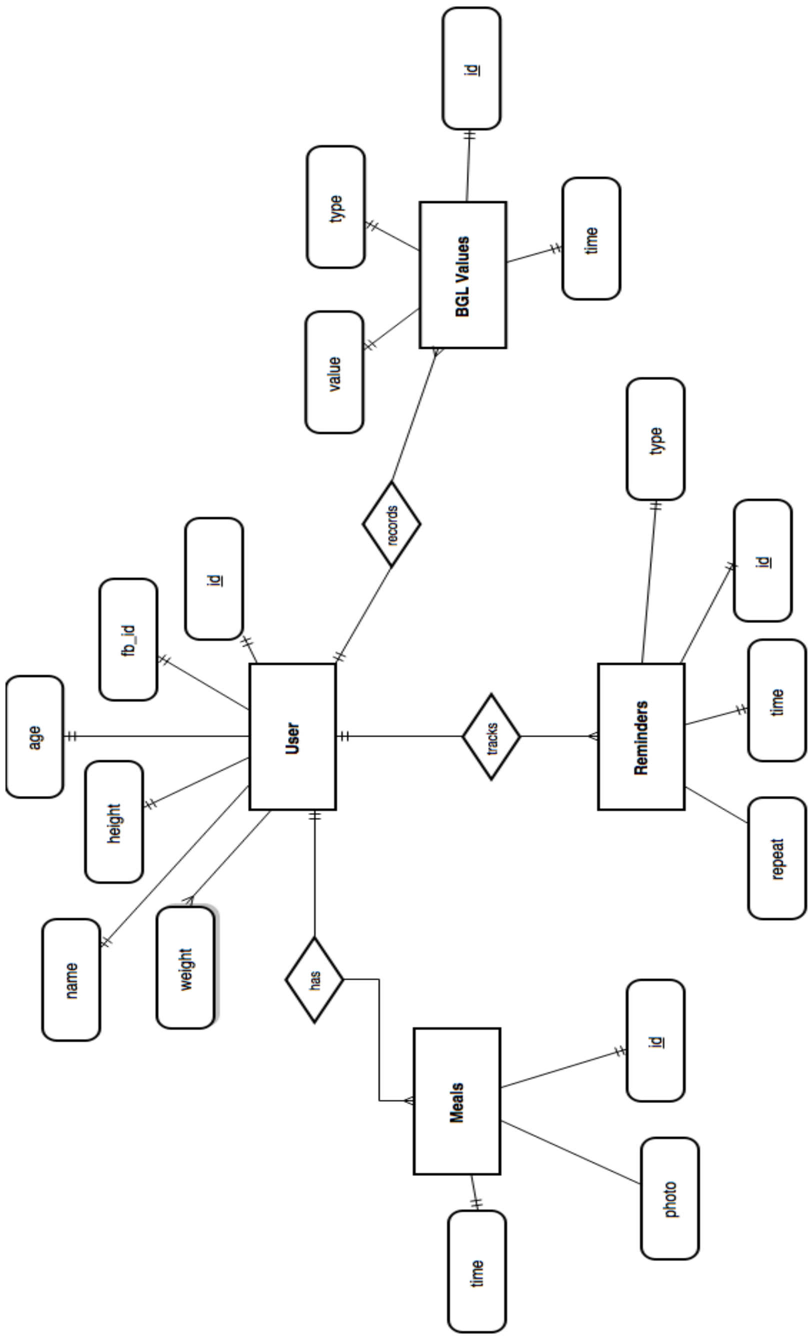
   4.2 Must finish debugging any localhost errors for demonstration furing presentation

5. Meeting adjournment and next meeting

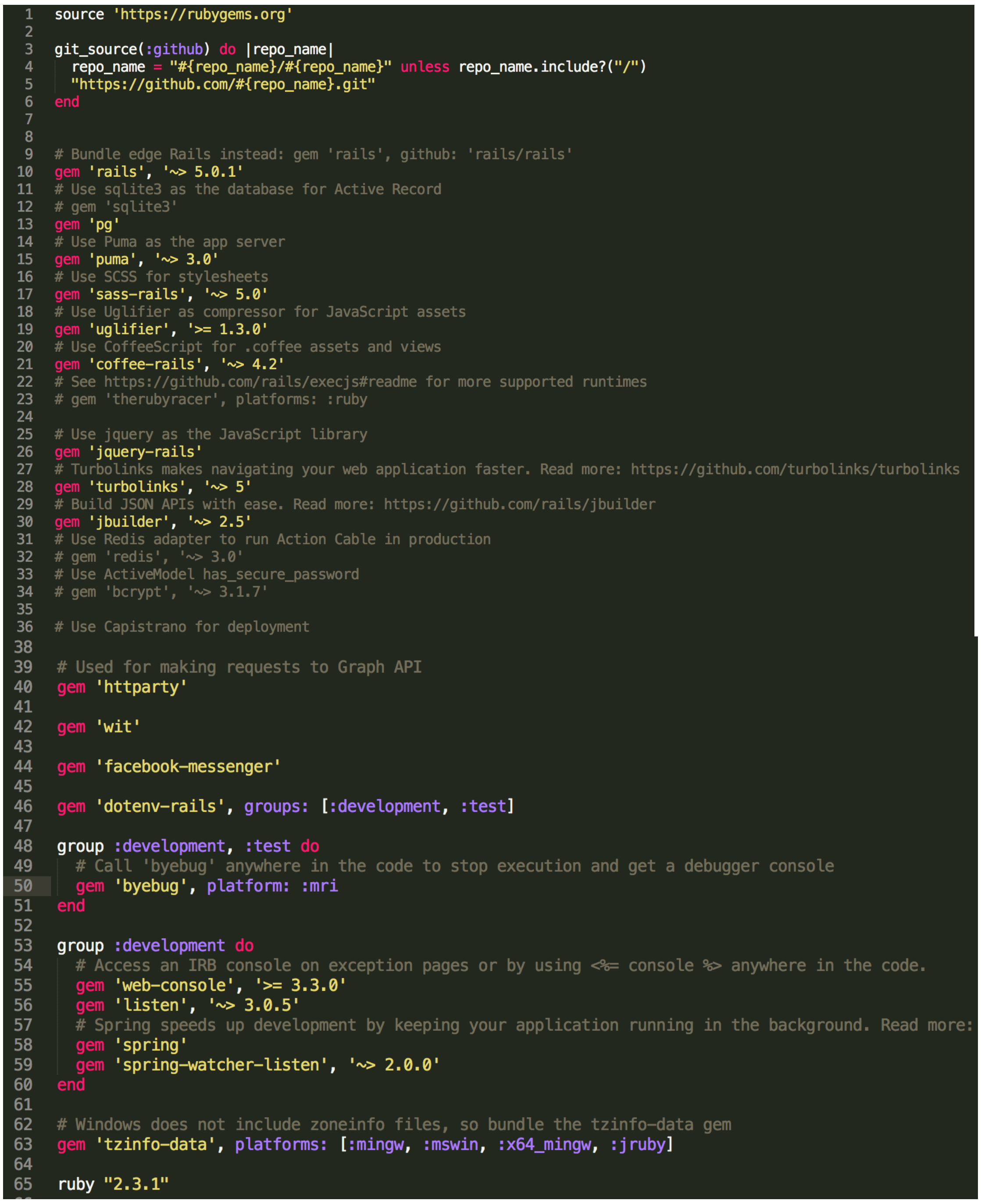
   5.1 The meeting was adjourned at 5:00 pm.

The date and time of the next meeting will be set later.

Appendix B: ER Diagram



Appendix C: Gemfile



```ruby
source 'https://rubygems.org'

git_source(:github) do |repo_name|
  repo_name = "#{repo_name}/#{repo_name}" unless repo_name.include?("/")
  "https://github.com/#{repo_name}.git"
end


# Bundle edge Rails instead: gem 'rails', github: 'rails/rails'
gem 'rails', '~> 5.0.1'
# Use sqlite3 as the database for Active Record
# gem 'sqlite3'
gem 'pg'
# Use Puma as the app server
gem 'puma', '~> 3.0'
# Use SCSS for stylesheets
gem 'sass-rails', '~> 5.0'
# Use Uglifier as compressor for JavaScript assets
gem 'uglifier', '>= 1.3.0'
# Use CoffeeScript for .coffee assets and views
gem 'coffee-rails', '~> 4.2'
# See https://github.com/rails/execjs#readme for more supported runtimes
# gem 'therubyracer', platforms: :ruby

# Use jquery as the JavaScript library
gem 'jquery-rails'
# Turbolinks makes navigating your web application faster. Read more: https://github.com/turbolinks/turbolinks
gem 'turbolinks', '~> 5'
# Build JSON APIs with ease. Read more: https://github.com/rails/jbuilder
gem 'jbuilder', '~> 2.5'
# Use Redis adapter to run Action Cable in production
# gem 'redis', '~> 3.0'
# Use ActiveModel has_secure_password
# gem 'bcrypt', '~> 3.1.7'

# Use Capistrano for deployment

# Used for making requests to Graph API
gem 'httparty'

gem 'wit'

gem 'facebook-messenger'

gem 'dotenv-rails', groups: [:development, :test]

group :development, :test do
  # Call 'byebug' anywhere in the code to stop execution and get a debugger console
  gem 'byebug', platform: :mri
end

group :development do
  # Access an IRB console on exception pages or by using <%= console %> anywhere in the code.
  gem 'web-console', '>= 3.3.0'
  gem 'listen', '~> 3.0.5'
  # Spring speeds up development by keeping your application running in the background. Read more:
  gem 'spring'
  gem 'spring-watcher-listen', '~> 2.0.0'
end

# Windows does not include zoneinfo files, so bundle the tzinfo-data gem
gem 'tzinfo-data', platforms: [:mingw, :mswin, :x64_mingw, :jruby]

ruby "2.3.1"
```

Appendix D :Project Planning

Distribution of Work

Task	Shivang	Allen
Do the Literature Survey	●	○
Analyze existing bots on Messenger	○	●
Collect datasets from doctors and online sources	●	●
Design the database	○	●
Design the conversational UX	●	○
Design the Messenger user interface	●	○
Design web server architecture	●	○
Build the database	○	●
Set-up web server	●	●
Develop chatbot UI	●	●
Train artificial intelligence	●	●
Integrate the server, engine and UI	●	○
Test the chatbot interface	○	●
Test the web server and database connection	●	●
Test the AI engine	○	●
Perform Integration Testing	●	●
Evaluate results	●	●
Write the Proposal	●	○
Write the Progress Report	●	●
Write the Final Report	●	○
Prepare for the Presentation	●	●
Design the Project Poster	●	○

● In-charge

GANTT Chart

Task	July	Aug	Sep	Oct	Nov	Dec	Jan	Feb	Mar	Apr
Do the Literature Survey	■	■								
Analyze existing bots on Messenger		■	■							
Collect datasets from doctors and online sources				■	■	■				
Design the database				■	■	■				
Design the conversational UX					■	■				
Design the Messenger user interface				■	■	■	■			
Design web server architecture				■	■	■	■	■		
Build the database						■	■			
Set-up web server						■	■	■		
Develop chatbot UI							■	■		
Train artificial intelligence					■	■	■	■	■	
Integrate the server, engine and UI								■	■	
Test the chatbot interface								■	■	■
Test the web server and database connection								■	■	■
Test the AI engine								■	■	■
Perform Integration Testing									■	■
Evaluate results									■	■
Write the Proposal			■							
Write the Progress Report								■		
Write the Final Report									■	■
Prepare for the Presentation										■
Design the Project Poster										■

Appendix E: Hardware and Software Requirements

Hardware

Development System:	Computer with Linux/macOS or later
Minimum Display Resolution:	1024 * 768 with 16 bit color
Testing Devices:	Android and iOS Mobile Phone

Software

PostgreSQL	Database server
Ruby	Main programming language
Rails	Web application framework
Google Chrome and Safari	Browsers
Adobe Photoshop, Illustrator	Graphic design
InvisionApp, Chatfuel	Prototyping
Git	Version control
ngrok	Local testing deployment

Cloud

Heroku/Digital Ocean	Live deployment
Wit.AI Console	For AI training