CHATBOT DEVELOPMENT

Submitted in part fulfillment for the degree of
B.Sc. in Science in Computing
Institute of Technology Blanchardstown
Dublin, Ireland

By
Daire Homan B00029598
Brian Kelly B00082716
Aaron Ward B00079288

2017

Declaration

Abstract

A chatbot is a computer program that mimics conversation with a human using artificial intelligence. The purpose of this project is to investigate the appropriate technologies involved in the implementation of a chatbot. With the advent of big messaging platforms opening up their applications for developers to integrate their chatbots, it presents interesting opportunities for developers to deploy their very own chatbot applications and have them available to use on some of the worlds largest and industry-leading messaging platforms. An exploration of common software engineering practices centred around chatbot development, the deployment of chatbot applications and the appropriate development stack required to bring a chatbot to fruition will be covered in this report.

Keywords: Artificial intelligence; Chatbot; Machine learning; Natural language processing;

Acknowledgements

We would like express sincere gratitude to our project supervisor Dr. Matt Smith, who supported and advised us throughout the research and development of this project.

Table of Contents

1 Introduction

Chatbots, whilst not a new technology, have gained considerable notoriety in recent times. This upsurge in interest from developers has come in lieu of some of the worlds largest messaging platforms opening up their applications to developers to deploy their chatbots and have them exist as part of their application ecosystem. This project will explore these new opportunities and aim to give a comprehensive oversight into what software development decisions to consider in order to implement a chatbot application. The report will begin by analysing relevant literature in this area and by providing the reader with the necessary elements in order to understand exactly what a chatbot is, their history, their current application use-cases today and a thematic overview of why they are relevant, accessible to developers and why chatbots present a positive learning opportunity for this project. Additionally, this report will outline the current methodologies and types of technology that are both required and recommended by developers in order to successfully implement a chatbot. The system requirements and specification section aims to provide a distinct vision on how the team arrived at a feature set for the chatbot and how these decisions where supported by surveys carried out. Furthermore, this section will provide an overview and explanation of the technologies and tools chosen by the developers supported by reasons as to why the technologies chosen were the adequate choice for this application. The system design section will cover how the system was designed architecturally and , because of the applications distributed nature, how each participant within the system exchanges instructions and data with the support of common application use cases. The implementation section will explore how the application was developed in areas in which where integral to the overall functionality of the application supported by code examples. Moreover, the testing and evaluation section will outline how each node in the system was tested to affirm its functionality and what tools were used to strengthen this process. Finally, this study is concluded with some closing remarks and key areas of interest should further work commence on this project.

2 Literature Review

The proceeding literature review will highlight several areas of notoriety regarding chatbots. Relevant journal articles and web articles will be chosen to review in order to establish a core understanding of the current state of the advancements in natural language processing (NLP), machine learning, artificial intelligence (AI) and the new opportunities for developers in light of these advancements. Whilst there are an abundance of resources available to research the history of NLP, machine learning and AI, there is a scarcity of academic resources centred around chatbots because of their recent emergence to the app market. In light of this, web articles from credible websites will be used to formulate an opinion on their current position in the eyes of developers and their projected success. In order to provide a foundation of understanding to the reader initially, the specifics of what a chatbot is and its origins will be explored. Additionally, current developments in the field of AI, virtual-assistants and current applications of chatbots will also be analysed and supported by current literature. Finally, an analysis of the literature reviewed will be delivered in a thematic way, whereby insights will be provided as to what the relevance of chatbot technologies are today, the increase in accessibility to tools for developers and the opportunities that reveal themselves for this project.

2.1 What Is a Chatbot?

Chatbots are computer programs that mimic conversation with people using artificial intelligence. It is a service powered by rules and sometimes artificial intelligence (Wong, 2016). Usually, the common use case for end-users using a text-based conversational interface would be when they are messaging each other on popular client devices like their phone or laptop. Now users can have similar conversations, with a piece of software that can intelligently respond to them in a human-like manner. This type of technology can be driven by basic parsing of an input to analyse for keywords and output a message, or the chatbot can be implemented in such a way whereby the input is parsed using a NLP engine that can extract message components like context, ambiguity and sentiment. The NLP engine can then statistically analyse the input and provide an output to the end-user based on the result it deems most accurate. Chatbots can also learn by implementing automated machine learning algorithms. A chatbot can perform peculiarly when queried by a high volume of inputs, and these inputs are difficult for developers to preempt during development (Wong, 2016). In this situation, whereby the chatbot does not understand an input, intelligent algorithms can be used to analyse the misunderstood input and suggest the most likely output based on the data gathered from parsing the initial input and previous conversation histories. There are several ways that these types of technology can come to fruition, however, the core use case is always similar. Based on the input provided from the user, the chatbot should always intelligently deliver an appropriate output.

2.2 Origins of the Chatbot

Joseph Weizenbaum, a professor at Harvard MIT, created the first chatbot application in 1966. Weizenbaum's implementation applied rule-based instructions and a now-primitive NLP engine that could decipher intents presented by open-ended inputs. Weizenbaum's work triggered philosophical interest around the implications of intelligent computer-based agents (Dale, 2016). In the wake of Weizenbaum's breakthrough, several other advances where made in the field of NLP and algorithmic enhancements provided more accurate and human-like responses. Several breakthroughs have been made since, most notably, that of IBM's 'Watson' project in 2006. The Watson implementation won a game of Jeopardy against two of the world's leading Jeopardy players and in the wake of this, this point in time became a defining moment in the timeline of NLP's progression (Best, 2017). This milestone signified the power that a piece of software with NLP capabilities could have and the potential applications it could have in the future. These intelligent agents solidified NLP as a viable route to building intelligent software that did not have to be explicitly instructed to compute something, rather, the software could decide for itself based on the algorithmic analysis of its inputs.

Further implementations began to emerge, mainly amongst the bigger software companies and corporations, most notably those of the Big Four. Apple's Siri, Microsoft's Cortana, Amazon's Alexa and Google's Assistant all came to fruition in lieu of Watsons success. These technologies harnessed the power of NLP through textual interfaces and intelligent voice recognition consistently with one core objective: to provide a richer and more interactive experience for their end-users. These technologies are now implemented amongst all of the aforementioned company's products and are a standard feature in today's end-user experience.

2.3 Current Developments in Industry

Whilst these technologies are accessible to end-users through their client devices, accessibility for developers to build applications with these powerful NLP capabilities has only recently being made available. Until recently, if a developer sought these powerful NLP capabilities for their own application they would have to implement the NLP algorithms and also build the applications front-end to interact with their custom language API. This is a lengthy and difficult task for developers that do not have access to Apple's, Microsoft's, Amazon's or Google's proprietary AI software API's (Evans, 2017). Now, with the emergence of software development frameworks that provide open-API's, programmers can access powerful NLP engines for their applications. A developer can develop and produce their own implementation of an intelligent agent. Furthermore, popular messaging platforms have also provided SDK's for developers to build chatbot and release them on their platform. These two defining changes within the industry has seen a rapid advancement in the field of chatbot development and subsequently created significant opportunity in the realms of e-commerce and added a new layer to mobile application capabilities (Dale, 2016).

2.4 Current Applications

Chatbot applications today are primarily deployed to a popular messaging platform in order to sit amongst their intended target-market of users. This not only increases penetration across operating systems and high-availability for the application, but also, provides a vehicle for the application to grow its user-base exponentially faster. For example, Facebook Messenger alone has over one billion users and provides an ideal platform for developers to deploy their chatbot and have it easily accessible to anybody that has a Facebook Messenger client on their device (Dale, 2016). Existing implementations can provide services to users whereby they can provide information instantly based on the question that the chatbot is prompted. In contrast to a standard application, whereby the user would usually have to navigate to the appropriate interface in the application or search-by-text and be re-directed to another interface to obtain the information they were seeking. It is still unclear whether chatbot will prove a more advantageous option for users to obtain their data (Deo, 2016). Chatbots are still early in the stages of user-adoption and their success is also heavily dependent on users diverting from their usual method of obtaining information which may prove to be a larger hurdle than what was once surmised (Ranger, 2016). However, beyond the obvious implementations of these particular types of software, several interesting use cases have been demonstrated and have yielded positive results. One application in particular 'Robot Lawyer' has successfully aided its users by allowing them to appeal and/or overturn parking tickets that they have been issued. Furthermore, this chatbot is also capable of providing important information to refugees seeking asylum (Cresci, 2017). This type of application may see success quicker than a standard implementation that provides users with answers to FAQ's. The positive publicity that a chatbot like this receives may also propel the status of chatbots and give them recognition amongst the average end-user.

2.5 Concluding Analysis of Literature Reviewed

The proceeding sections aim to clarify several themes that emerged whilst reviewing journals and web articles for this project. As this project is a prototype implementation of a chatbot, it is beneficial to the project members and the reader to have a clear and defined vision for the final implementation and to identify the core learning components that can be gained from developing the chatbot and, should further work on this project be deemed necessary, what advancements can be brought to fruition in light of this.

2.5.1 Relevance of Chatbot Technologies

Judging from the articles that have been reviewed, there is clear cause for interest in this emerging technology. Although this technology is not new in regard to research and its origins reach back as far as 1966, there has been a notable surge in interest as of late owing namely to the growth of NLP development frameworks and tools. A sub-reason to this explosive new-found relevance can also be attributed to the large messaging platforms providing SDK's for developers. Although it is hard to gauge whether chatbots will take over traditional applications in this space, little argument can be made against the up-surge in publications on them and their backing received from the big messaging platforms. Based on common consensus between articles reviewed for this paper, ultimately, time and user adoption will dictate chatbots validity in the app marketplace of tomorrow. However, in light of doing research, they present themselves with ample opportunity to explore the world of NLP and AI from a novice developer's perspective.

2.5.2 Accessibility to Tools

With the advent of NLP platforms being open to developers to use their API's it is now possible for novice developers to harness the computational power of proven NLP algorithms and build their own intelligent agents. Furthermore, developers can implement NLP features into their applications without having to have an in-depth knowledge of the lower-levels of developing software that can handle complex linguistic queries. These NLP API's, because of the surge in chatbot popularity, are mostly free to use and provide an ideal alternative to developers who do not want to 're-invent the wheel'. This shift has provided a broad spectrum of opportunity to developers who may have once been prohibited from implementing applications because of a lack of resources/knowledge in this area.

2.5.3 Opportunities for the Project

This newly introduced accessibility to NLP engines and popular messaging platforms provides ample reasons to explore these platforms during this project for the purposes of gauging their accessibility and performance. Moreover, these API's will, most importantly, provide the team members with an opportunity to develop the chatbot in-line with the API's specifications, providing the participants with experience of querying web services to further enhance the applications capabilities. Also, the participants will gain experience with deciding which platform to deploy the chatbot application to. Knowledge will be gained in regards to deciding which platforms that best-support the technologies and programming languages that the team members are familiar with. Above all else, experimentation with new types of software, especially AI driven implementations, can provide the participants with a fresh perspective on software development in general and elevate their understanding of designing a distributed application that interacts with several nodes to fulfil its intended functionality.

3 Method

The purpose of the subsequent section is to provide a detailed understanding and comprehension of the technologies and tools commonly used whilst developing a chatbot. This will encompass a high-level description and illustration of the many different technologies used to successfully implement this type of the software. Furthermore, it will also highlight the core services, architectures and programming languages used in the development and creation of chatbots. In addition, exploration of how the conversation flow aspect of this technology is achieved with NLP will be investigated. Additionally, there are some considerations for different technologies that will be highlighted in this selection. For clarity and to gain a better appreciation of all the aforementioned aspects that goes into the development of chatbots, each core feature (technology) will be broken up into its own section. These sections will include the platform that the bot is hosted on, the platform as a service (PaaS), version control platform where the application will be developed, and other useful tools that will aid the participating team members.

3.1 Messaging Platform

One of the core considerations in the development of a chatbot application is the platform that the chatbot will be hosted on. There is a superabundance of messaging platforms accessible to developers. And now big companies such as Facebook, WhatsApp, Telegram and WeChat have allowed developers to integrate chatbots into their messenger platforms. Thus, making accessibility to millions of users effortlessly. In addition to this, they provide considerable amounts of documentation to detail what their platforms can provide. However, each platform has its own interface, and different user base that are receptive to different types of products and experiences. Therefore, selecting the appropriate platform depends on the type of chatbot being developed and the targeted demographic the application is intended for.

3.2 NLP Tools

An important aspect of a chatbot application is its ability to comprehend what the user's messages intents and contexts are when the user and the chatbot are conversing. This is achieved in chatbot technologies through the use of Natural Language Processing (NLP). NLP employs computational techniques for the purpose of learning, understanding, and producing human language content through spoken language and text (Hirschberg et Manning, 2015). This type of technology provides the capability of understanding the user's intent which drives the actions of the chatbot. As aforementioned, NLP is an integral component of a chatbot, and in light of this, there has been many developments in the accessibility to NLP tools and platforms. There are now a number of services that provide NLP as a service, platforms such as Api.ai, Google Natural Language API and Wit.ai. With the creation of the above-mentioned services, bot developers can now easily implement NLP into chatbot architectures. However, there are a number of considerations in the deliberation of selecting one of these services. These may include the price, limitations and quality of the service.

3.3 Deployment Options

A deployment service is another pre-development decision to be considered. In selecting such a service there are a number of considerations that need to be considered which entail cost, CPU, RAM scalability and complexity. However, there are a number of deployment services that offer a Platform as a Service (PaaS) with sufficient CPU specification which is easy to use and free of cost. Deployment facilities such as Microsoft Azure Bot Service, Google App Engine, Heroku and Docker provide this type of service. PaaS is a cloud based technology that delivers a platform as well as hardware, software and tools to allow developers to easily deploy and run chat applications over the internet (Interoute, 2017). Chatbot developers can take advantage and use the PaaS environments from conception, development and creation of their chatbot applications.

3.4 Programming language

A fundamental aspect to consider before developing a chatbot is the programming language it will be implemented in. The choice will be contingent on a number of factors. This includes the messaging platform (Facebook Messenger, WhatsApp etc.) in which the chatbot is hosted as each of the aforementioned platforms have their own technology compatibility guidelines for implementing this type of technology. However, chatbots housed on messenger platforms like Facebook Messenger and Telegram can be developed in any language due these platforms being language agnostic in comparison to another platform like Slack, in which chatbots must be developed in Python. In addition, the knowledge and preference of the developer in a particular language and the accessibility of the libraries suitable to the chatbot's design must be considered. In addition, server side languages such as PHP, Python and Node.JS (JavaScript) are typically used to write chatbots as they are popular server-side web programming technologies.

3.5 Database

A database is also a significant element in the development and creation of a chatbot. There are several considerations to be reflected in terms of what type of database to be incorporated within the chatbot architecture. These can include a SQL database or a NoSQL database. The difference between these types of architectures are SQL databases are table based while NoSQL databases are document based. This means that SQL databases embody data in form of tables which encompasses rows of data whereas NoSQL databases are the collection of documents or wide-column stores (Thegeekstuff, 2017). Both database infrastructures can be incorporated into a chatbot. However, in the deliberation of selecting the appropriate database model, there are a number of considerations to be contemplated. For instance, what type of data model does each hosting platform support, the scalability, complexity and cost of each architecture.

3.6 Version Control

Version control software is not a key element nor an integral part in the creation of chatbots however, they are in the collaborative development of software amongst a team. A version control system is a type of software or tool to track files or modifications made to a code base or project and can be shared via a centralised server. There are a great number of version control systems such as Git, Bazaar and Mercurial. Furthermore, this type of technology enables the hastening and simplification of the software be developed (Otte, 2016). In addition, there are many benefits to be gained using this type of software. If any errors occur during the development process, version control systems provide the capability of rolling back to a stable version of the software to facilitate debugging and error checking. Moreover, this type of tool enables there to be many backups of the software on developer's local machines and in the cloud.

3.7 RESTful API Development

Although the development of a RESTful API is not an essential component in the creation of a chatbot. Implementing this type of service into this software's structural design provides a simplistic method in which to consolidate exchanges amongst independent systems that might be required in a decentralised system. RESTful API's are based on the Representational State Transfer (REST) technology, which is the architectural approach commonly used in web development. Furthermore, this type of technology operates on a messaging based approach meaning that they receive HTTP requests and send back HTTP responses. This is perfect for any web based application. Additionally, incorporating this type of technology into a chatbot's architecture provides modularisation and loose coupling. Moreover, there are many frameworks to be considered in the creation of RESTFUL API's such as Node.JS, Express and Django.

3.7.1 Documentation

Adequately documenting an API can have several distinct advantages. An API, by its very nature, is a service. A service by definition is "the action of helping or doing work for someone" and in most instances, a service is merited by the quality of service it provides and its ease of accessibility. A REST API is no different and it should provide accurate and quality data to the participant that requests it and should also be clearly documented so that programmers, upon reading the documentation, can interact with the REST service with ease. There are many documentation tools that can be utilised whilst developing API's. The most popular in this category of tools are Swagger, API Blueprint and Apiary. They all provide mechanisms whereby the programmers can create websites for their API documentation, test API endpoints from the documentation website and verify response formats. Another feature that these tools provide is a way whereby the documentation code can sync with the application code to provide real-time updates to the API documentation based on live changes made to the codebase.

3.8 Concluding Analysis of Methodologies

After the investigation of the current methods and techniques used to implement a chatbot application, a number of essential technologies have emerged as essential/highly beneficial to the development process.

(1) A messaging platform such as Facebook, Twitter, Whatsapp etc. must be chosen before the project development commences. This is so the development team can adequately investigate what technologies (programming language, data formats etc.) will need to be explored in order for successful compliance and integration with the messaging platform.

(2) An NLP tool such as Api.ai or Wit.ai is a necessary decision to be made prior to development. This factor is imperative to training the chatbot with how to handle certain inputs in a novice-developer-friendly way whereby the chatbot can be trained in a simplistic manner and nullifying the need for the development team to have to implement their own NLP back-end system.

(3) A PaaS platform for deploying the application is an easy-deployment solution for getting the app hosted and running and available for users. There are many providers that are available and choosing the right one for the project centres around the compatible technologies that the platform supports.

(4) Deciding the appropriate server-side programming language and data model for the application are considerations that need to be made in-line with the PaaS provider compatible run-time environments and the messaging platform compatibly requirements.

(5) RESTful API development, for decentralising features of the application can also be considered as a formidable development decision for decoupling features and nullifying a single-point-of-failure within the application.

4 System Requirements and Specification

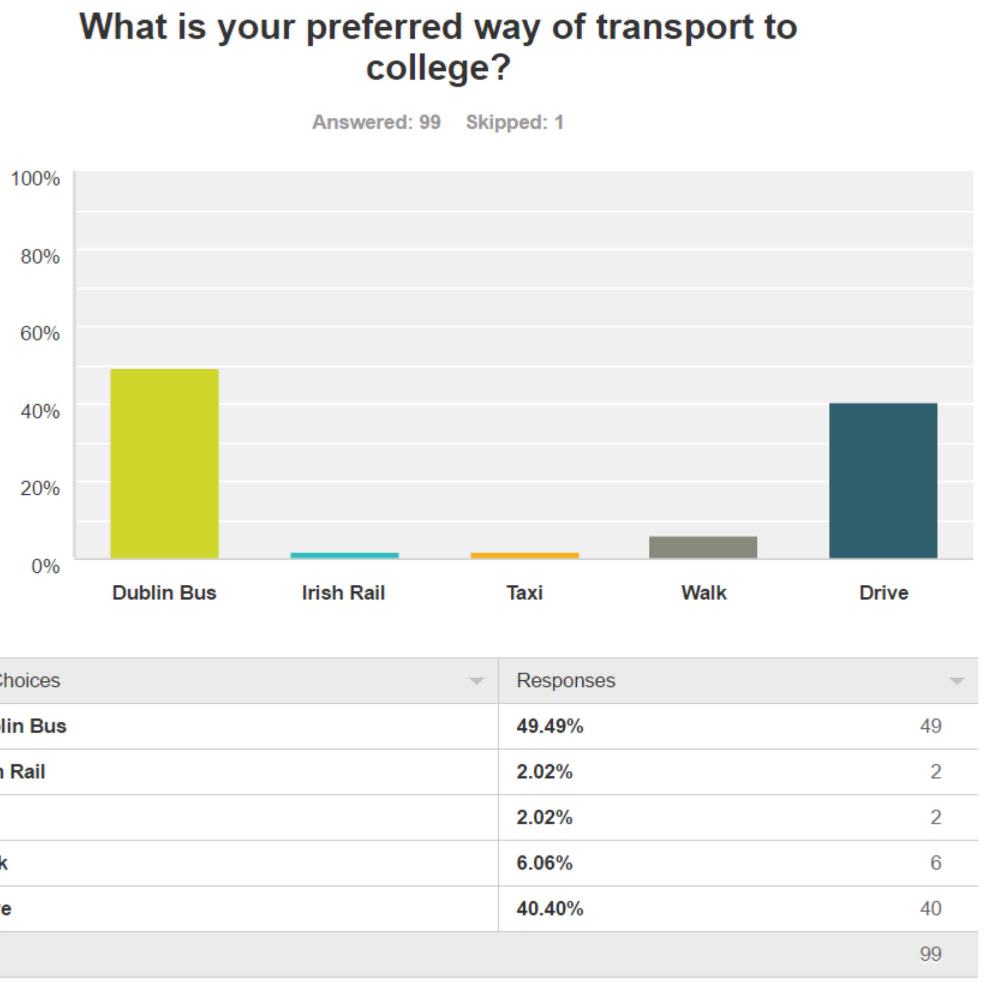
This section will present the requirements and specifications of this project's application. As understood from the methodology section of this paper, decisions needed to be made by the project participants in regard to what feature set the application would have, what the preliminary functionality requirements are, what platform to deploy the application, what messaging platform to integrate with, how to version control the project and also, what natural language processing API to use.

4.1 Preliminary Requirements Specification

A number of surveys have been carried out using the online survey development software, SurveyMonkey, to outline the direction of the project. From inception, this project was envisaged as an application that is specific to ITB and that the chatbot could respond to users looking for information about the college. As there is three participants in the project, it was decided to split the workload in three directions, which each team member conducting a survey in their chosen realm of responsibility. These categories where transportation, educational facilities and fitness facilities. Respectively, three surveys were created on three topics: Transportation survey, Educational Facilities survey and Fitness Facilities survey (See Appendix A.1). The results received were used to gain incite of the functionalities and features the application should possess, and what features may have revealed themselves as redundant and of little consequence.

4.1.1 Transportation Survey

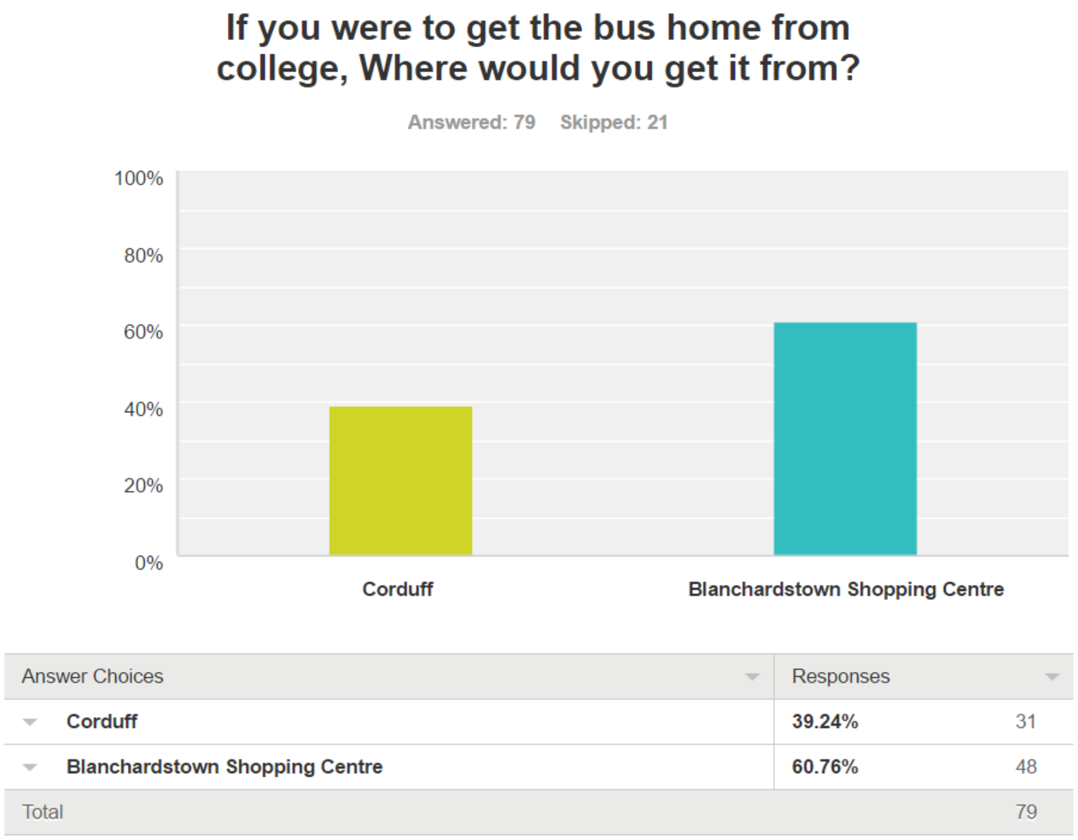
There was a total of 99 participants who have taken part in the transportation survey. In regards to this, it was stated that 49.5% of respondents chose Dublin Bus as their primary means of transport to college, this was followed by driving to college, coming to a sum of 40.4%. This aided in the motivation to implement a transport data service into the application for users to easily obtain information. See figure 4.1.



Answer Choices	Responses	
Dublin Bus	49.49%	49
Irish Rail	2.02%	2
Taxi	2.02%	2
Walk	6.06%	6
Drive	40.40%	40
Total		99

Figure 4.1: Transportation: Question 1.

Secondly, the partakers where asked to specify their typical area for travelling home from college, and to state any alternative areas should they not be on the survey. A majority of the participants responded with 60.76% favour for Blanchardstown Shopping Centre, and succeeded by Corduff, with a high rate of 39.24%. A further understanding of features and attributes to incorporate was provided with these features, therefore it was conceived higher emphasis should be placed on transportation information for the Blanchardstown Shopping Centre. See figure 4.2.

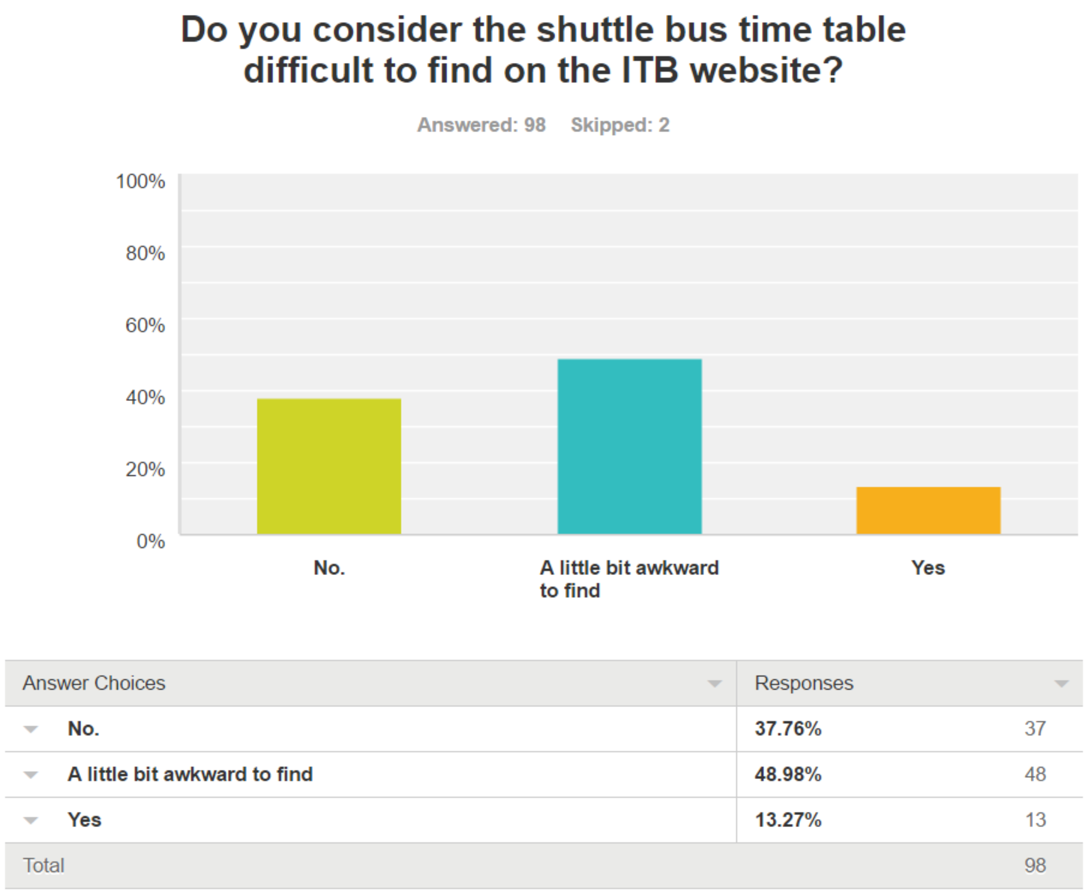

If you were to get the bus home from college, Where would you get it from?

Answered: 79 Skipped: 21

Answer Choices	Responses	
Corduff	**39.24%**	31
Blanchardstown Shopping Centre	**60.76%**	48
Total		79

Figure 4.2: Transportation: Question 2.

Furthermore, it was asked to the average commuter of the shuttle bus provided by ITB whether they felt that the journey times for were difficult to locate on the college website. The results found that 13.27% found it difficult. A proportionally high fraction of commuters at 48.9% found it to be considerably difficult and a 37.76% of applicants found it to not be difficult. Although there is a high portion of participants that do not consider the shuttle bus times complicated to obtain, there is however a sum of 62.25% that may believe there is improvements that could be made. The evaluations of these results lead to considerations on providing journey times for the ITB shuttle bus. See figure 4.3.



Do you consider the shuttle bus time table difficult to find on the ITB website?

Answered: 98 Skipped: 2

Answer Choices	Responses	
No.	**37.76%**	37
A little bit awkward to find	**48.98%**	48
Yes	**13.27%**	13
Total		98

Figure 4.3: Transportation: Question 3.

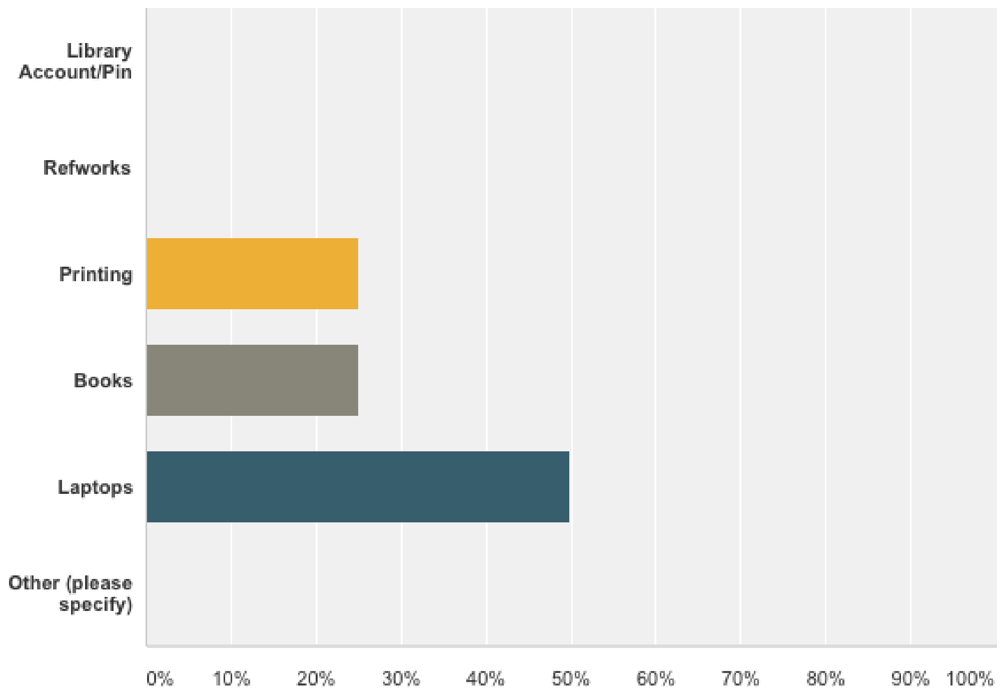
4.1.2 Educational Facilities Survey

For educational facilities, such as the library, four staff members were asked to undergo a survey based on their experience of working in library. Initially, they were asked to highlight the most frequently asked question that they receive from students when working at the student information desk. Laptop loans were the highest selected query, resulting in 50% of the overall given choice. Followed by a tying percentage of 25% between printing information and locating books within the library. See figure 4.4.

Moreover, the staff members where asked to indicate the most recurrent queries in regards to account passwords. Forgotten pin was a the most prominent queries asked according to the participants. Secondly was queries regarding information on how to use their online accounts. See figure 4.5.

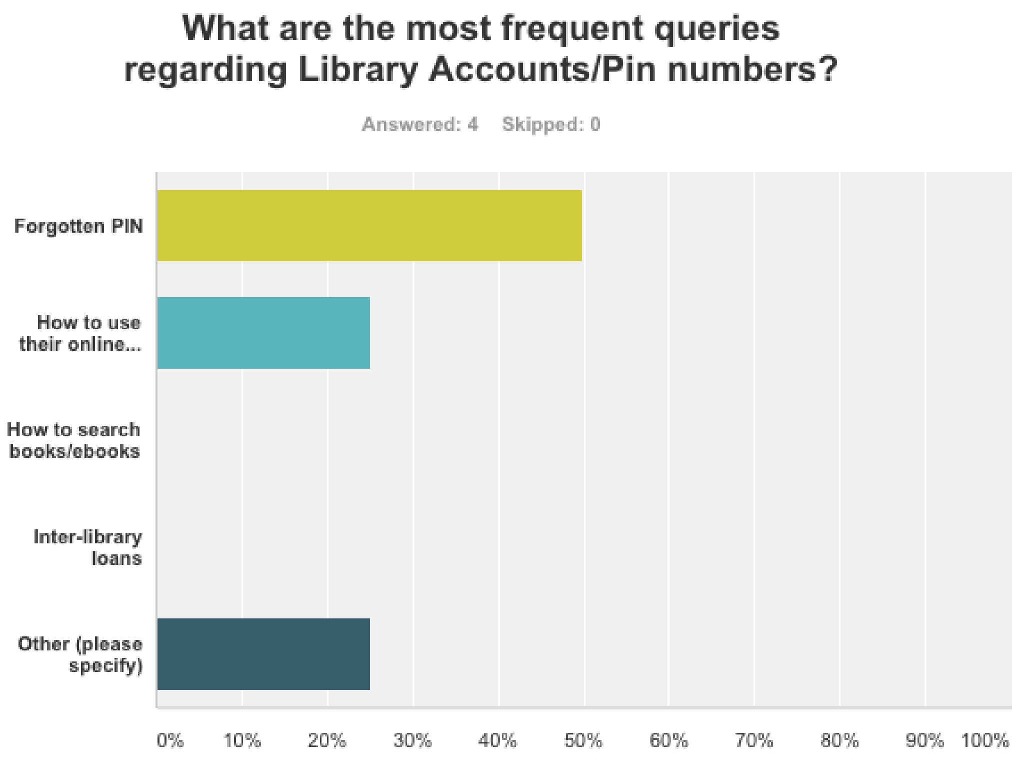
What type of query do you get asked the most frequently at the desk?

Answered: 4 Skipped: 0



Answer Choices		Responses	
Library Account/Pin		0.00%	0
Refworks		0.00%	0
Printing		25.00%	1
Books		25.00%	1
Laptops		50.00%	2
Other (please specify)	Responses	0.00%	0

Figure 4.4: Educational Facilities: Question 1.

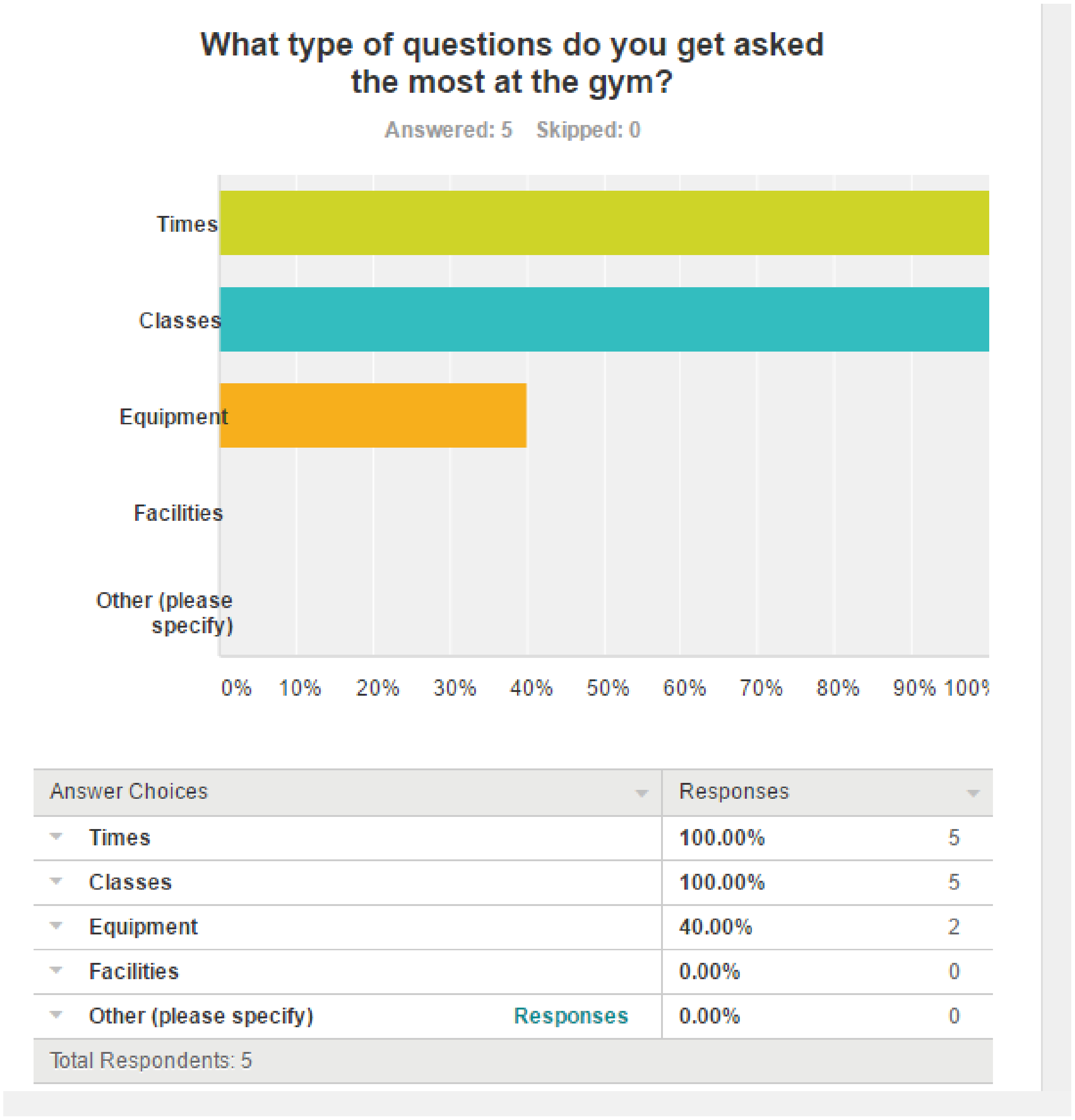


Answer Choices		Responses	
Forgotten PIN		50.00%	2
How to use their online account		25.00%	1
How to search books/ebooks		0.00%	0
Inter-library loans		0.00%	0
Other (please specify)	Responses	25.00%	1
Total			4

Figure 4.5: Educational Facilities: Question 2.

4.1.3 Fitness Facilities Survey

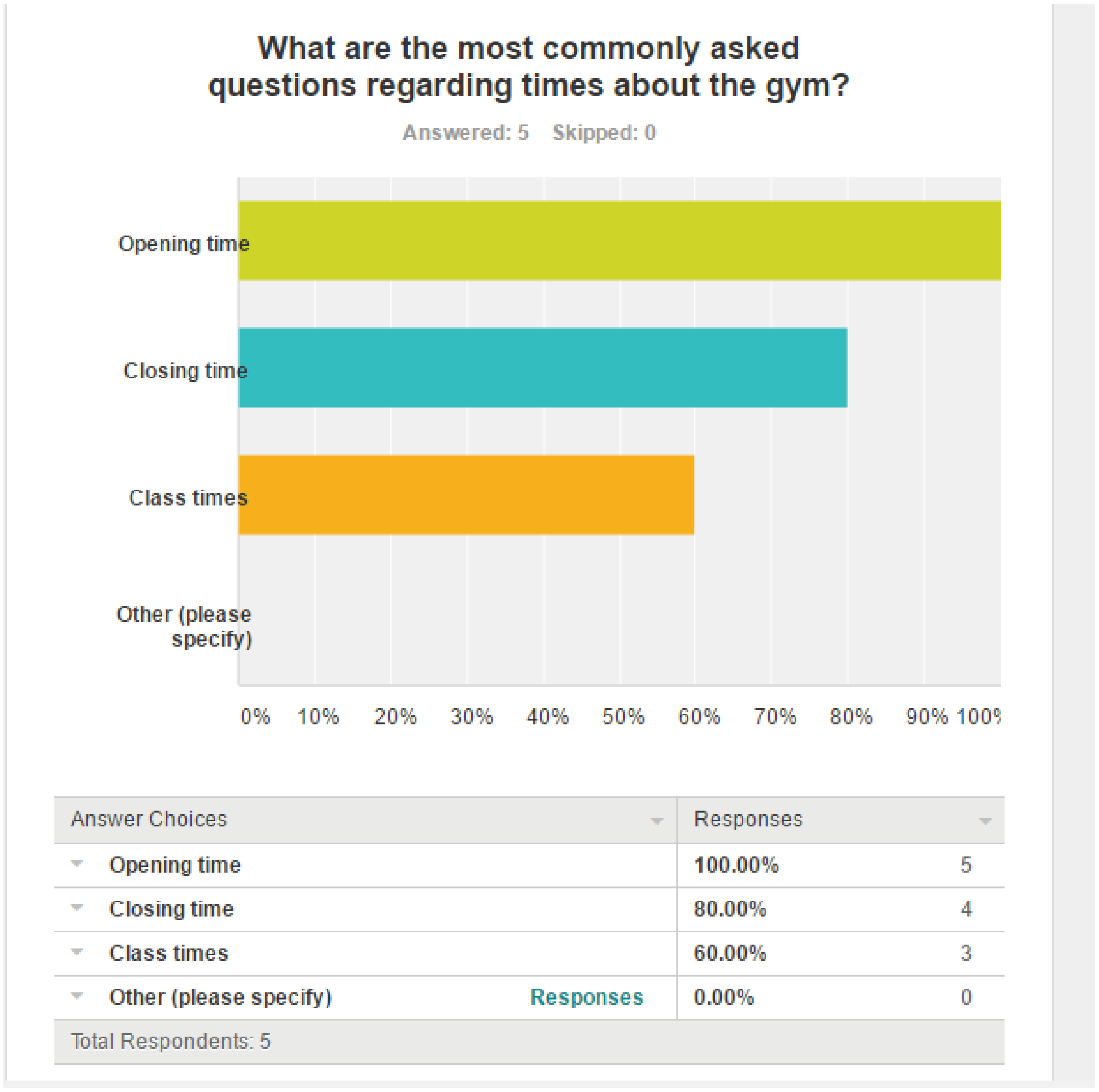
The fitness facilities survey also provided an accurate gauge of the commonly asked questions that the staff members in the ITB gym facilities are consistently queried with. The survey was conducted and five staff members participated. Firstly, staff members were asked to highlight the most frequent questions they are consistently asked. These questions were narrowed down to broad categories. See figure 4.6 to demonstrate how this was done.



Answer Choices		Responses	
Times		100.00%	5
Classes		100.00%	5
Equipment		40.00%	2
Facilities		0.00%	0
Other (please specify)	Responses	0.00%	0
Total Respondents: 5			

Figure 4.6: Fitness Facilities: Question 1.

Furthermore, the staff members were asked to complete a question isolated to a specific area mentioned in the previous question. For example, they were asked to identify commonly asked questions surrounding the area of opening times. See figure 4.7 to demonstrate how the question was delivered.



Answer Choices		Responses	
Opening time		100.00%	5
Closing time		80.00%	4
Class times		60.00%	3
Other (please specify)	Responses	0.00%	0
Total Respondents: 5			

Figure 4.7: Fitness Facilities: Question 2.

The remainder of these survey questions were designed in a similar fashion in order to gain a better understanding of each area in isolation. These survey questions contributed to arriving at a viable and plausible feature set for the application.

4.2 Functional Requirements

The proposed chat bot application's main desirable functionality is to provide students with a number of informational conversations about the college in real time. This involves the implementation of an easy-to-use system that shall engage with the user and provide an intuitive flow of conversation. The chat bot shall also be given with the ability to participate in basic conversational pieces to provide a more "human-like" interaction for the user.

Functional	Requirements
1	Instant responses
2	Conversational capabilities
3	Natural language processing
4	Intuitive conversation flow and ease of use

(1) The chatbot application shall consist of the ability to response to a user almost instantaneously when a conversation is invoked. A user may begin a conversation with the application by using the `Get started` button or by simply messaging the chatbot. In turn, the chatbot shall respond instantly to provide the user with a quick and efficient means of obtaining information.

(2) Conversation capabilities shall be incorporated in order to provide a "human-like" experience for the user. The chatbot shall respond using informal and natural phrases and expressions. An example of this use case would be a user saying "hello" to the chatbot. This should result in a natural reply in terms of language and structure. Additionally, the chatbot shall have multiple responses for the same input. To give a more of an organic interaction, the chatbot shall be able to respond in several ways for the same kind of message it has received. For instance, if a user asks "what's up?", the chatbot should be able to react with several meaningful responses.

(3) The application shall be able to handle the input of natural language by humans. The use of natural language process provides a means to extract meaning full data and contexts from human input and operate with large conversations. Additionally, it shall be able to understand the input from a human given the event of a human incorrectly spelling a word using the method of lexical analysis and evaluating the conversational discourse. A use case of this would be if a user typed "I am on the **bua**" instead of saying the word "bus", the chatbot shall still be able to understand the intent of the conversation.

(4) Lastly, the chatbot establishes an easy and intuitive conversational flow to guide the user through the dialogue to provide the most meaningful interaction as possible. This shall be achieved by providing the user with as much information as possible in order to prevent the user from being confused or disinterested with the interaction. This could be done through the use of buttons or menus.

4.3 User Interface Specification

The system shall be comprised with an intuitive and clean design. Chatbot applications should give a means for the user to instantly interact without an unsolicited need to type every response they wish to give. To accommodate for this, the use of buttons should be implemented to allow for a user to interact quickly and efficiently. In regard to graphical user interface design, no GUI implementation is necessary as the interface is provided by Facebook's Messenger application. However, Facebook does provide a number of templates, menus and buttons that can be utilised to design a smooth and straightforward interaction with the application. These content types use raw JSON which is sent to the Messenger Send API to generate graphical components for a messenger chat conversation. The application makes use of a number of these content types in order to display the understanding and comprehension of using the assets that Facebook provides.

4.4 Platforms

The following section shall touch upon the appointed platforms used in the process of development, and go in to detail about some of the terminologies and platform specific phraseology concerning these platforms. Additionally, it will describe the benefits, limitations and necessities of the chosen platforms

4.4.1 Heroku

Heroku was chosen as the adequate Platform as a Service (PaaS) tool for this project. Heroku differs from it's competitors because developers on Heroku don't purchase managing software or server instances. Instead, the code that a developer wishes to publish is pushed to a GitHub repository and deployed instantaneously. This proves beneficial in the fact that a version control system is in place, which provides a means for applications to be rolled back to previous versions should they need to. There is also five major reasons why Heroku is a preferable service to use for application development (Upreti, 2013). Firstly, Heroku required minimal prior knowledge of platform terminologies, in regards to git commands, in turn making it very easy for developers to acquaint themselves with the environment. Another prominent reason for the utilisation of Heroku is integrity. Heroku is built on Amazons Web Services, which is known for their high uptime rate (Upreti, 2013). Heroku is also provided with vast amounts of documentation on how to use the platform, and also consists of a large community of developers. The fourth reason is the variety of application types given to the developers by Heroku. Heroku supports a wide range of programming languages to be deployed on to it's service, including JavaScript, Ruby, PHP and python. Lastly, economy is a outstanding incentive for using Heroku as it allows a free plan for developers to deploy applications, but with limitations. There are a number of terminologies to be acquainted with when working with Heroku. A singular instance of an application running on Heroku is referred as a Dyno. Each Dyno, on a free plan, consists of a quad core central processing unit, 512 Megabytes of random access memory (RAM) and 550 free Dyno hours per month (Heroku, 2017). To prevent redundant consumption of hours, the free plan provides Dyno sleeping, where the application goes into a sleeping state after thirty minutes without receiving any HTTP requests. This plan proves beneficial to the proposed project as it allows for cheap deployment, and is also suitable for scale of the application being developed. Heroku also provides an add-on service, whereby fully maintained infrastructures for third-party applications are provided for developers. These third-party add-ons build upon the existent Heroku platform to provide a range of services for applications (Heroku, 2017). This enables developers to utilise existing database storage capabilities best suited for their applications without the need of general maintenance. Some examples include Redis and SQL storage such as Compose MongoDB, Heroku Redis, JawsDB MySql and Heroku Postgres.The Heroku toolbelt a package that consists of a of the Heroku CLI, Git and Foreman. These tools are used by the Heroku environment in order to deploy applications to the cloud. The Foreman package in a command line utility used for running applications that use Procfiles (Heroku, 2017) and allows developers to test applications locally on their machine. The toolbelt also include the necessary Heroku commands for remote version control deployment. Unlike regular git commands that deploy code to a remote repository on GitHub, the toolbelt possesses it's very own command's to push code to a Heroku repository. See command below:

```
git add .
git commit -m "message"
git push heroku master
```

4.4.2 Facebook Messenger

One of the worlds leading social media sites, Facebook, has opened its platform for developers to build upon its existing infrastructure. The Facebook Messenger app has quickly grown to be the most popular mobile app in the world. Facebook's Newsroom website states that there is "1.23 billion daily active users on average for December 2016". The use of this platform gives a great advantage to developers as it provides an pre-existing congregation of users. Developing applications for multiple platform types such as IPhone, Android or Blackberry becomes redundant as the Facebook Messenger application already has a large community. A number of tools have been release by Facebook to allow developers to utilise its resources, send data and harvest profiles information to provide better user experience, especially in the case of developing chatbots. The Facebook Graph API is an application program interface that allows for a applications to connect to the Facebook Messenger Service. This is done through the use of a webhook. A webhook enables applications running on a server to subscribe and listen for an event to happen. In this case, an event would be a HTTP Post request is invoked to the callback URL (Webhook) when user interacting with the application triggers an event. For example: send a message or a picture. Real-time updates with the use of a webhook can allow an application to cache data or updates immediately. This approach removes the need for polling the Facebook servers continuously which can reduce performance, in turn, making the application more reliable and decrease load times. Other additional features the Graph API includes are the sending and receiving JSON using a HTTP request using the POST methods. This sends a response to Facebook servers using the a page access token generated by Facebook.

```
curl -X POST -H "Content-Type: application/json" -d '{
    "recipient": {
        "id": "USER_ID"
    },
    "message": {
        "text": "hello, world!"
    }
}' "https://graph.facebook.com/v2.6/me/messages?access_token=PAGE_ACCESS_TOKEN"
```

The JSON snippet as shown above demonstrates a post to the graph API sending parameters of the user who should receive the message, and the message that will be sent to the user. Facebook handles this JSON response accordingly to update the UI the user is viewing, be it a phone application or through computer.

4.4.3 Git and GitHub

Git is a version control system (VCS) used for collaborative development of code. A copy of the source code and other essential assets are stored in what is known as a repository. This repository, also known as a repo, is tracked and maintained securely should the team or developer roll back on to a previous version of an application. Git is also accompanied with an online graphical interface called GitHub that allows team members to review, analyse and track repositories. GitHub provides a number of additional services such as graph representations of commits to a repository, online documentation using the GitHub pages service, visual portrayal of code differences between separate commits and issue tracking, where users can collaborate, provide insight and help solve problems that may arise during code development.

4.4.4 GitHub integration on Heroku

Although Heroku provides remote repositories for source code be stored, the option for GitHub integration is also permitted. This allows for developers to use a GitHub repository on their own account, or in a GitHub organization, to deploy applications to GitHub without the need of the Heroku Toolbelt (Heroku, 2017). Should the developer or team decide to enable automatic deployment on their application, a user can commit changed to the GitHub remote repository and it is then deployed to Heroku. In result, this is beneficial in this case as it allows for multiple team members to review the changes made to the application online and, in turn, speeding up the development process.

4.5 Tools

A wide range of tools and technologies have been considered and evaluated during the planning process of the proposed project. Each having similarities and differences, strengths and disadvantages and fundamental necessities for a project of this scale. The following sections will describe in detail the affirmed tools used, their features and why they have been chosen ahead of their competitors.

4.5.1 Node.js

Node.js is a server side platform that is typically utilised for server side applications. Node.js expands upon the JavaScript language to give additional functionalities and advantages for applications. There are a few core aspects why Node.js is a desirable platform to use for development of the project. As it is built upon the Google V8 JavaScript engine which consists of the V8 interpreter and compiler, it is very efficient and is capable of fast execution times. Secondly, it is event driven and asynchronous. This means that all the application program interfaces of node are non-blocking, and loops continuously, waiting for an event to occur. In the case of an event, a callback function is invoked and code is executed. It also only uses one execution thread to serve as the backbone. This makes the execution of a Node.js application lightweight and efficient. In addition, Node.js also accompanied by the Node Package Manager (npm). The npm is a command line client which enables programmers to pre-existing code packages known as Node Modules. This approach aids in speeding up development as it eliminates the need for programmers writing code for a certain functionality when the dependency can be implemented easily and efficiently. The modules are stored on remote repositories that are searchable using the NPM website. Although chatbot development is achievable in a range of programming language, Node.js JavaScript proves to be the most suitable as it proficient in server-client communication. As well as being reliable, it's speed is a preeminent advantage for real-time application such as a chatbot.

4.5.2 MongoDB

MongoDB is a non-relational database that uses document-oriented storage. MongoDB Documents objects are files that stores JSON-like binary files that are often referred to as “BSON”. MongoDB Collections are similar to tables in a relational database management system. Collections consist of groups of Documents to store data. There are a few reasons why using a non-relational (NoSQL) database proves to be advantageous for the project at hand. Relational databases can only store structured data within tables. Should an application store various pieces of information that are not specifically the same for every instance, NoSQL is ideal for this. The option for scalability should be something to possess if new or unexpected pieces of information need to be added to the database (Wodehouse, 2017). Additionally, as the use of cloud storage is essential in the case of the application being hosted on Heroku, Cloud-based storage is better served when fragmenting and sharing information across multiple servers. MongoDB caters to that need as it has built in sharding capabilities, which are used for partitioning and distributing data across a number of machines accordingly. (MongoDB, 2017).

4.5.3 Api.ai

Api.ai is Google's natural language processing (NLP) platform. API.ai provides an interface for highly sophisticated conversations with a chatbot. This is done through the means of natural language understanding (NLU) that involves taking an unstructured input from a human, and reorganise it into a structured form that is understandable to a machine, and in turn, the machine triggers an event based on that input (Bryan, 2016). When an input is received, API.ai converts the input to what is called "actionable data" and returns a JSON object output. API.ai also consists of a dialog management system that uses computational linguistics to study the context and intent of the input, and allows the developer to regulate how the conversational flow should be directed. API.ai also uses machine learning (ML) to train the bot to understand intents of inputs and determine a correct and suitable output. An Agent (the API.ai terminology for the chatbot) uses ML to learn from data provided by the user and builds a model for evaluating and determining correct responses accordingly. This model is unique for every agent (API.ai, 2017). To ensure that responses are accurate, API.ai provides a training tool that allow developers to analyse conversation logs with the agent being tested, and assign intents for sentences with unexpected wording or structure. Furthermore, there are a few terminologies that are associated with the API.ai platform. Entities are objects that are used to extract parameterised value data from natural inputs. Entities come in three forms. There are System entities, which are defined by the API.ai platform. This entity if used for popular concepts and consolidates common attributes like colour, date, numbers, location etc. Secondly, are the developer defined entities. These are agent entities that are produced by the developer. The developer-mapping entity type can allow the developer to specify synonyms. Natural language provides multiple ways to say something with the same meaning, for example: a Car entity could be synonymous to vehicle, or BMW or even automobile. The developer-enum type are entities that do not have a mapping of reference value (API.ai, 2017). Developer composite entities are entities that contain an aggregation of other entities with the same aliases. For example: a composite entity for a car's `Drive` function could contain multiple other enum entities, Like `Accelerate` and `indicate`. Lastly, there are the User entities. These entities are specified by the user. an example of this would be a user's playlist, because usually playlists of songs are specific to and are made by each user (API.ai, 2017). The second eminent term involved in the API.ai platform are actions. An action is the route that application takes when an intent specified by the user is given as input. Actions can be provided with parameterised values that can be extracted. Examples given by developers in intents containing the specific words can be given an action and a value. This is a valuable functionality when specifying something like quantity. An example of this would be a user asking an agent to order "one" item instead of ordering "five". The value is extracted from the user input and synchronized with the conversation.

4.6 Concluding Analysis of System Requirements and Specification

This section will give a summarisation of the technology stack chosen to implement this project. In lieu of the current methods being adopted to build chatbot applications several tooling considerations had to be made and as a result the appropriate tools for this project where chosen.

(1) Heroku was chosen as the PaaS provider because of its ease-of-integration with GitHub and its flexible free tier. It provides 550 hours of up-time. This tier also allows for the app to 'sleep' when not being pinged with HTTP requests, therefore, not eroding the up-time quota of hours when the app is not is in an in-active state.

(2) Facebook Messenger was chosen because the availability of resources and documentation regarding integrating with their platform was extensive and well-documented. Facebook messenger is also language agnostic and this provided a freedom for the development team to use whatever server side language they were most comfortable/experience with.

(3) Api.ai was chosen as the NLP engine to use for parsing input that the server receives and returning the intent

(4) Node JS was chosen as the server-side programming language for several reasons. Perhaps the most notable factor is that the majority of literature and tutorials on designing and developing chatbots are all implemented with Node JS. Nodes single threaded execution model also lends itself to messaging applications because of its rigid enforcement of the ordering of requests. Node, built on top of the Google Chrome V8 JavaScript engine and implemented in C and C++, provides fast execution times which is essential for a chat-based server.

(5) MongoDB was chosen as the primary data model for the application. This is because of its ability to shard easily and evolve in-line with the data model changing. It is difficult to preempt exactly the data format during development because naturally the database will have to expand whilst the application is being used by end-users. There is also a lot of tutorials on developing chatbots with the MongoDB database client and this was also a contributing factor to its selection.

(6) Api.ai was chosen as the NLP engine for this application because its accessibility to developers and its powerful integration capabilities with the big messaging platforms.

5 System Design

The design of the system incorporates multiple technologies and platforms to meet the requirements of the application. As specified, the application must possess a means to commission data across a number of components. Many factors have been taken into consideration during the planning process of this project and the following section describes in detail how the technologies utilised interact with one another.

5.1 Decentralised applications

The approach of decentralised computing was applied using application program interfaces to provide information. These peripheral applications are autonomous to the central chatbot application and are hosted independently from the main application instance. Each API developed provides a specific type of information and runs on their own individual server hosted on Heroku. This proves highly beneficial in a case that if one application fails, it does not disrupt the rest of the system. Three applications have been implemented to provide information in correlation with the main chatbot application. Each of these administer data about gym information, library information and real time bus information. Firstly, the bus API involves extracting parameterised values sent from the primary application. These values are then used to send a request to the Public transport predictions and schedules RTPI REST Web Services API and, in turn, composes the response data to a human-like response. The bus API then sends the data back in a JSON format to the central application. Suitably, no database is integrated within this API as it uses dynamic data that is retrieved by the RTPI RESTful API. In contrast, the gym and library API's use routing to specify the type of data needed. Each route is used to extract data from a NoSQL MonogoDB database hosted on mLab. These are static JSON documents that store information specific to the route specified in the API request from the main app. See figure 5.1 for a general topology of the API architecture.

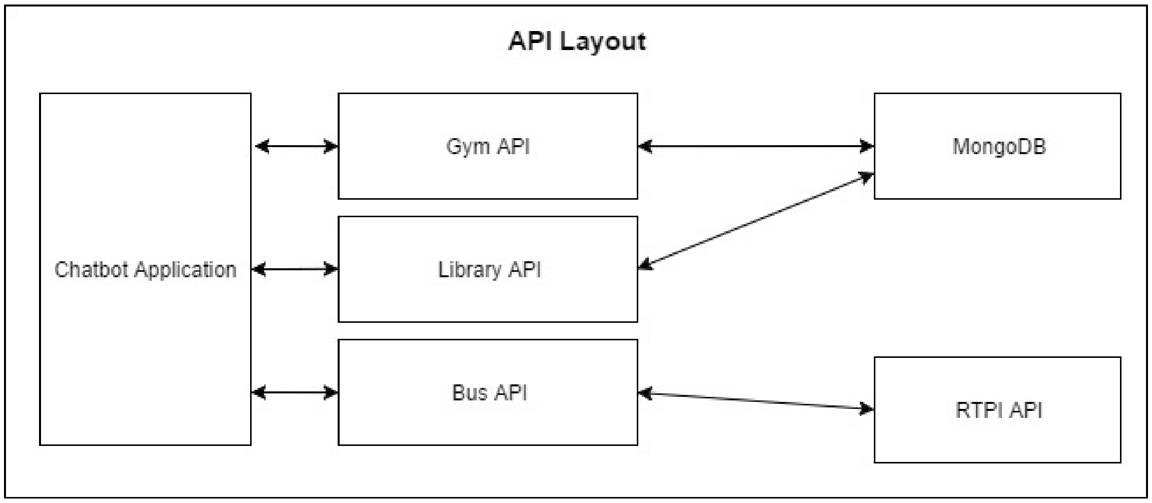


Figure 5.1: API Topology.

5.2 GitHub and Heroku Integration

In order keep track of code being written during the process of implementation, GitHub repositories and Heroku applications were created for each of the API's and the central application. These repositories were configured to integrate with the Heroku applications. Once a commit has been pushed from a team member's local machine to the remote repository respectively, the code is automatically deployed to the Heroku server. A build is then commissioned by Heroku, and should the build succeed, the application is released and hosted on the server. For the following integrations, the Aaron_api (bus) repository was assimilated with the aaronapi application, the Brian_api (gym) repository with the brianapi application and lastly, the Daire_api (library) repository with the daireapi application. The main central application is integrated using the Chatbot_Assignment repository in correspondence with the itbchatbot application.

5.3 Application Architecture

This section will present the architecture of the system in a logical view that will clearly outline each node/participant in the application. The application relies on each participant to fulfil a certain use case that contributes to the overall successful processing of an input. The main participants are:

1. The developers
2. The Version Control System/GitHub
3. The instant messaging platform/Facebook
4. The PaaS Provider/Heroku
5. The NLP engine/Api.ai
6. The database server/MLab

Whilst the version control system may be deemed an inconsequential part of the overall system architecture, because of Heroku's dependency to deploy directly from GitHub repositories, it is seen as an important participant, thus, it can be deemed as an appropriate inclusion to the proceeding system architecture diagram. Also, the inclusion of the developers in the system architecture is because of a consistent need of monitoring the applications performance via logging and configuration updates that may need to be run on the applications dashboard interface. See figure 5.2 for a logical view of the system architecture.

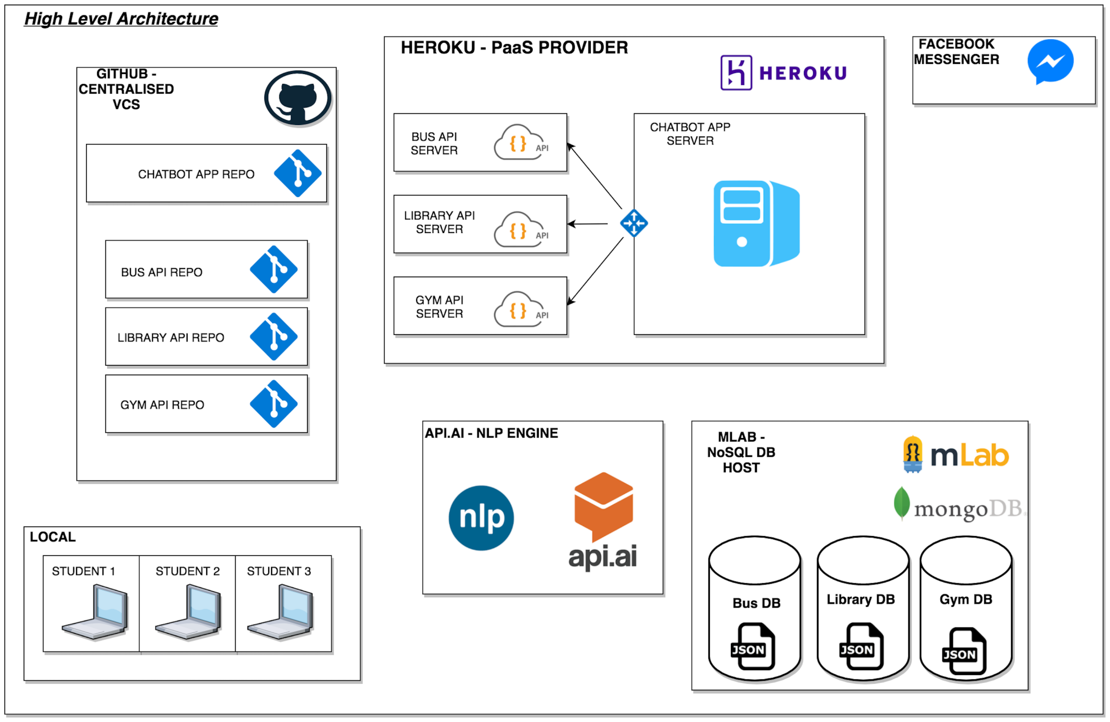


Figure 5.2: High Level Architecture.

5.4 Build Execution

This section will demonstrate how the code progresses from the developer's machine to running live on the Heroku app servers. This process will highlight the strategy that the participating team members took in order to implement a structured routine for development and also to see an overview of how the code migrates from machine to app-server. There are five steps in order to fulfil this journey-to-deployment and they are:

1. **Develop code locally**

For feature implementation, the code must be developed locally first on the team members laptop completely decoupled from the 'live' code that is being served. This is to avoid interfering with the current working build and to nullify uploading any 'breaking' code prematurely. The development environment on the team members machine should be set up in such a way where the application can be hosted from the machine and tested.

```
api.get('/newroute',function(req,res) {
            message= "The new response"
            res.send(message);
    });
```

2. **Test locally**

Local testing takes place whilst a new feature is being implemented. An example would be testing a HTTP GET request on a newly-implemented endpoint on the REST API that is being developed. This ensures that the business logic code is behaving as it should be, is returning the correct response and working as intended with the rest of the routes/endpoints.

```
http://localhost:3333/api/newroute
"The new response"
Success! Response code 200!
```

3. **Push to GitHub repository**

Once the new iteration has been tested locally and all code quality assurances have been made the newly updated software is ready to be pushed to the host repository on GitHub. This will trigger a deployment automatically to the Heroku app server.

```
git add fileThatWasChanged
git commit -m "added 'newroute' to the api, tested it, and it works"
git push origin <branch-im-working-on>
```

4. **Ensure Heroku has auto-deployed**

Check the Heroku dashboard for the application, ensure it has deployed without-error and running with no issues. 'Build succeeded' and 'Deployed' messages will appear on the dashboard as a result of successful deployment.

5. **Test deployed implementation**

Finally, to make sure that the newly implemented feature works as intended now that it is deployed and live, testing the endpoint is carried out in order to verify the new features function.

```
https://myapp.herokuapp.com/api/newroute
```

```
"The new response"
Success! Response code 200!
```

A logical representation of how the code builds from development to app server is shown in figure 5.3. This diagram highlights the behaviours that take place once a new feature has been pushed to GitHub with reference to the project application. Each team member has the sole responsibility for their own API and each team member has the responsibility to develop the main application also.

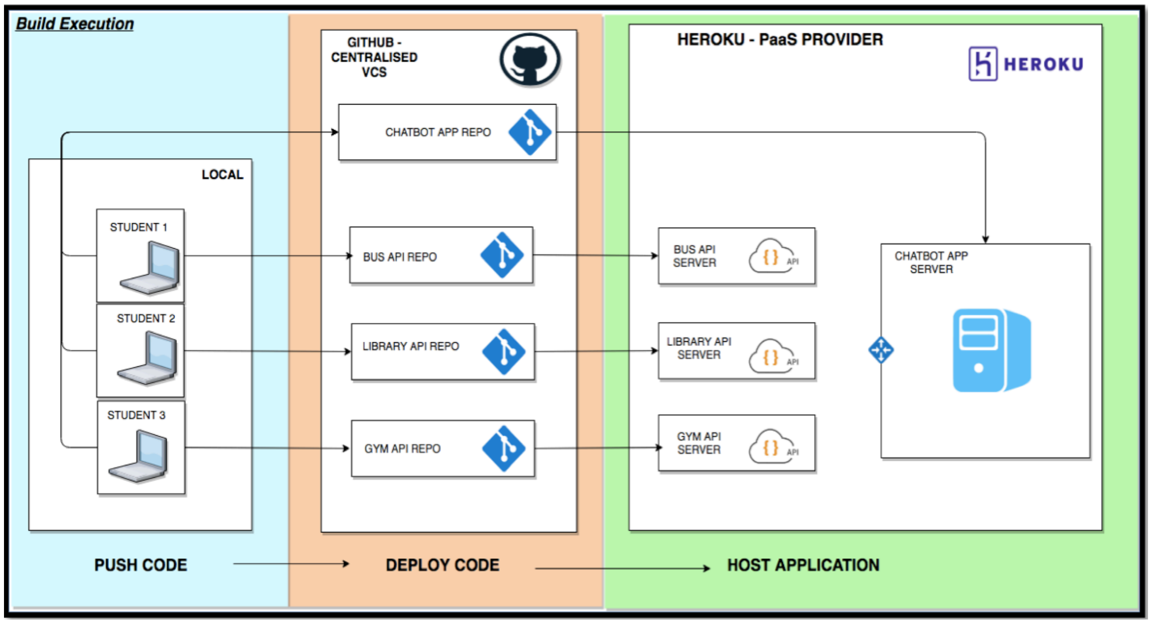


Figure 5.3: Build Execution.

5.5 Conversational Flow Use Cases

This section aims to give an adequate understanding of the conversational design within the application. Depending on the context of the conversation that the user has engaged with, there are three pathways that the conversation can flow. These three different pathways are representative of the individual REST API's that each team member developed and outline the primary conversation flow that a user can be brought down. See figure 5.4 for a logical representation of this.

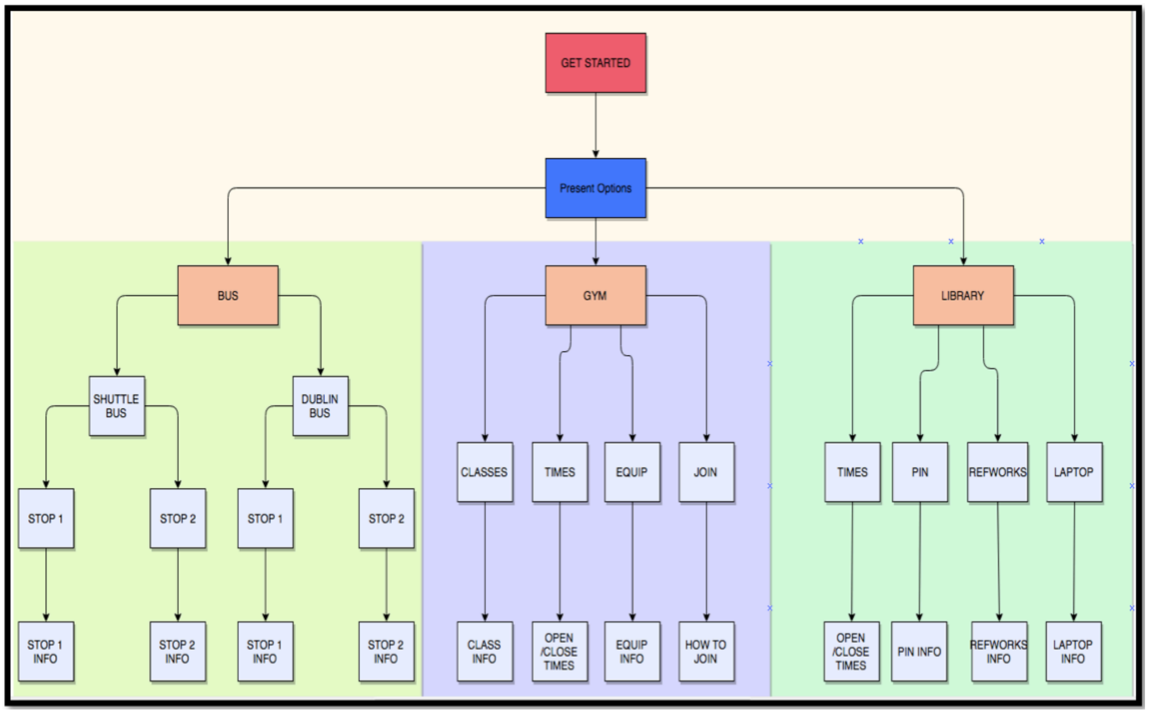


Figure 5.4: Conversational Flow.

5.6 Typical Use Case Execution

This section aims to provide a logical overview and a sequential explanation as to what happens from the moment the users sends a message to the chatbot. This step-by-step interaction between all the separate participants in the application happens every instance that the application receives a message from the Facebook Messenger webhook. See figures 5.5 and 5.6 that shows every step from the user sending a text to the chatbot, right through to receiving a response back.

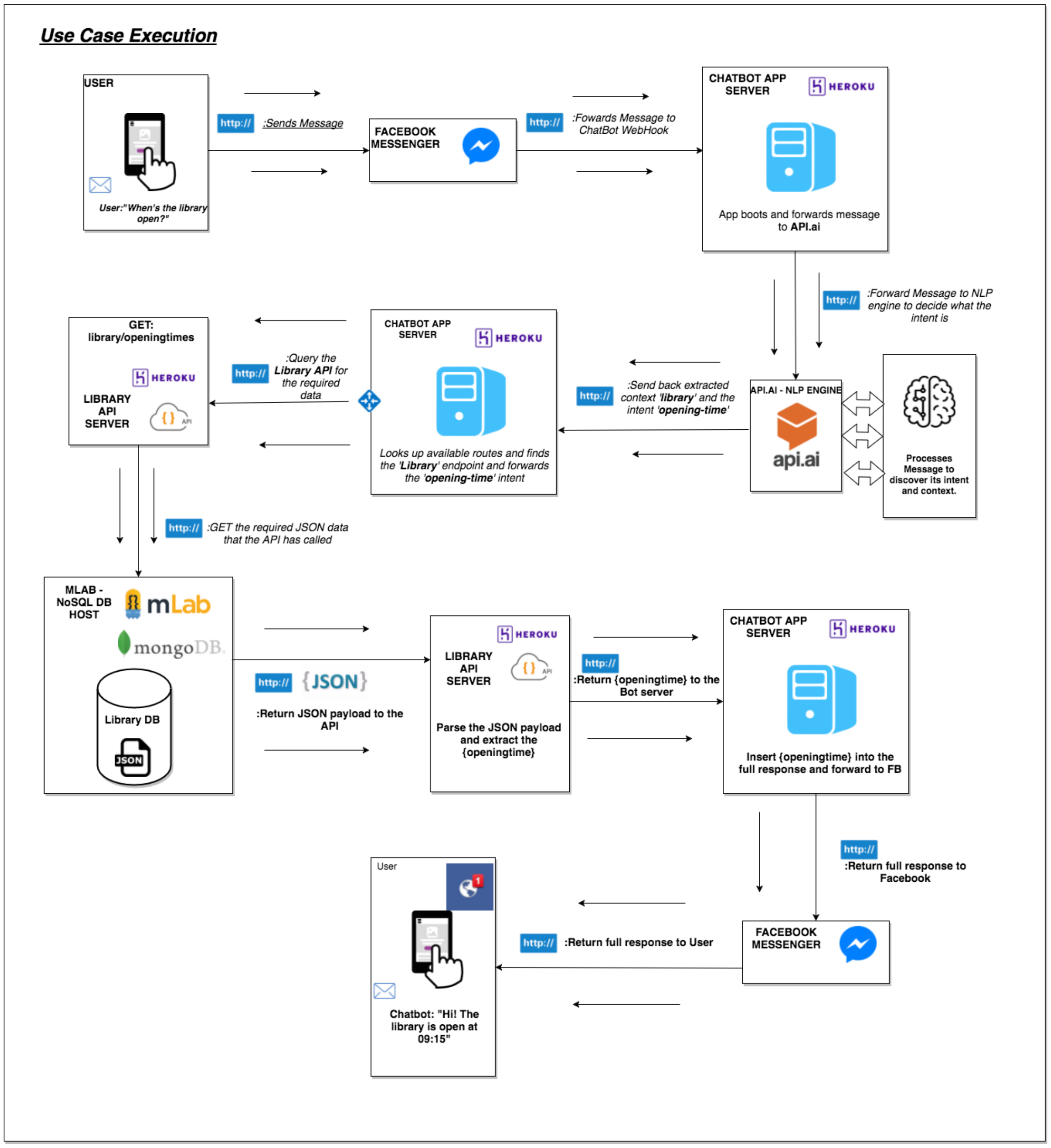

Use Case Execution
USER
User:"When's the library open?"
http:// :Sends Message
FACEBOOK MESSENGER
http:// :Fowards Message to ChatBot WebHook
CHATBOT APP SERVER
HEROKU
App boots and forwards message to API.ai
http:// :Forward Message to NLP engine to decide what the intent is
API.AI - NLP ENGINE
api.ai
Processes Message to discover its intent and context.
http:// :Send back extracted context 'library' and the intent 'opening-time'
CHATBOT APP SERVER
Looks up available routes and finds the 'Library' endpoint and forwards the 'opening-time' intent
http:// :Query the Library API for the required data
GET: library/openingtimes
LIBRARY API SERVER
http:// :GET the required JSON data that the API has called
MLAB - NoSQL DB HOST
mLab
mongoDB.
Library DB
JSON
http:// {JSON}
:Return JSON payload to the API
LIBRARY API SERVER
Parse the JSON payload and extract the {openingtime}
http:// :Return {openingtime} to the Bot server
CHATBOT APP SERVER
Insert {openingtime} into the full response and forward to FB
http:// :Return full response to Facebook
FACEBOOK MESSENGER
http:// :Return full response to User
User
Chatbot: "Hi! The library is open at 09:15"

Figure 5.5: Use Case Execution.

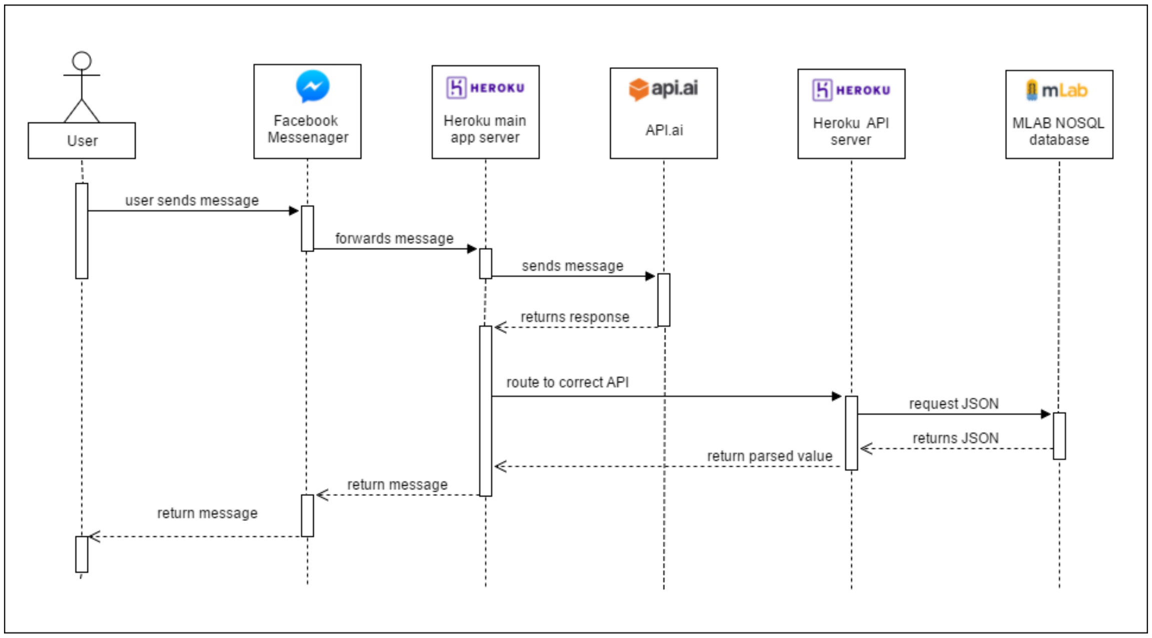


Figure 5.6: Use Case Sequence Diagram.

5.7 Concluding Analysis of the System Design

The way in which the system was designed was a thorough analysis of how each participant in the system would interact with each other. Upon investigating the technologies for this application, a complete understanding of how these nodes would participate with each other and exchange instructions was a vital understanding prior to development. As an exercise in individual work amongst team members, the decision was made to implement each feature as its own RESTful service and have these services deployed to Heroku. The team chose to segregate the conversational flows into the three feature domains and then have a ‘master’ application which called upon these services when required. This led to a decoupling of the applications feature and nullified any single points-of-failure.

6
Implementation

The following section will illustrate the implementation of the applications utilised within the project based on the system requirements and specifications and the system design that has been aforementioned. Firstly, a detailed description of the primary chatbot server application will be provided followed by the implementation of the individual API's. It shall also be described the process of developing a comprehensive natural language processing agent with Api.ai. Additionally, it will be explained how the separate applications communicate with one another in accordance to functionality and purpose. Code snippets shall also be given to demonstrate the algorithmic logic implemented into the application.

6.1 Chatbot Application

The structure of the main application entails a configuration file to set the ground work for the entire application. This file contains the configuration variables that are utilised across the main index file to gain access to the additional tools and services mentioned in the specification. Page access tokens provided by the Facebook messenger platform, developer defined verify token and application secret are used to authenticate access to the Messenger Send API. In addition to this, a verify token for API.ai is also stored to authenticate communication between the chatbot application and bot engine. Lastly, a `SERVER_URL` is provided to give a short-hand to accessing the main application host address.

```
module.exports = {
    FB_PAGE_TOKEN: '<TOKEN>.',
    FB_VERIFY_TOKEN: 'secret',
    FB_APP_SECRET: '<SECRET>',
    API_AI_CLIENT_ACCESS_TOKEN: '<TOKEN>',
    SERVER_URL: "https://itbchatbot.herokuapp.com/",
};
```

Ensuing the preliminary basis to the application follows the Node.JS modules that have been utilised within the project. These dependency injections aid in the implementation of the functionalities and provide additional services to the application. Initially, the `apiai` module is installed to allow the application to communicate with the API.ai natural language processing service. The `body-parser` module is used as middleware to parse then response bodies. `Crypto` is used to verify the secret located in the request header that has been sent from the Facebook. The `express` module is a framework for Node.js that provides JavaScript to be executed without the aid of a web browser. In order to make HTTP calls, the `request` module is used and lastly, the `uuid` module is utilised to generate a session ID.

The main server JavaScript file named `index.js` is the backbone of the chatbot application. This file contains all the logic for sending and receiving messages, as well as dealing with responses and JSON templates used to drive conversations. Initially, the basic configurations for the server are set, this involves providing a port for the application to run on. This is allocated dynamically by Heroku. Also, a connection to the API.ai engine was then established.

```
//Set the port of the app
app.set('port', (process.env.PORT || 5000))

const apiAiService = apiai(config.API_AI_CLIENT_ACCESS_TOKEN, {
    language: "en", requestSource: "fb"
});
```

A webhook route is set to receive notifications from the Facebook Messenger platform. This connection was achieved by inserting the webhook URL to the developer settings on Facebook. The following function authenticates the application and verifies access. In addition to authentication, condition handling was implemented in order to appropriately deal with messages being sent by the user. This higher-level message events delineate the type of messages being sent. For example, the message could be a regular text message, delivery confirmation, read receipt or postback etc.

```
app.get('/webhook/', function (req, res) {
    if (req.query['hub.mode'] === 'subscribe' &&
        req.query['hub.verify_token'] === config.FB_VERIFY_TOKEN) {
        res.status(200).send(req.query['hub.challenge']);
    } else {
        console.error("Verification was not valid.");
        res.sendStatus(403);
    }
    });

data.entry.forEach(function (entry) {

            // Iterate over each messaging event and handle accordingly
            entry.messaging.forEach(function (messagingEvent) {
                if(messagingEvent.message) {
                    receivedMessage(messagingEvent);
                }
                else if (messagingEvent.delivery) {
                    receivedDeliveryConfirmation(messagingEvent);
                }
                else if (messagingEvent.postback) {
                    receivedPostback(messagingEvent);
                }
                else {
                    console.log("Unknown event type ..")
                }
            });
        });
```

To provide a easy means of understanding the functions in the source code, a three type naming convention was used. This approach was used to aid in comprehending how the code worked and what it dealt with. These three types of functions where used to handle communication between three end points. Functions beginning with the word "receive" are used to carry out operations when a request has been sent to the server webhook URL from Facebook. The functions that use the beginning word "handle" are functions used to manage payloads such as templates. Lastly, functions using the name "send" are used to administer data to other end points via the chatbot application. The following sections describes how these functions were implemented in addition with the most importantly used functions as examples.

6.1.1 Receive functions

The primary function used in a regular use case is the `receivedMessage` function. This takes in the message event as a parameter and then uses the data to initialise variable values to be used in the code. The most prominently used of these variables is the `senderID` with is needed to reply to the user who has interacted with the chatbot. This ID is unique for each Facebook user and is generously passed to other functions within the code.

```
function receivedMessage(event) {

    //Set variables from json
    var senderID = event.sender.id;
    var recipientID = event.recipient.id;
    var timeOfMessage = event.timestamp;
    var message = event.message;

    if (!sessionIds.has(senderID)) {
        sessionIds.set(senderID, uuid.v1());
    }

    //Set variables
    var isEcho = message.is_echo;
    var messageId = message.mid;
    var appId = message.app_id;
    var metadata = message.metadata;

    // You may get a text or attachment but not both
    var messageText = message.text;
    var messageAttachments = message.attachments;
    var quickReply = message.quick_reply;

    //check type of message
    if (isEcho) {
        handleEcho(messageId, appId, metadata);
        return;
    } else if (quickReply) {
        handleQuickReply(senderID, quickReply, messageId);
        return;
    }
    //Check if it's a text message
    if (messageText) {
        //send message to api.ai
        sendToApiAi(senderID, messageText);
    } else if (messageAttachments) {
        handleMessageAttachments(messageAttachments, senderID);
    }
}
```

The `recievedPostback` block is a significantly important function used to handle postbacks sent back from the user when a button or a quick reply is clicked. The postback is defined within the message event and is dealt with accordingly using a switch statement. When the condition is met, the suitable method is then called depending on what postback has been received. The following example illustrates how the chatbot would deal with a postback when a user initiates a conversation for the first time by clicking the `Get Started` button.

```
function receivedPostback(event) {

//Set variables
var senderID = event.sender.id;
var recipientID = event.recipient.id;
var timeOfPostback = event.timestamp;

// The 'payload' param is a developer-defined field which is set in a postback
// button for Structured Messages.
var payload = event.postback.payload;

switch (payload) {

    //messages sent if get started button is clicked
    case 'GET_STARTED' :

            var messageData = {
            recipient: {
                id: senderID
            },
            message: {
                text:"Hi, I am the ITB Chatbot",
                quick_replies:[
                    {
                        content_type :"text",
                        title : "What can you do?",
                        payload : "What can you do?"
                    },
                    {
                        content_type :"text",
                        title : "Who made you?",
                        payload : "Who made you?"
                    }
                ]
            }
        };
        callSendAPI(messageData);
        break;
```

```
            . . .
```

6.1.2 Handle Functions

Handle functions are primarily used when dealing with responses by the API.ai bot engine. These responses are in a JSON format and require specific algorithmic logic depending on the payload received. One of the highest level of the handle functions is the `handleMessage`, which checks the message type value passed as a parameter. This is then used to delegate which function should next be executed appropriately. These message types are defined by API.ai.

```
function handleMessage(message, sender) {
    switch (message.type) {

        //If it is text
        case 0:
            sendTextMessage(sender, message.speech);
            break;

        //if it a quick reply
        case 2:
            let replies = [];
            for (var i = 0; i < message.replies.length; i++) {
                let reply =  {
                    "content_type": "text",
                    "title": message.replies[i],
                    "payload": message.replies[i]
                }
                replies.push(reply);
            }
            //Send a quick Reply
            sendQuickReply(sender, message.title, replies);
            break;
        //Handle Custom payloads
        case 4:
            var messageData = {
                recipient: {
                    id: sender
                },
                message: message.payload.facebook
            };
            callSendAPI(messageData);
            break;
    }
}
```

Second to the previous function is the `handleApiAiAction` function. Only if an action is defined then this function will be called. When a action is caught in the switch statement, the appropriate method is taken. In this case, functions to call external API's are invoked (which in will be touched upon in a later section).

```
function handleApiAiAction(sender, action, responseText, contexts, parameters) {

    switch (action) {

        /*************** Dublin Bus Actions ******************/

        //Corduff bus stop
        case "corduff-route-picked" :
                    var busNum = contexts[0].parameters.bus_id;
                    getDublinBusTimes(sender,"1835", busNum);
            break;

                     . . .

        /*************** Library Actions *******************/

        case 'library-pin-not-enrolled' :
                getLibraryInfo(sender, action);

                    . . .

        /***************** Gym Actions *********************/

        case "gym-class-times-days-picked" :
                var dayPicked = contexts[0].parameters.gym_days;
                getGymInfo(sender, action, dayPicked);
            break;

                  . . .
        /****************************************************/
        default:
            //unhandled action, just send back the text
            sendTextMessage(sender, responseText);
    }
}
```

6.1.3 Send Functions

Send functions are used to send certain type of data to a particular end point and called when a certain content type is sent back from API.ai. These contain JSON templates to dispatch different response types ranging from images, button messages, quick replies, quick replies to other additional features likes sending read receipts or typing bubbles which was used to simulate human like characteristic to the chat. The most dominantly used of these functions is the `sendTextMessage` function, which provides JSON to reply with just a string of text. The function takes in the users unique ID and the text that was given back from API.ai and adds it to a JSON object accordingly. The `callSendAPI` function is then invoked, passing the JSON object.

```
function sendTextMessage(recipientId, text) {
    var messageData = {
        recipient: {
            id: recipientId
        },
        message: {
            text: text
        }
    }
    callSendAPI(messageData);
}
```

Another example of a send function would be `sendToApiAi`. Initially the sender ID and the message is passed. The `sendTypingOn` function is invoked to display a typing bubble on the interacting user's Messenger interface. A request is then sent to API.ai containing the text and the session ID assigned to that user. It then waits for a response from API.ai and sends the received data to the `handleApiAiResponse` function.

```
function sendToApiAi(sender, text) {

    //sends the typing bubble to the sender until
    //a response is given
    sendTypingOn(sender);

    //Send message to API.ai
    let apiaiRequest = apiAiService.textRequest(text, {
        sessionId: sessionIds.get(sender)
    });

    //Wait for response to API.ai
    apiaiRequest.on('response', (response) => {
        if (isDefined(response.result)) {
            handleApiAiResponse(sender, response);
        }
    });

    apiaiRequest.on('error', (error) => console.error(error));
    apiaiRequest.end();
}
```

6.1.4 API Call Functions

For each of the API's that have been developed, a function was made which is invoked from `handleApiAiAction`. A request is made using the `options` JSON object which contains the API server URL, applicable parameters and the HTTP method. When the request is made a callback function takes the response body and applies it to the `messageData` object, along with additional quick reply buttons to drive conversational flow. The `callSendApi` is then invoked to send the message back to the user. The example below displays a function that uses the bus API that has been developed.

```
function getDublinBusTimes(recipientId, stopId, busNum){

    var options = {
        url: "https://aaronapi.herokuapp.com/bus/" + stopId + "/" + busNum + "/",
        method : "GET"
    }
    //Make a request to the API
    request(options, function(error, res, body){

            var text = res.body;
            var messageData = {
                recipient: {
                    id: recipientId
                },
                message: {
                    text: res.body,
                    quick_replies:[
                        {
```

```
                        content_type :"text",
                        title : "Pick another Bus?",
                        payload : "Dublin Bus"
                    },
                    {
                        content_type :"text",
                        title : "Main Menu ",
                        payload : "Main menu"
                    },
                    {
                        content_type :"text",
                        title : "No thanks",
                        payload : "No thanks"
                    }
                ]
            }
        }
        callSendAPI(messageData);
    });
}
```

Lastly, the `callSendApi` function is the main function that connects the application with the Facebook interface. It makes a request to the Facebook Graph API (See Appendix A.4) using the page access token stored within `config.js` file. A post is made using the `messageData` JSON object which possesses the senders ID and the message to be sent. Additionally.

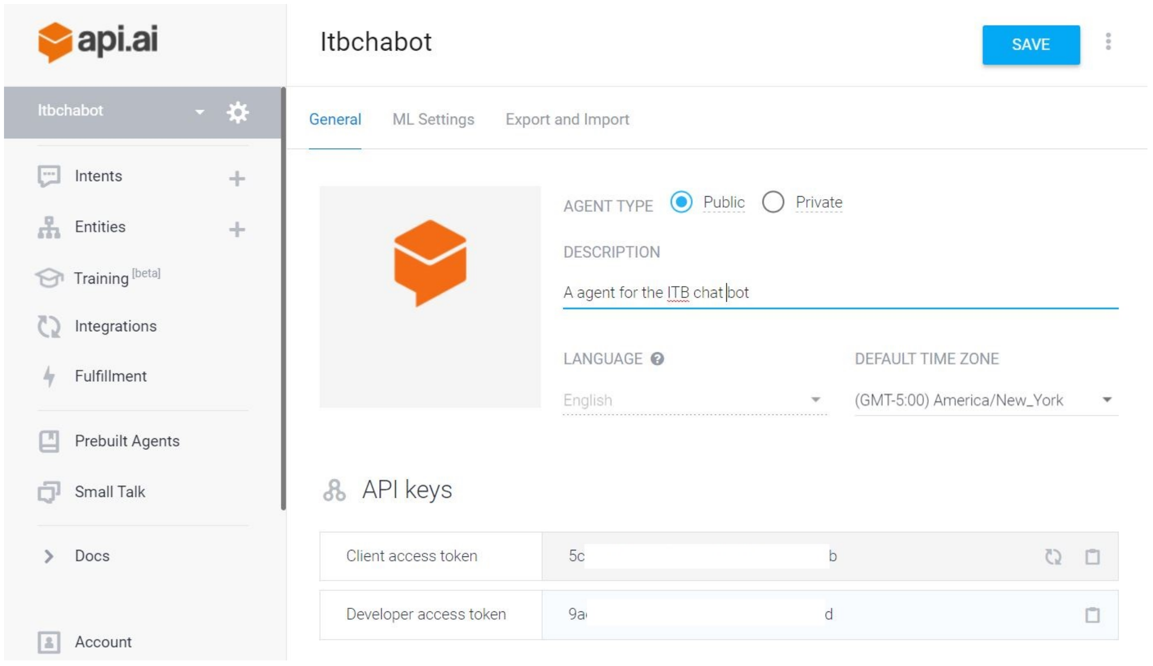
```
function callSendAPI(messageData) {
    request({
        uri: 'https://graph.facebook.com/v2.6/me/messages',
        qs: {
            access_token: config.FB_PAGE_TOKEN
        },
        method: 'POST',
        json: messageData
    }, function (error, response, body) {
        if(error){
            console.log(error);
        }else{
            console.log("Message Sent successfully");
        }

    });
}
```

In conclusion, the main server application consists of a range of functions to handle and distribute data between external entities. It also incorporated intuitive naming for a better understanding of the code. For further code listing, see the repository link in Appendix A.2.

6.2 Api.ai

An agent is first created using the Api.ai web interface named itbchatbot. A client access token and a developer access token are provided (Figure 6.1). The client access token was then stored to the `config.js` file to allow access to the remote API.ai agent from the chatbot server application. This agent is then used by the main application for the natural language processing and intent driving. Machine learning (ML) settings are then set to enforce iterative learn from data being sent to agent. Two options are given by the service: hybrid (Rule based and ML) or just standard ML. In this case, hybrid machine learning was implemented as it. In addition, a ML classification threshold is set to 0.2. The agent will only trigger an intent if it's confidence percentage is above this threshold value, otherwise a fallback intent will be invoked.



AGENT TYPE [✓ Public] Private

DESCRIPTION

A agent for the ITB chat bot

LANGUAGE: English

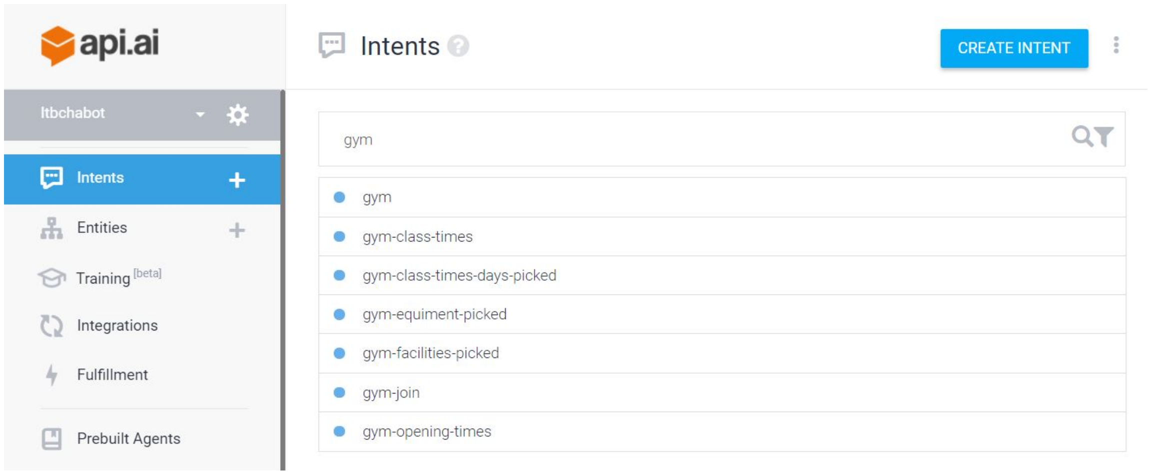
DEFAULT TIME ZONE: (GMT-5:00) America/New_York

Client access token: 5c b

Developer access token: 9a d

Figure 6.1: API.ai Agent Interface.

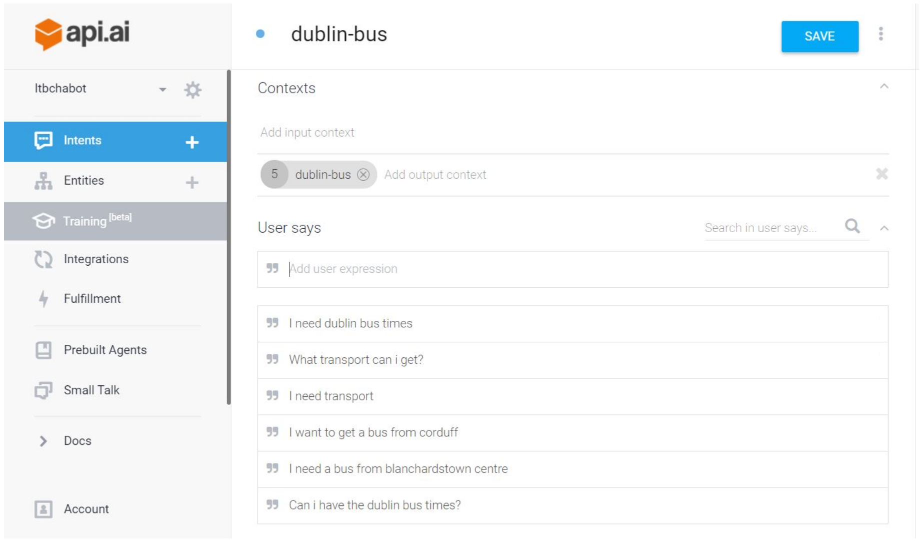
Intents were created using the API.ai web interface for conversational driving. For each service that the application provides (bus, gym and library) the intents are named fittingly. The follow-up intents are named after the service name followed by the operation it performs. This is shown in the figure 6.2 as an example. As seen below, the base intent `gym` is the initial starting point for the gym conversation and is then followed by a number of other intents that can be triggered depending on how the user directs the conversation.



gym
gym-class-times
gym-class-times-days-picked
gym-equiment-picked
gym-facilities-picked
gym-join
gym-opening-times

Figure 6.2: Intent Structure.

To create a base intent, the first step is to set the output context of the follow-up intents. This context is given a default life span of five but can be changed depending on approximately how long the discussion should last. This output context is used to link intents with one another and decrement the lifespan value each time a follow up intent is triggered. Secondly, expected user input is given by the developer. This involved entering several sentences that have the same meaning. The more expected sentences entered the more likely that the API.ai agent will be able to accurately extract semantics from user text. As shown in figure 6.3, it can be seen how a number of sentences entered with a similar meaning. The NLP algorithms extract the meaning from these sentences and can understand a text that doesn't match any of the sentences already specified.



I need dublin bus times
What transport can i get?
I need transport
I want to get a bus from corduff
I need a bus from blanchardstown centre
Can i have the dublin bus times?

Figure 6.3: Dublin Bus Intent.

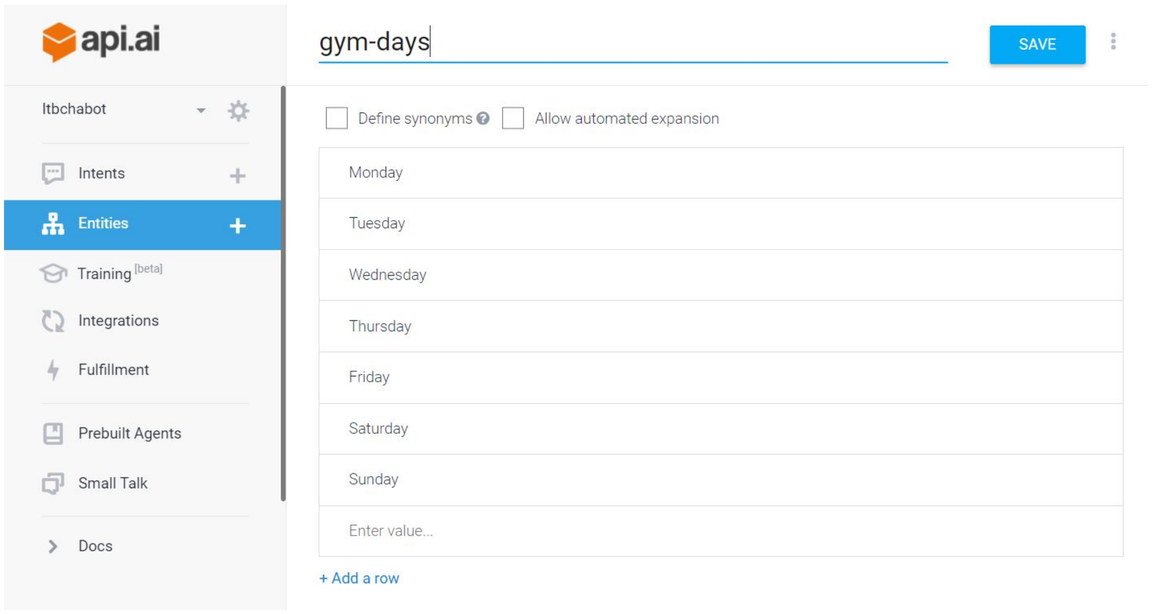
Responses were created for when a certain intent has been triggered. API.ai provides a number of templates to use including a text response, images, card template, quick replies and custom payloads to apply payload values to be returned when the user interacts with a content type. The following JSON snippet illustrates a custom payload used to design a generic template in addition with a number of quick reply buttons.

```
{
  "facebook": {
    "attachment": {
      "type": "template",
      "payload": {
        "template_type": "generic",
        "elements": [
          {
            "title": "Which bus do you want?",
            "image_url": <IMAGE_URL>
          }
        ]
      }
    },
    "quick_replies": [
      {
        "content_type": "text",
        "title": "All Routes",
        "payload": "All Routes"
      },
      {
        "content_type": "text",
        "title": "220 to Ballymun",
        "payload": "220 Towards Ballymun"
      }

        . . .

    ]
  }
}
```

Entities are then made for extracting values from payload responses from the users interaction with the bot. Entities have been made for acquiring the bus areas, such as Blanchardstown Shopping Centre and Corduff, and days for the gym classes. These entity attributes are sent back the main server application to pass these values to the external service APIs. Figure 6.4 shows the `gym-days` entity the attributes to be extracted from the users response.

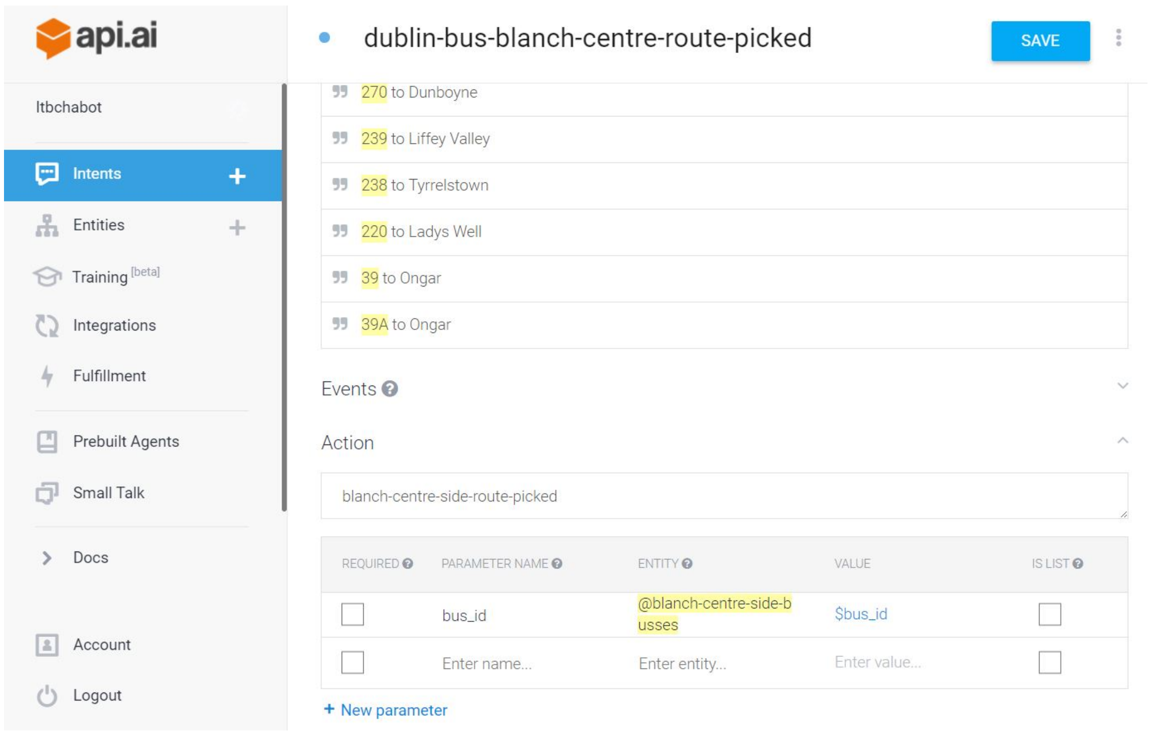


gym-days

Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
Sunday
Enter value...

Figure 6.4: Gym Entity.

When the user reaches the end of a conversation flow, no response from API.ai is set and an action must be implemented to be caught by the `handleApiAiAction` function in the main application code. This action indicates that a conversation has reach the end point and now must retrieve data from one of the three APIs. An action encompasses extracting a value from the users response and assigning a parameter name. This parameter name which is given an entity value, and the action name is sent back to the chatbot application.For instance, in an ending intent of a Dublin bus conversation, the action with the value of `blanch-centre-side-route-picked` is set. Additionally, the bus route picked is assigned to the `bus_id` parameter and no response is created (Figure 6.5).



dublin-bus-blanch-centre-route-picked

270 to Dunboyne
239 to Liffey Valley
238 to Tyrrelstown
220 to Ladys Well
39 to Ongar
39A to Ongar

Events

Action

blanch-centre-side-route-picked

REQUIRED	PARAMETER NAME	ENTITY	VALUE	IS LIST
☐	bus_id	@blanch-centre-side-busses	$bus_id	☐
☐	Enter name...	Enter entity...	Enter value...	☐

Figure 6.5: Dublin Bus Action.

6.3 Application Program Interfaces

As previously mentioned in the system design, three application program interfaces were developed to provide additional information. Each API was hosted on Heroku and run entirely autonomously from the main application. The following section provides a detailed explanation of the implementation of bus API and because of the similar implementation steps for the remaining other API's (gym and library) the code for their implementation will be excluded from this document. However, should the reader wish to see these implementations, see Appendix A.2. All examples shall be aided with code segments to further exemplify the API's functionalities and capabilities.

6.3.1 Bus API

The initial dependencies are imported to the `server.js` file. These include the `express`, `request` and the `body-parser` node modules. Port configurations are declared to allow the API to run on Heroku with a dynamically allocated port number.

```
var express = require('express');
var bodyParser = require('body-parser');
var request = require('request');
```

```
    var app = express();

        . . .

    app.set('port', (process.env.PORT || 5000))

    app.get('/', function (req, res) {
        res.send('This is the landing page for the dublin bus API for the chatbot...')
    })

    app.listen(app.get('port'), function () {
        console.log('Bus API server is running on port ', app.get('port'))
    })
```

As has been aforementioned in implementation of the chatbot application server code, the call to the bus API sends two parameters. The bus number parameter is extracted from API.ai and sent to the `handleApiAiAction`. Also, depending on what action is caught, the bus stop ID of the corresponding bus area is also passed and added as a parameter to the URL.

```
    var options = {
        url: "https://aaronapi.herokuapp.com/bus/" + stopId + "/" + busNum + "/",
        method : "GET"
    }
```

The API uses these values and stored them to a JSON object to be utilised within the program. Should the value of the bus ID be set to "All" then a Boolean value is set to true. This is used to get all the estimated bus times for all routes at that bus stop. Additionally, a check for the 39 or 39A bus is put in place, as there is two bus stops with separate stop IDs at the Blanchardstown Centre side bus stop. This will change the value of `stopId` accordingly.

```
app.get('/bus/:stop_id/:bus_num', function(req, res) {

        /**
         * Assign parameters passed in URL to
         * a JSON object
         */
        var data = {
            "bus": {
                "stop_id": req.params.stop_id,
                "bus_num": req.params.bus_num
            }
        };
        var message = "";

        //All routes button picked by user
        var all = false;

        /**
         * Assign stop id and bus number to the values
         * passed by chat bot
         */
        var stopId = data.bus.stop_id;
        var busNumber = data.bus.bus_num;

        if(busNumber == "All"){
            all = true;
        }

        //Check because 39a and 39 are at a different stop
        if(stopId == "7026"){
            if(busNumber == "39"|| busNumber == "39A"){
                stopId = "7025";
            }
        }
```

A JSON object named `options` is then initialised with the URL for the RTPI REST Web Services API, the HTTP GET method and strictSSL set to false to allow responses from an uncertified source. Furthermore, a request is made and an error checking is present to log any errors should they occur. If there is no error, the body of the response is parsed to a JSON format. When the `numberofresults` parameter has a value of 0, this usually indicates that the bus services are not in service because of the time of day. The response message is given a value telling the user that it is too late to check the times for the bus.

```
    var options = {
        url: 'https://data.dublinked.ie/cgi-bin/rtpi/realtimebusinformation?stopid=' + stopId + '&fo
        method : 'GET',
        strictSSL: false
    };

    /**
        * Make a request to for Dublin bus API and repond to chatbot
        */
    request(options, function(error, response, body) {
    if(error){
        console.log(error);
    }
    else{

        body = JSON.parse(body);
        //numberofresults will return as 0 if it past half 11
        if(body.numberofresults === 0){
            message = "It's too late for busses ";
        }
```

Should there be a number of results, the results set is traversed. A guard to check if the `route` is the same as the one passed from the main application code or if the `All` variable is set to true. This is to ensure that only the bus route specified or all the bus routes are extracted. The value of the bus route and the number of minutes until it is due is concatenated to a message string over each loop of iteration. Additional checks are put in place to verify that the message being sent back makes sense. For instance, should the `duetime` value be equal to one, then the message is changed from "due in X minutes" to "due in 1 minute". Additionally, when the loop of the result set has finished executing, a counter for the results are checked. During each iteration, this value was incremented. If the value has a value of zero prior to the loop, this shows that there is no bus due at the bus stop specified. Lastly, the resulting message is then send back to the main application and dealt with accordingly.

```
    else{
        var resultCount = 0;
        //Display all the bus routes and due times available
        for( var i in body.results){
            if(body.results[i].route == busNumber || all == true){
                //If the bus is due now, dont display "due in due minutes"
                if(body.results[i].duetime === "Due"){
                    message +=  body.results[i].route + " to " +
                                body.results[i].destination + " due now\n";

                }
                //Stop 1 minute appearing as "1 minutes"
                else if(body.results[i].duetime === "1"){
                    message +=  body.results[i].route + " to " +
                                body.results[i].destination + " due in " +
                                body.results[i].duetime + " minute\n";
                }
                else{
                    message +=  body.results[i].route + " to " +
                                body.results[i].destination + " due in " +
                                body.results[i].duetime + " minutes\n";
                    }
                    resultCount++;
                }
            }
            //Check if there is not times available
            if(resultCount === 0){
                message = "There is no times available for " + busNumber + "  ";
            }

            //Send the result back to the requester
            res.send(message);
            }
        }
    });
```

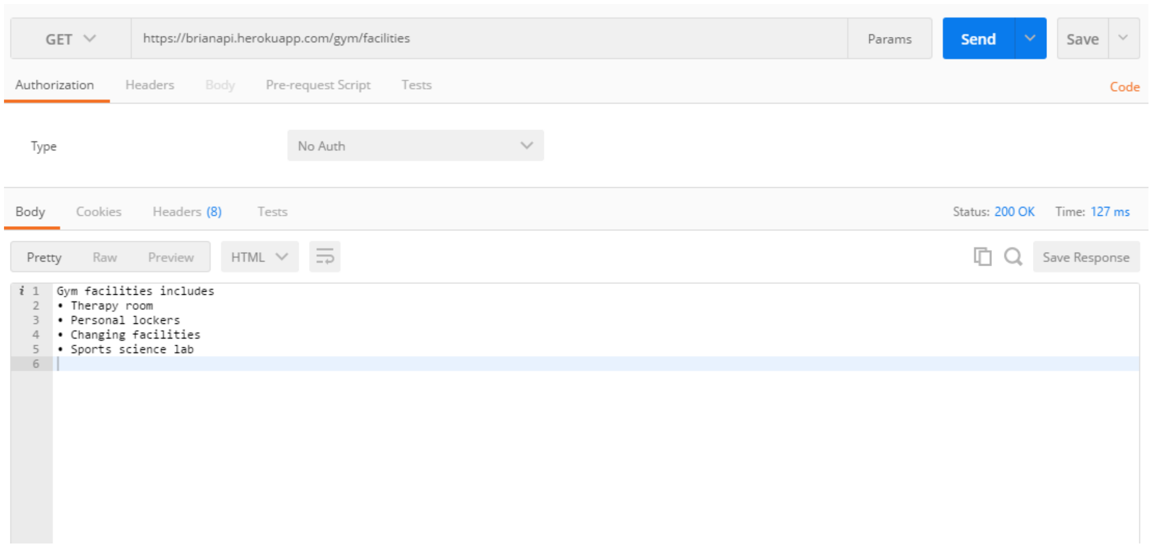
To conclude, the API successfully takes in values sent by the chatbot application and returns the appropriate real time information in a suitable and human like message. See appendix A.5 for screenshots of working application.

7 Testing and Evaluation

Testing and evaluation is a crucial and important part of any piece software to identify any bugs, errors or inconsistencies. This section will discuss and highlight the testing and evaluation of the chatbot system. In addition, it will detail the tools and testing methods used to examine the functional correctness and usability of the chatbot. The testing phase was carried out to determine and ensure that the chatbot functioned in the appropriate manner and met the functional requirements as aforementioned. The tests were implement based on two types of testing methods which are API and Usability testing. The API category of testing encompasses testing an application program interface to examine its performance and dependability. The means in which these tests were carried was through the API testing tool Postman, which provides the capability for sending HTTP requests and reading HTTP responses. In addition, the Usability testing technique comprises of a non-functional testing method in which it processes how easy the application can be interacted with the end users (Tutorialspoint, 2017). This was done by adding a select group of testers to the Facebook messenger platform in which the chatbot was made accessible to them.

7.0.1 API Testing

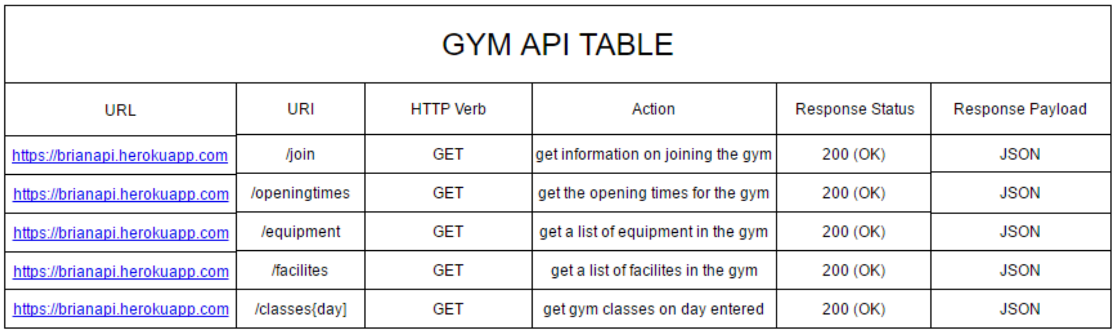
API testing was incorporated to establish if each RESTful API implemented into the chatbots framework returned accurate responses based on each HTTP request sent. To accomplish this task a testing environment was setup. Which encompassed a setting in which tests can be carried out and a system whereby to document each test. As aforementioned the API tests were carried out using Postman which is an API testing tool. Postman provides an environment in which to interact with API's to construct HTTP requests and read HTTP responses. The tests constructed within Postman were designed to examine endpoints in each RESTful API with the required set of parameters using the HTTP GET function as each RESTful API only returns data. In order to successfully document all the tests carried out on each API endpoint tables were constructed to contain all test conditions and results for each API. In addition, to test in the Postman environment there are a number of parameters that must be provided, these include a URL, URI and a HTTP method. Frigure 7.1 below displays an example of a test performed in Postman.



```
Gym facilities includes
• Therapy room
• Personal lockers
• Changing facilities
• Sports science lab
```

Figure 7.1: Example of test in Postman environment.

In order to successfully document all the tests carried out on each API endpoint tables were constructed to contain all test conditions and results for each API. See figure 7.2 for an example of the test documentation.



GYM API TABLE

URL	URI	HTTP Verb	Action	Response Status	Response Payload
https://brianapi.herokuapp.com	/join	GET	get information on joining the gym	200 (OK)	JSON
https://brianapi.herokuapp.com	/openingtimes	GET	get the opening times for the gym	200 (OK)	JSON
https://brianapi.herokuapp.com	/equipment	GET	get a list of equipment in the gym	200 (OK)	JSON
https://brianapi.herokuapp.com	/facilites	GET	get a list of facilites in the gym	200 (OK)	JSON
https://brianapi.herokuapp.com	/classes{day]	GET	get gym classes on day entered	200 (OK)	JSON

Figure 7.2: Table of documented tests.

7.0.2 Usability Testing

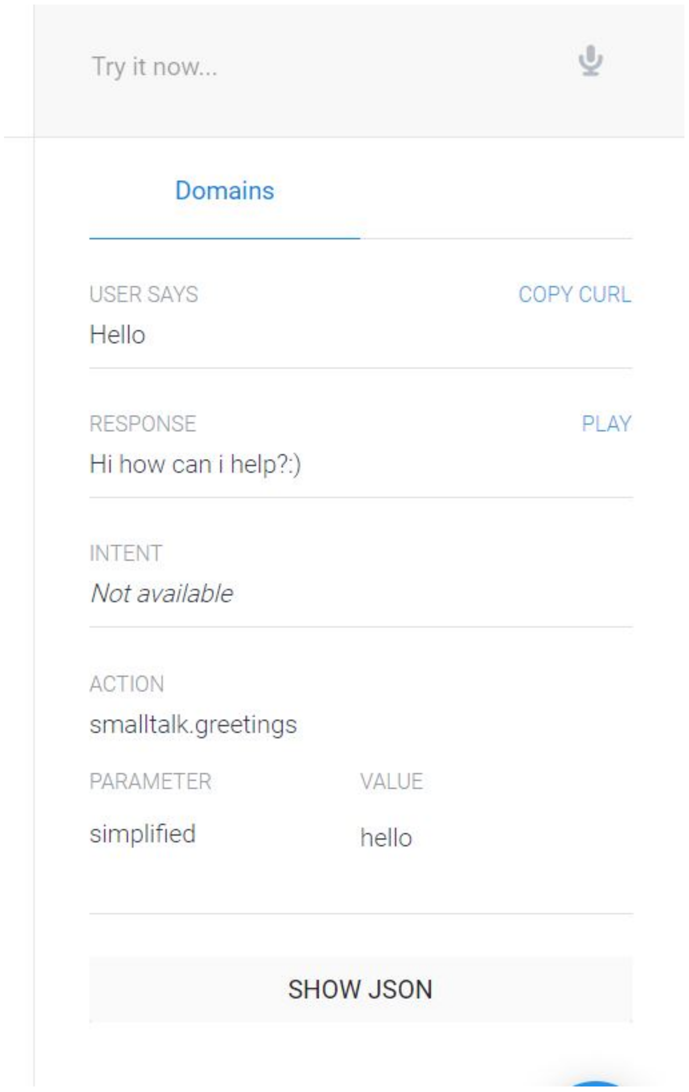
Usability testing was implemented to examine the manner in which the end user interacts with the chatbot. Additionally, this form of testing was incorporated to determine and quantify the chatbots ease of use and the overall experience of the user while interacting with the application. As above mentioned this task was achieved by adding a select number of users as testers to the platform were the chatbot is housed. In addition, the users were observed interacting with the chatbot in an informal setting, thus making them more at ease and comfortable to suggest change. Changes were made to the chatbot in terms of conversational flow. A more button driven approach was taken due to the response of the testers. See appendix 1.5 for screenshots of user interactions.

7.1 Training API.ai

Training the API.ai agent is an important vital component of ensuring that the right intents are triggered when interacting with the chatbot. API.ai provides a testing console for developers to perform test conversations with the agent on a web interface. The console displays the sentence entered by the tester, the response that the agent returns, the intent that was triggered based on the semantics extracted from the input, a context of the conversation that has been instigated, the appropriate action set in the intent triggered (if any) and the parameters name of the value extracted from the user input (See figure 7.3) .

As shown above, the correct response was given back to the user. No intents or contexts appear as the there is no follow up intents to succeed the response given. Also, a `smalltalk.greeting` action is from the SmallTalk domain that API.ai provides. In the case of a conversational flow, more attributes from developer defined intents will be shown. As seen in figure 7.4, in the case of a user going through a bus conversation, the `dublin-bus` and `picked-blanch-side` contexts can be seen. This allows for developers to verify that the right intent is triggered when an input is given to the agent.

In the case that the wrong intent is triggered from a certain input, API.ai provides a machine training tool that keeps a history of the tested inputs. When wrong intents were triggered, the training tool was utilised to ensure that the machine learning algorithms of API.ai can remember what intent to trigger should a similar input be given again. As shown in figure 7.5, a test input with the sentence "can I have gym closing times?" was entered. As there was no classification of what this sentence meant, there was no intent to trigger. Therefore, the `gym-opening-times` was assigned using the training tool. Through machine learning, the agent can remember this input and can handle a similar sentence should it be received again.



USER SAYS
Hello

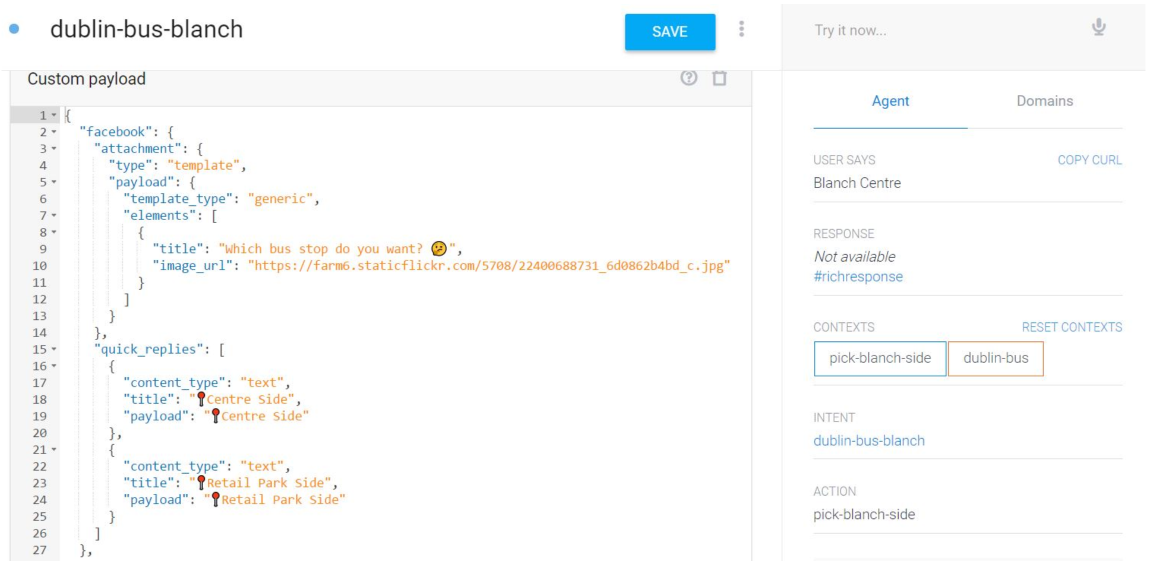
RESPONSE
Hi how can i help?:)

INTENT
Not available

ACTION
smalltalk.greetings

PARAMETER	VALUE
simplified	hello

Figure 7.3: API.ai Testing Console.



dublin-bus-blanch

SAVE

Custom payload

```
{
  "facebook": {
    "attachment": {
      "type": "template",
      "payload": {
        "template_type": "generic",
        "elements": [
          {
            "title": "Which bus stop do you want? 🤔",
            "image_url": "https://farm6.staticflickr.com/5708/22400688731_6d0862b4bd_c.jpg"
          }
        ]
      }
    },
    "quick_replies": [
      {
        "content_type": "text",
        "title": "📍Centre Side",
        "payload": "📍Centre Side"
      },
      {
        "content_type": "text",
        "title": "📍Retail Park Side",
        "payload": "📍Retail Park Side"
      }
    ]
  },
```

Try it now...

Agent | Domains

USER SAYS: Blanch Centre

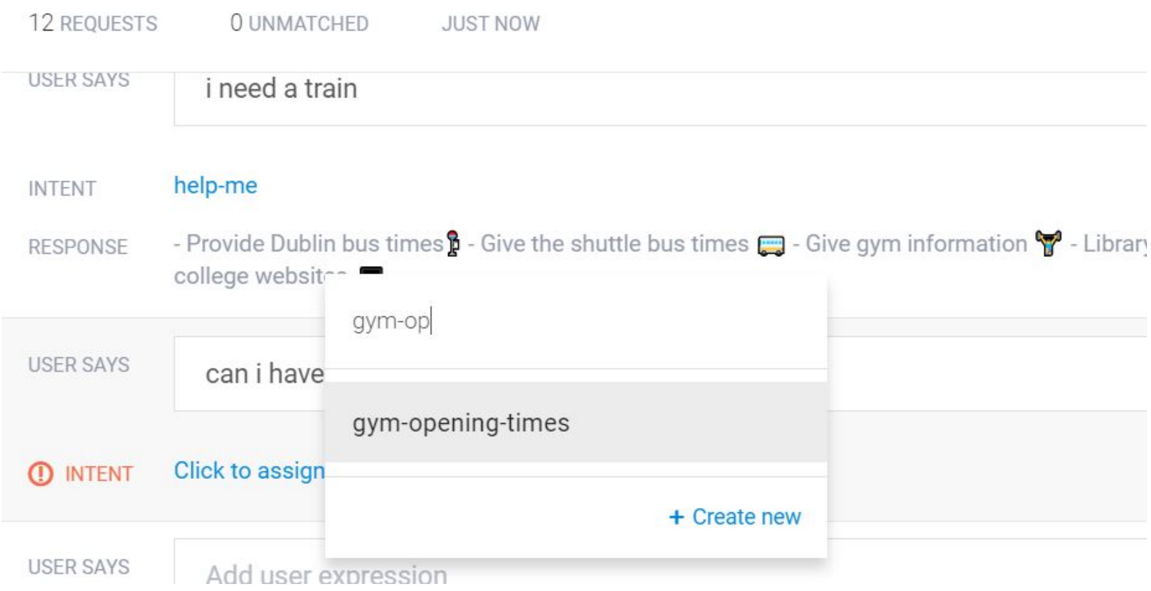
RESPONSE: *Not available* #richresponse

CONTEXTS: pick-blanch-side, dublin-bus

INTENT: dublin-bus-blanch

ACTION: pick-blanch-side

Figure 7.4: API.ai Testing Console 2.



12 REQUESTS 0 UNMATCHED JUST NOW

USER SAYS: i need a train

INTENT: help-me

RESPONSE: - Provide Dublin bus times🚏 - Give the shuttle bus times 🚌 - Give gym information 🏋 - Librar
college websit

gym-op

gym-opening-times

+ Create new

USER SAYS: can i have

INTENT: Click to assign

USER SAYS: Add user expression

Figure 7.5: API.ai Training Tool.

8

Conclusions and Further Work

8.1 Conclusion

This project looked at implementing a chatbot application and investigating the necessary technologies and tools for development. This project was centred around creating a chat application that could aid college students with frequently-asked-questions that they may have in relation to their college life. An investigation was carried out as to what the available messaging platforms were for deployment of the chatbot application, which NLP tools were available to give the chatbot the capability of responding to users intelligently and also other development tools that were highly beneficial in the development of this application. This project was successful in decoupling each feature into its own RESTful service and provided a modularity that nullified any single-point-of-failure within the application and led to a satisfactory separation of responsibility amongst the development team. Whilst, perhaps more measures could have been taken with continuous integration(CI) with the Heroku platform, to nullify build/deployment failures, it was decided amongst the team that it was outside of the scope of this project but will definitely be explored for future implementations of this system.

8.2 Further work

Although the development of the chatbot application was successful, there is a few alternative approaches to appraise should the opportunity arise again to recreate the project. One notable consideration is the utilisation of the a continuous integration tool like Travis CI. Travis CI allows team members team members integrate code to a repository and run automated builds. This would prove advantageous as it tests the changes that are being made and prevents defective code to be deployed to the application. Another considerable approach to the development of the chatbot is to implement a more comprehensive natural language processing agent. As there are minor conversational pieces that can be obtained from the chatbot, the conversations are mostly button driven. It would be desired to give a balance between button driven and text driven conversations. Furthermore, the ability to provide other campus information like real time class room or parking space availability would also prove to be another interesting endeavour.

A Appendices

This section provides supplementary material such as surveys and GitHub repository links, API server URLs and documentation for the tools implemented in the project. Screenshots of the completed implementation are also provided.

A.1 Appendix 1 - Survey Links

Transport Survey: https://www.surveymonkey.com/r/WTVDY8H

Educational Facilities Survey: https://www.surveymonkey.com/r/YPPYR77

A.2 Appendix 2 - Github Repositories

Main application repository: https://github.com/CoderYoyoS/Chatbot_Assignment

Library API repository: https://github.com/CoderYoyoS/Daire_api

Gym API repository: https://github.com/CoderYoyoS/Brian_api

Bus API repository: https://github.com/CoderYoyoS/Aaron_api

A.3 Appendix 3 - Heroku App Links

Main application link: https://itbchatbot.herokuapp.com

Library API link: https://daireapi.herokuapp.com/library/

Gym API link: https://brianapi.herokuapp.com/gym/

Bus API link: https://aaronapi.herokuapp.com/

A.4 Appendix 4 - Tool Documentation

Facebook Documentation: https://developers.facebook.com/docs/messenger-platform

API.ai Documentation: https://docs.api.ai/

Heroku Documentation: https://devcenter.heroku.com/articles/nodejs-support

A.5 Appendix 5 - Application Screenshots

Figure A.1 - Welcome

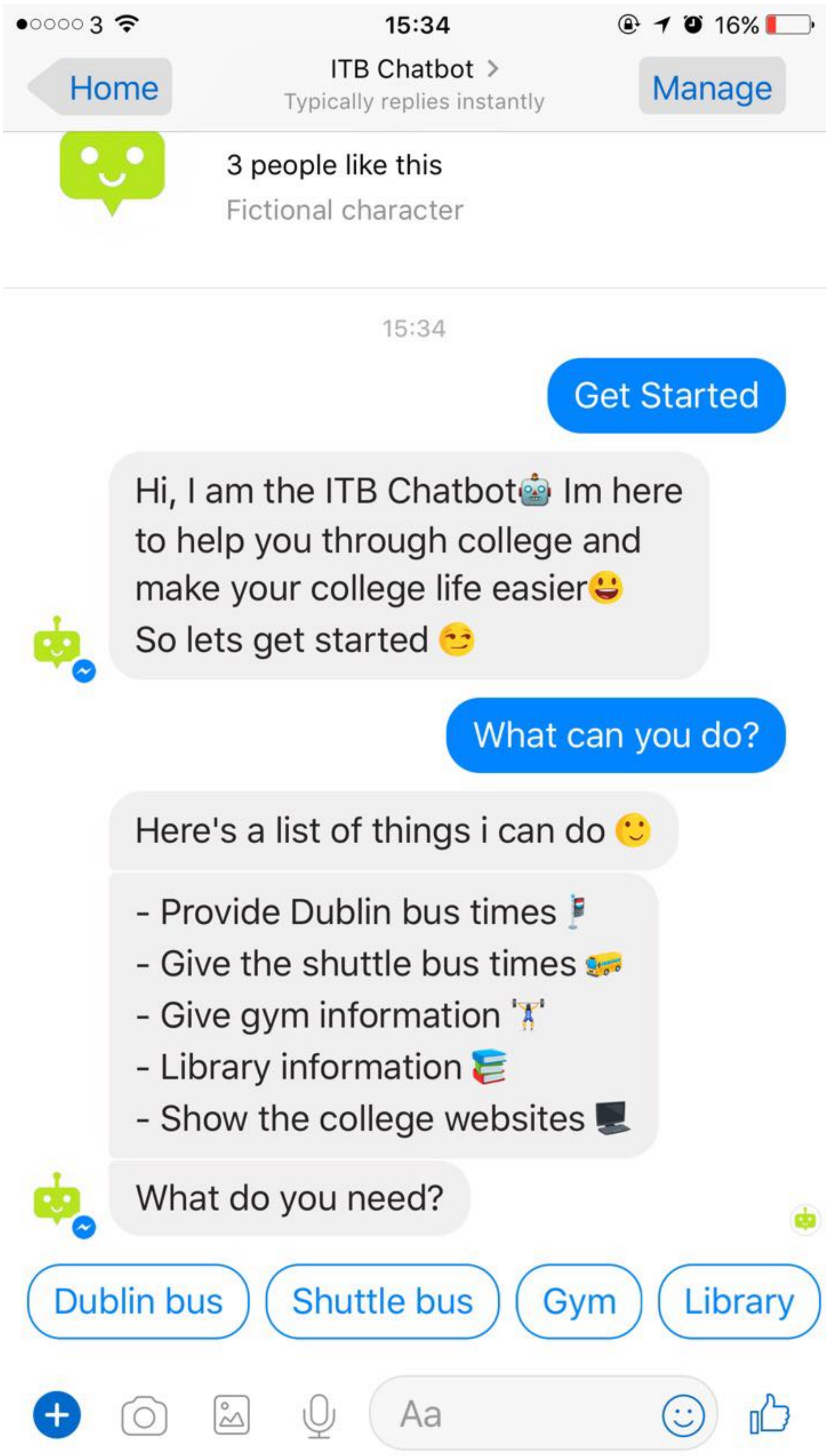
Figure A.2 - Bus

Figure A.3 Library

Figure A.4 Gym

Figure A.5 Pin

Figure A.6 Classes



Get Started

Hi, I am the ITB Chatbot🤖 Im here to help you through college and make your college life easier😃
So lets get started 😏

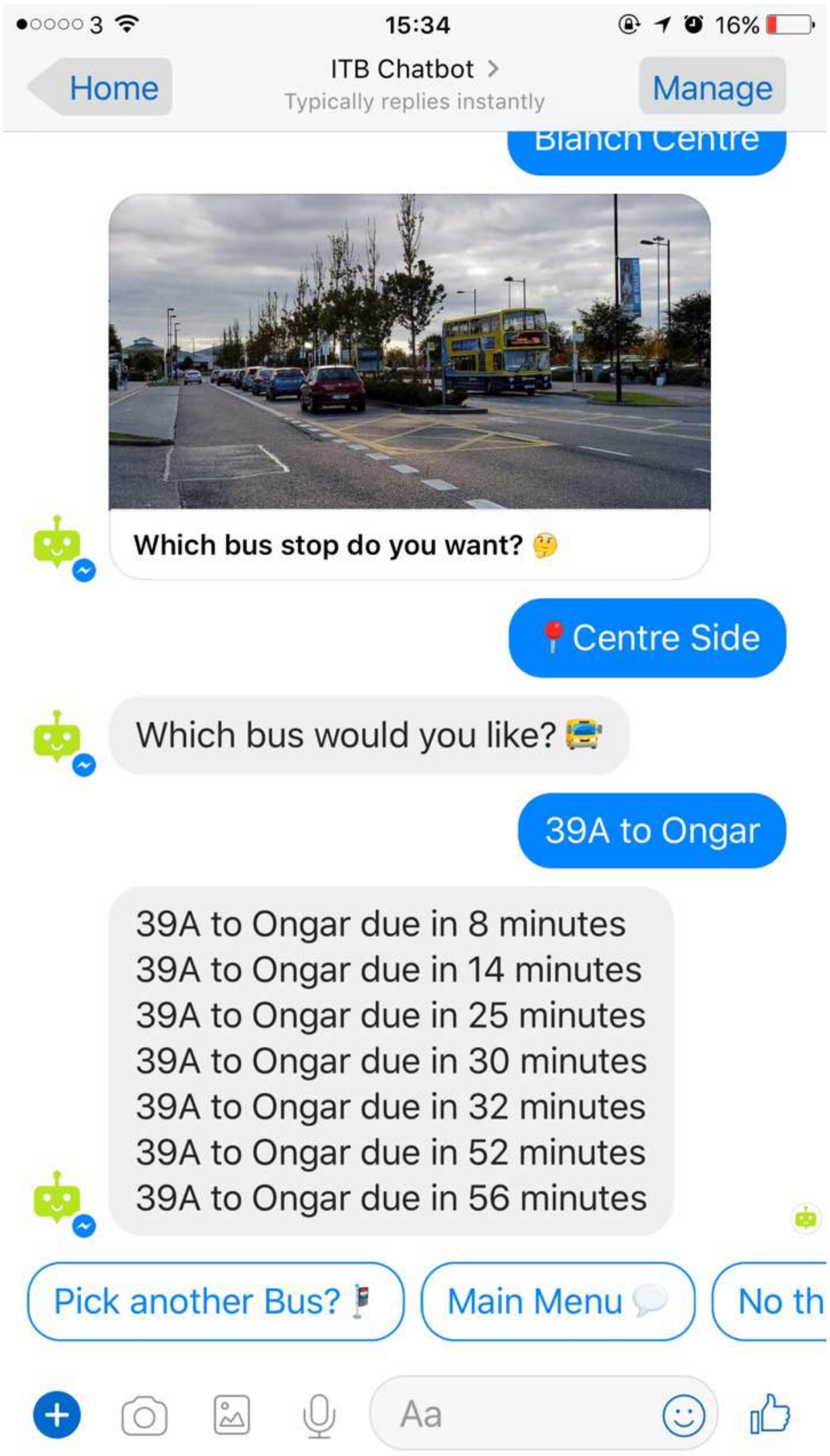
What can you do?

Here's a list of things i can do 🙂

- Provide Dublin bus times 🚏
- Give the shuttle bus times 🚌
- Give gym information 🏋
- Library information 📚
- Show the college websites 💻

What do you need?

Figure A.1: Welcome.



Which bus stop do you want? 🤔

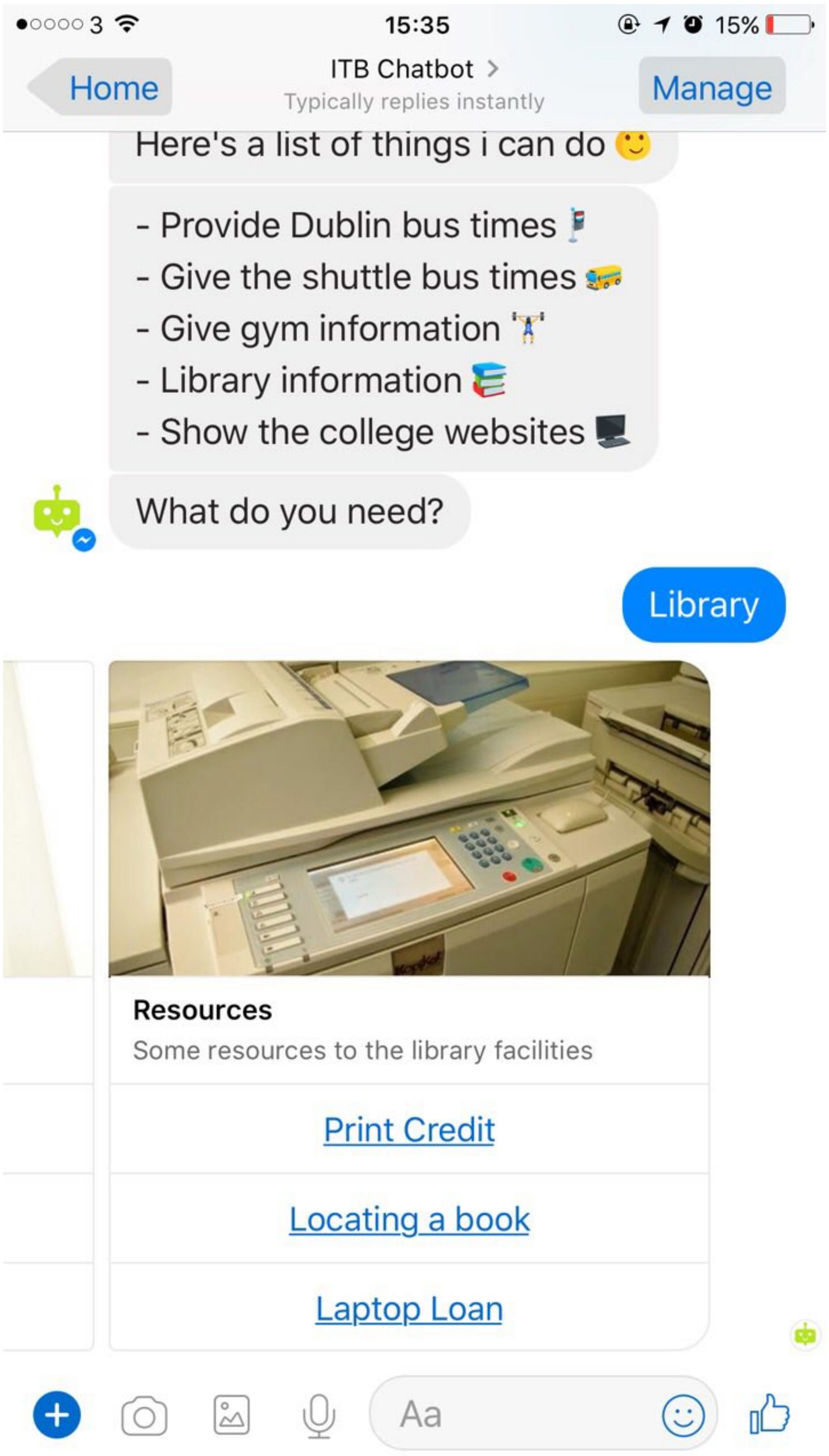
📍Centre Side

Which bus would you like? 🚌

39A to Ongar

39A to Ongar due in 8 minutes
39A to Ongar due in 14 minutes
39A to Ongar due in 25 minutes
39A to Ongar due in 30 minutes
39A to Ongar due in 32 minutes
39A to Ongar due in 52 minutes
39A to Ongar due in 56 minutes

Figure A.2: Bus.



●○○○○ 3 15:35 15%

Home ITB Chatbot > Manage

Typically replies instantly

Here's a list of things i can do 🙂

- Provide Dublin bus times 🚏
- Give the shuttle bus times 🚌
- Give gym information 🏋
- Library information 📚
- Show the college websites 💻

What do you need?

Library

Resources

Some resources to the library facilities

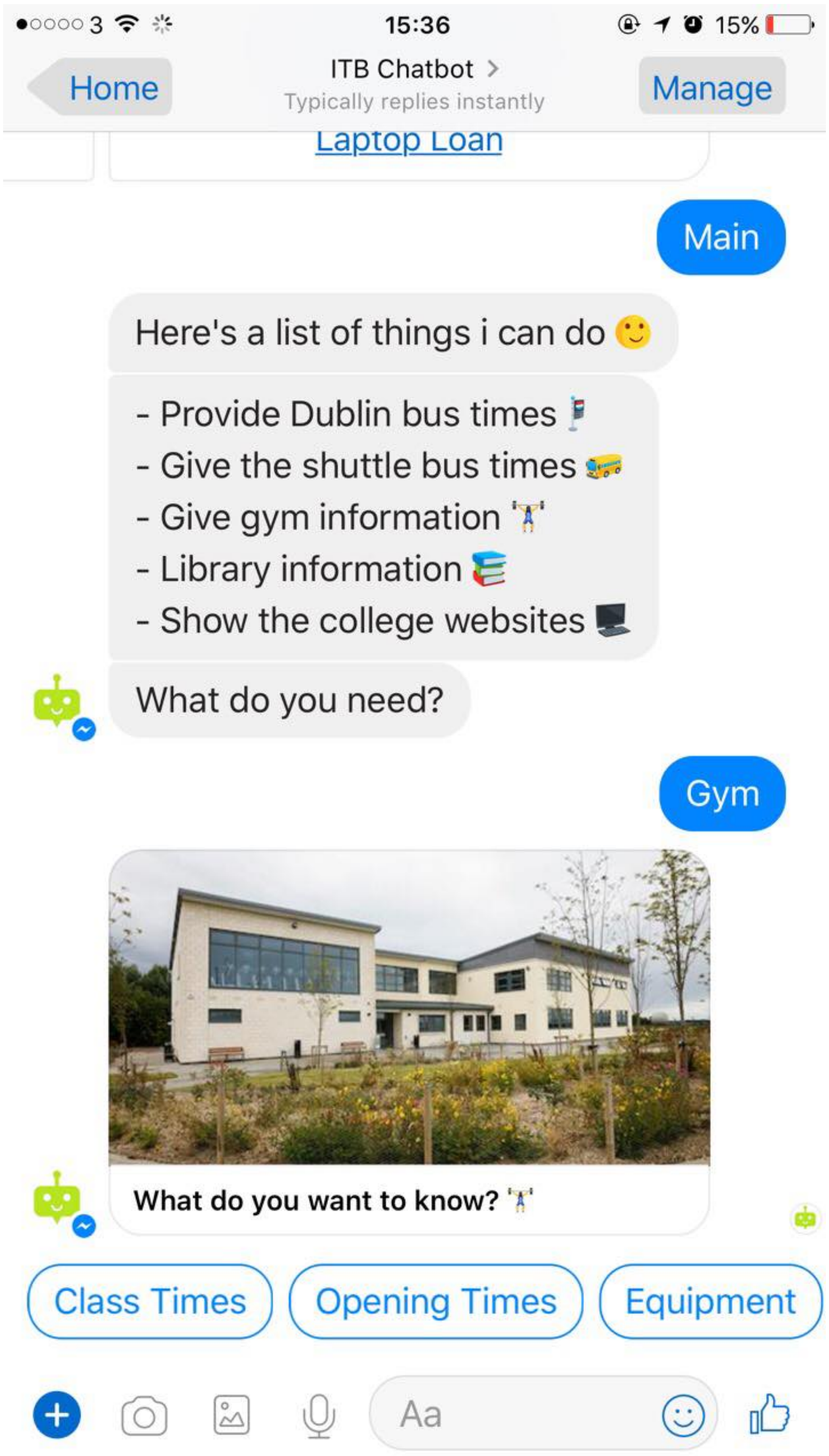
Print Credit

Locating a book

Laptop Loan

Aa

Figure A.3: Library.



Main

Here's a list of things i can do 🙂

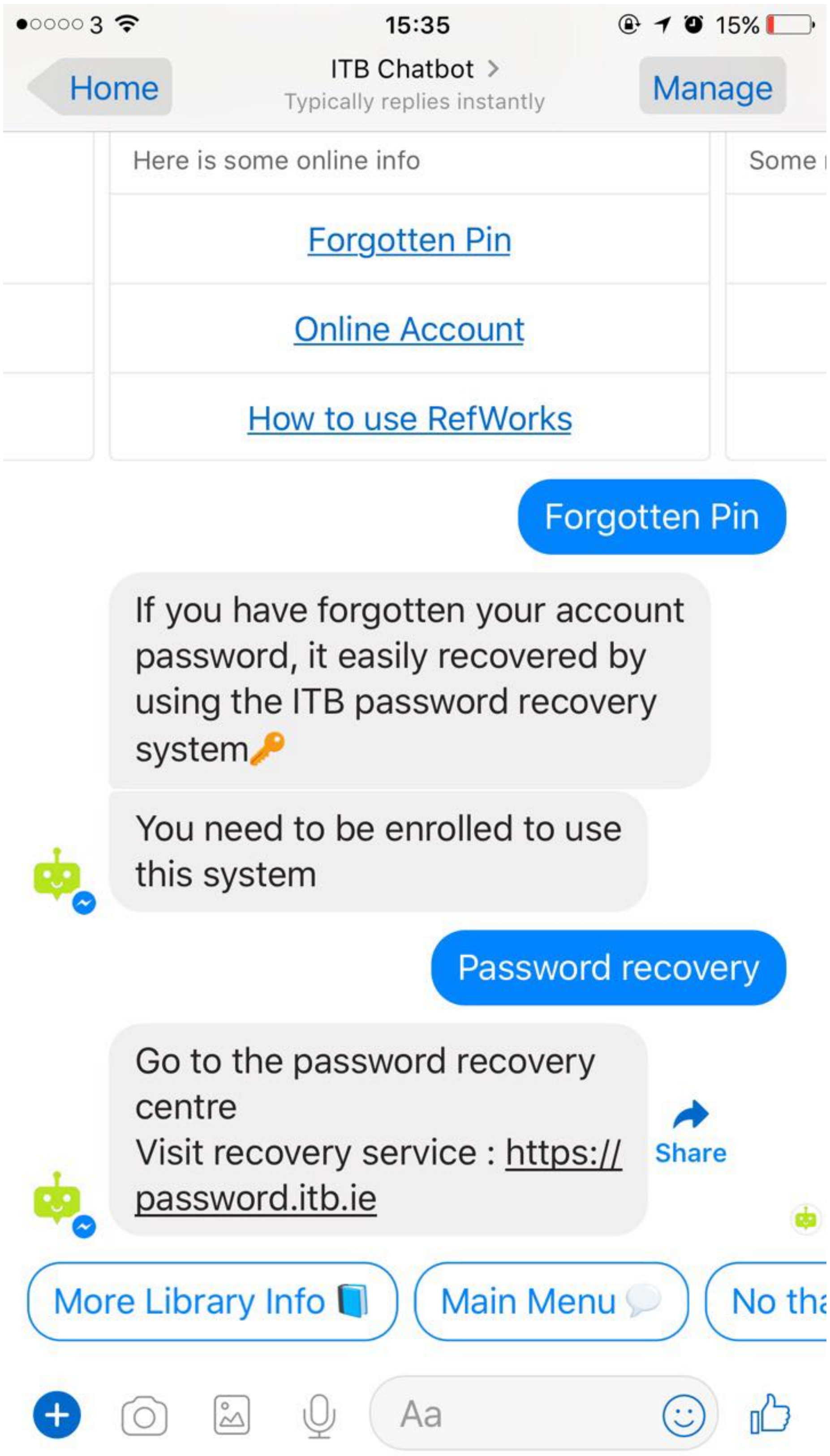
- Provide Dublin bus times 🚏
- Give the shuttle bus times 🚌
- Give gym information 🏋
- Library information 📚
- Show the college websites 💻

What do you need?

Gym

What do you want to know? 🏋

Figure A.4: Gym.



Here is some online info

Forgotten Pin

Online Account

How to use RefWorks

Some

Forgotten Pin

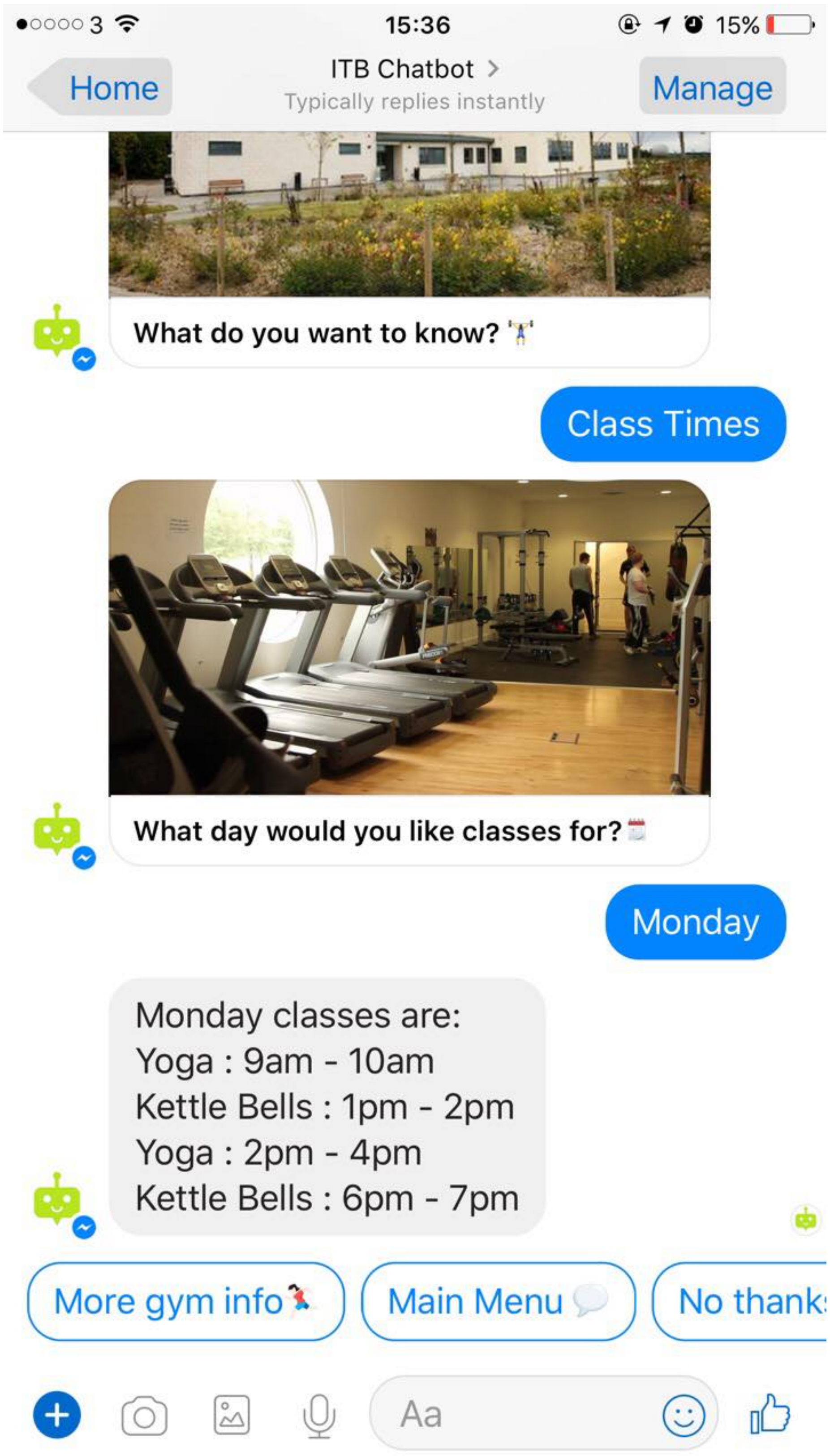
If you have forgotten your account password, it easily recovered by using the ITB password recovery system🔑

You need to be enrolled to use this system

Password recovery

Go to the password recovery centre
Visit recovery service : https://password.itb.ie

Figure A.5: Pin.



●○○○○ 3 15:36 15%

Home ITB Chatbot > Typically replies instantly Manage

What do you want to know? 🏋

Class Times

What day would you like classes for? 📅

Monday

Monday classes are:
Yoga : 9am - 10am
Kettle Bells : 1pm - 2pm
Yoga : 2pm - 4pm
Kettle Bells : 6pm - 7pm

Figure A.6: Classes.

A.6 Appendix 5 - How this report was written

This report was written using Pandoc, a tool used for converting file formats and enabling quick collaborative development on documents.

Visit Pandoc website here: http://pandoc.org/

Visit the repository where this report was developed: https://github.com/CoderYoyoS/ChatbotReport

A.7 Appendix 6 - API Documentation Tool

All API's developed in this project were documented using Swagger, a tool that helps to design, build and document a RESTful API

Visit Swaggers website: http://swagger.io/

References

Dale, R. (2016) The Return Of The Chatbots. Natural Language Engineering 22.05, 811-817.

Best, J. (2017) IBM Watson: The Inside Story Of How The Jeopardy-Winning Supercomputer Was Born, And What It Wants To Do Next. Retrieved from http://www.techrepublic.com on 6 Mar. 2017.

Evans, B. (2017) Chat Bots, Conversation And AI As An Interface. Retrieved from http://www.benevans.com on 03 Apr. 2017.

Deo, V. (2016) Bots Vs Apps, Which Side Are You On. Retrieved from http://www.chatbotsmagazine.com on 05 Apr. 2017.

Ranger, S. (2016) Chat Bots: How Talking To Your Apps Became The Next Big Thing. Retrieved from http://www.zdnet.com on 05 Apr. 2017.

Cresci, E. (2017) Chatbot That Overturned 160,000 Parking Fines Now Helping Refugees Claim Asylum.Retrieved from http://www.theguardian.com. on 12 Apr. 2017.

Hirschberg, J. and Manning, D. (2015) Advances in Natural Language Processing, 349(6245), 261-266.

Interoute (2017) What is PAAS? Retrieved from: http://www.interoute.com on 16 April 2017.

Issac, L.P. (2014) SQL v NoSQL Database Differences Explained. Retrieved from http://www.thegeekstuff.com on 24 April 2017.

Otte, S (2009) Version Control Systems, (Computer Systems and Telematics). Institute of Computer Science Freie Universitat Berlin. Retrieved from https://pdfs.semanticscholar.org on 11 April 2017.

Heroku (2017) Dyno Types Retrieved from: https://devcenter.heroku.com/articles/dyno-types on 03 April 2017.

Heroku (2017) Add-ons Retrieved from: https://devcenter.heroku.com/articles/add-ons. on 03 April 2017.

Heroku (2017) Heroku Local Retrieved from: https://devcenter.heroku.com/articles/heroku-local on 03 April 2017.

Carey Warehouse (2017) Should You Use MongoDB? A Look at the Leading NoSQL Database. Retrieved from: https://www.upwork.com/hiring/data/should-you-use-mongodb-a-look-at-theleading-nosql-database/. on 05 April 2017.

MongoDB (2017). Sharding. Retrieved from: https://docs.mongodb.com/manual/sharding/ on 4 May 2017.

Wong, J. (2017) What Is A Chat Bot, And Should I Be Using One. Retrieved from: https://www.theguardian.com/technology/ on 6 Apr 2017.

Constine, J. (2016) Facebook Launches Messenger Platform With Chatbots. Retrieved from: https://techcrunch.com/ on 6 Apr 2017.

Bryan , A (2016). NLP vs. NLU: What's the Difference? Medium.com. Retrieved from: https://medium.com/@lolatravel/nlp-vs-nlu-whats-the-difference on 5 April 2017