Botman

An enterprise-grade framework for chatbot development

Console Web Application and Chatbot Framework Dashboard

Functional Specification

Version 0.1

Aditi B. Shetty
(aditi.shetty@sjsu.edu)

Table of Contents

Version History

Version	Changes
1.0	Initial Draft

Introduction

Botman is an enterprise-grade framework for chatbot development. It allows the user to build, deploy and manage chatbots. The chatbots provide a highly engaging and conversational experience. This can be customized as per user requirement through voice and text.

In addition to this, a dashboard can be maintained by the user. A dashboard will provide a visual display of the important and relevant user data. The displayed data usually contains performance measures and metrics which will be automatically updated without any help from the user. With the advent of natural language processing (NLP) platforms, cloud deployment, speech recognition and text-to-speech technologies, chatbots are capable of simulating human conversations which enhance the user experience.

The framework is developed keeping in mind that the users might have less or no prior coding experience. It has a simple drag and drop interface. First, the users have to enter the bot name and bot description. Later, the users are provided with three options namely NLP with wit.ai, NLP with api.ai and none. The database link has to be provided. Finally, the bot will be deployed on the cloud.

This document will explain the functional specifications for console web application and chatbot framework dashboard in detail.

References

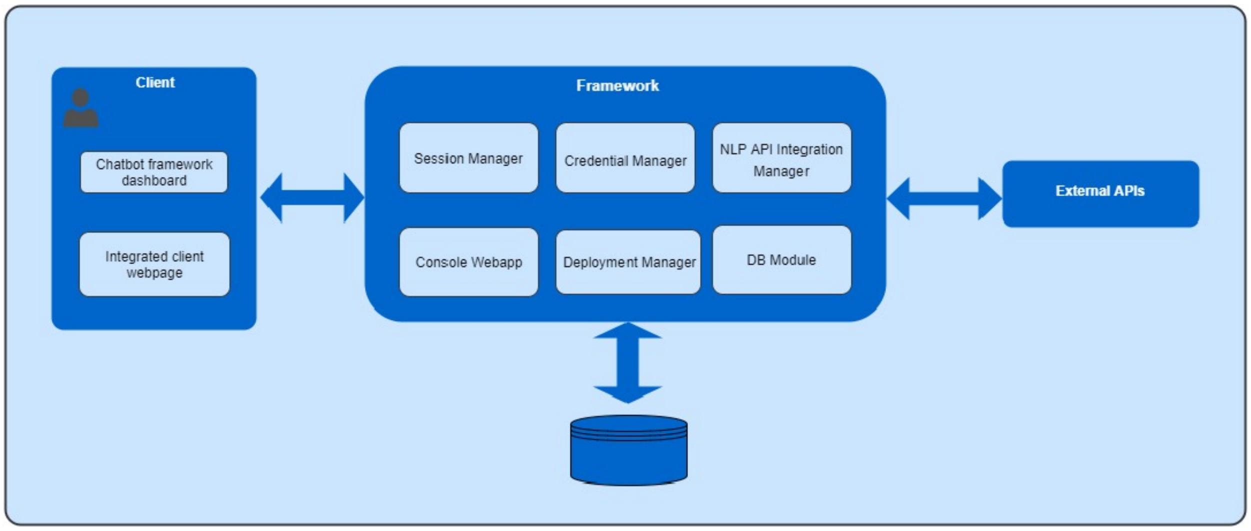
1. IBM Developer works – https://developer.ibm.com/code/topics/chatbot/
2. Dashbot – https://www.dashbot.io/
3. Microsoft bot framework – https://docs.microsoft.com/en-us/bot-framework/
4. Mind browser – http://mindbowser.com/chatbot-market-survey-2017/
5. Using chatbots to make dashboards – https://www.linkedin.com/pulse/using-chatbots-make-dashboards-more-powerful-dan-pillay/
6. Workbook 2 – Botman

Requirements

1. The system should be able to create, maintain and delete chatbots.
2. The system should allow the user to chat.
3. The system should allow the user to search any information.
4. The chatbot should be able to respond to any input it receives from the user.
5. The chatbot should be able to use text-to-speech technology to respond to the user.
6. If the chatbot understands the user input, it should respond with the correct information.
7. If the chatbot does not understand the user input, it should ask the user for more information.
8. The chatbot should be able to query the data from the API.
9. The chatbot should be able to simulate human conversations to enhance user experience.
10. The chatbot should be able to successfully integrate with NLP products like api.ai and wit.ai.
11. The dashboard data should be accurate.
12. The dashboard should be able to compute
13. The dashboard should be able to display visual data if available and required by the user.
14. The dashboard should automatically update data without any help from the user.

Functional Overview

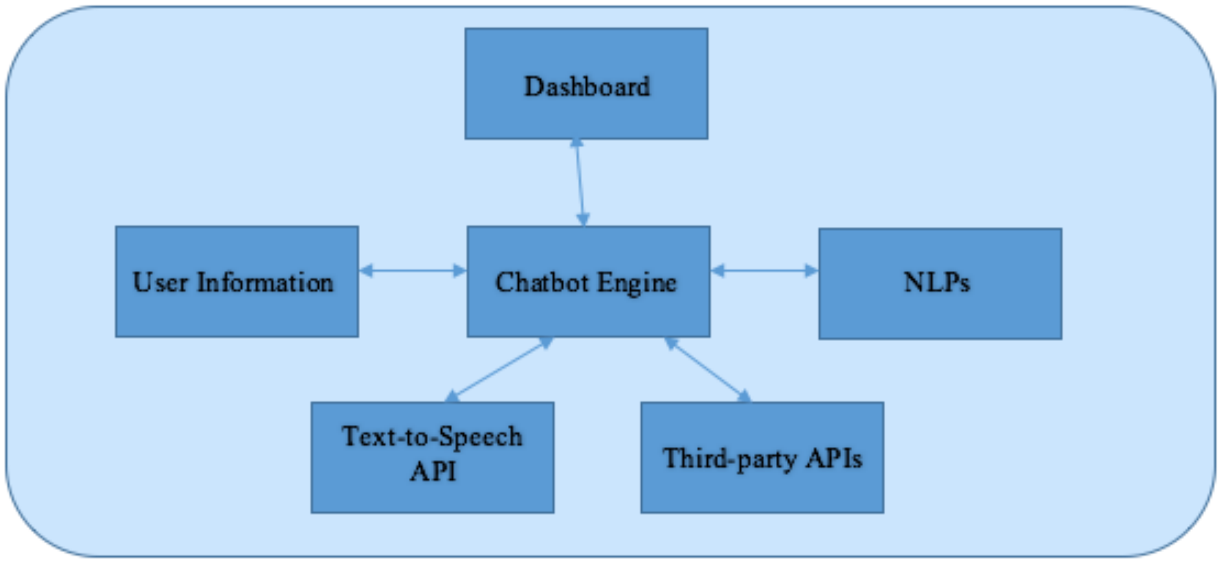
A high-level architecture diagram of the framework is given below:



The architecture diagram contains a client module, which further consists of chatbot framework dashboard and client webpage. Both of them are user facing and direct interaction point with the users. This module also interacts with the framework on the other end.

The framework contains components like session manager, credential manager, NLP integration manager, console web application, deployment manager and database module. All these components enable the system to process user inputs.

The external APIs allow integration of any third-party API. The developer can rely on this to formulate the chatbot's response to the user inputs. Lastly, the database stores user information, login credentials, number of chatbots created and cloud account credentials.



1. Dashboard
   The dashboard will provide visual display of the important and relevant user data. It is updated with the latest and accurate data at any given point.

2. Chatbot Engine
   The chatbot should be quick in responding to the user's questions. It communicates with the other components in order to do this.

3. User Information
   The chatbot requires continuous learning and improvement. There is an option of storing the user information with the chatbot to understand the user requirement in more detail.

4. Text-to-Speech API
   This will help in simulating a human conversation so that the user feels that it is interacting with another human and not an application.

5. Third-party APIs
   There is an option for integration of any third-party API. The chatbot's response can be easily formulated using this option.

6. NLPs
   The chatbot is integrated with NLP products like api.ai and wit.ai. The users are given the facility to select one of them.

Configuration

1. Set up login feature for the user.
2. Set up the user key and authentication token.
3. Set up the required bot files.
4. Set up the options for bot creation and deletion
5. Set up bot maintenance facility.

External Interfaces

1. Integration support from NLP products like wit.ai and api.ai.
2. Integration support from text-to-speech technologies.
3. Integration support from third-party APIs.

Dependencies

1. Availability of framework.
2. Accessibility of user input.
3. Availability of required content.
4. Repurposing the available required content.

Debug

The easiest way of debugging chatbots is by using Bot Framework Emulator. It is an application that enables the developers to test and debug chatbots either locally or remotely. The developer sends messages to the chatbot and inspects the responses given by the chatbot. The emulator is capable of displaying the messages exactly how they will appear in the web chat UI.

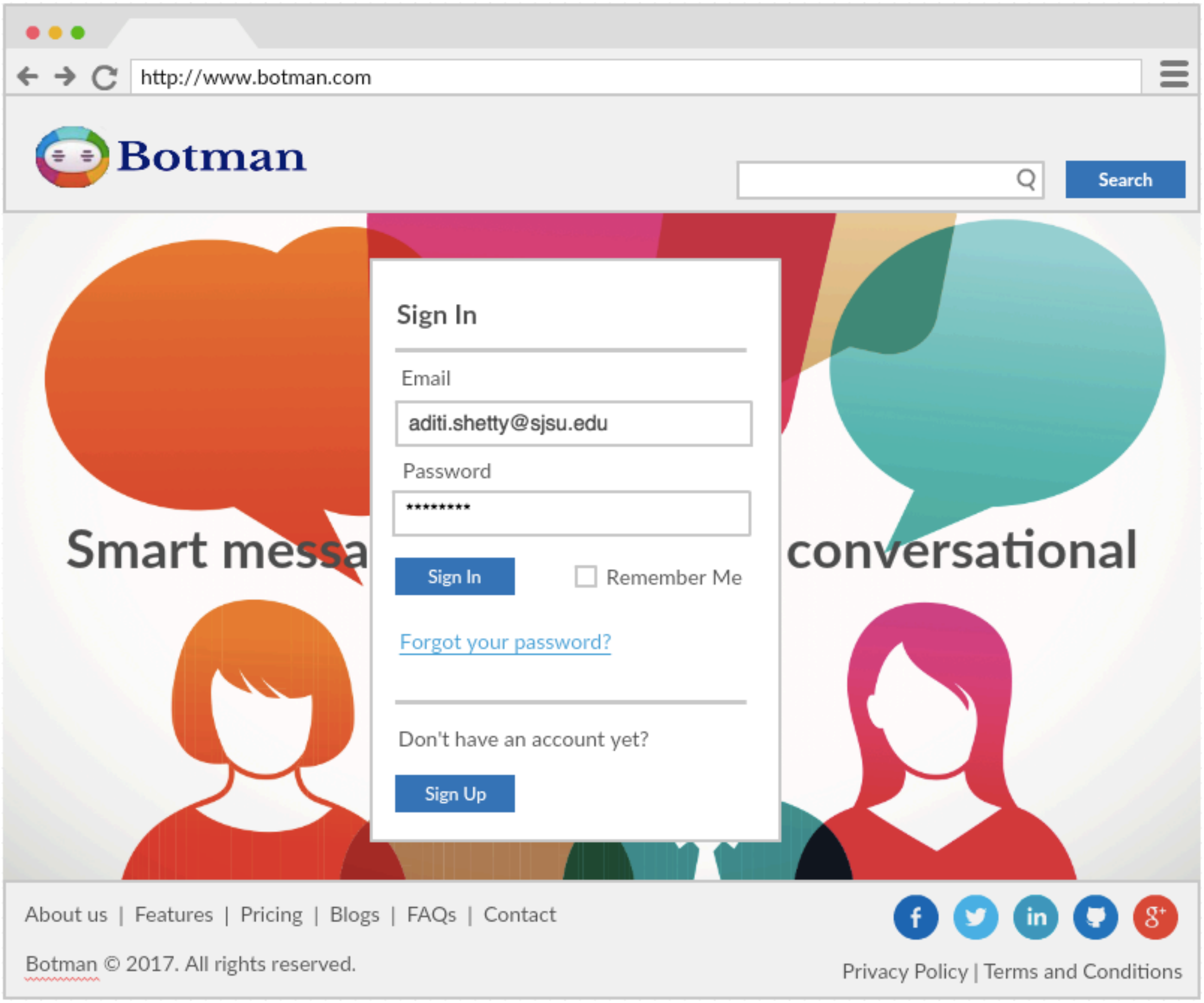
Before the bot is deployed on the cloud, it is run locally and tested using the emulator. Once the chatbot performs satisfactorily, it can be deployed on the cloud.

Logging

Logs are excellent tools for testing and debugging. The emulator logs JSON requests and responses when the developer chats with the bot. The logs are retained and maintained for testing purposes. It is extremely useful to diagnose problems and figure out which errors are happening. Logs provide more information which helps the developer to fix them and gain certain insights on what is going on in the chatbot.

Implementation

1. Log in using the user credentials



http://www.botman.com

Botman

Search

Smart messa

conversational

Sign In

Email

aditi.shetty@sjsu.edu

Password

****

Sign In

Remember Me

Forgot your password?

Don't have an account yet?

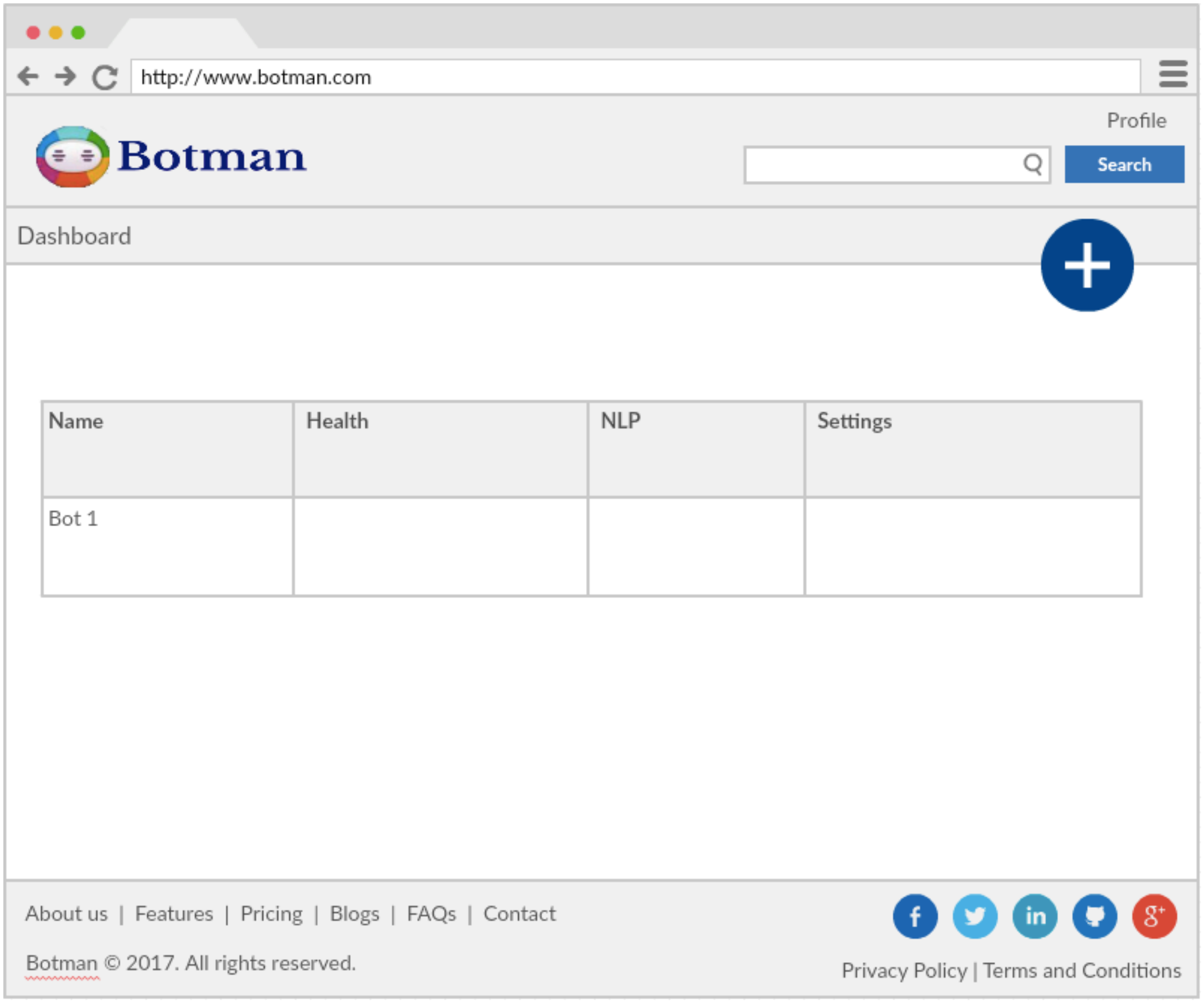
Sign Up

About us | Features | Pricing | Blogs | FAQs | Contact

Botman © 2017. All rights reserved.

Privacy Policy | Terms and Conditions

2. Select the option of creating a bot from the dashboard.

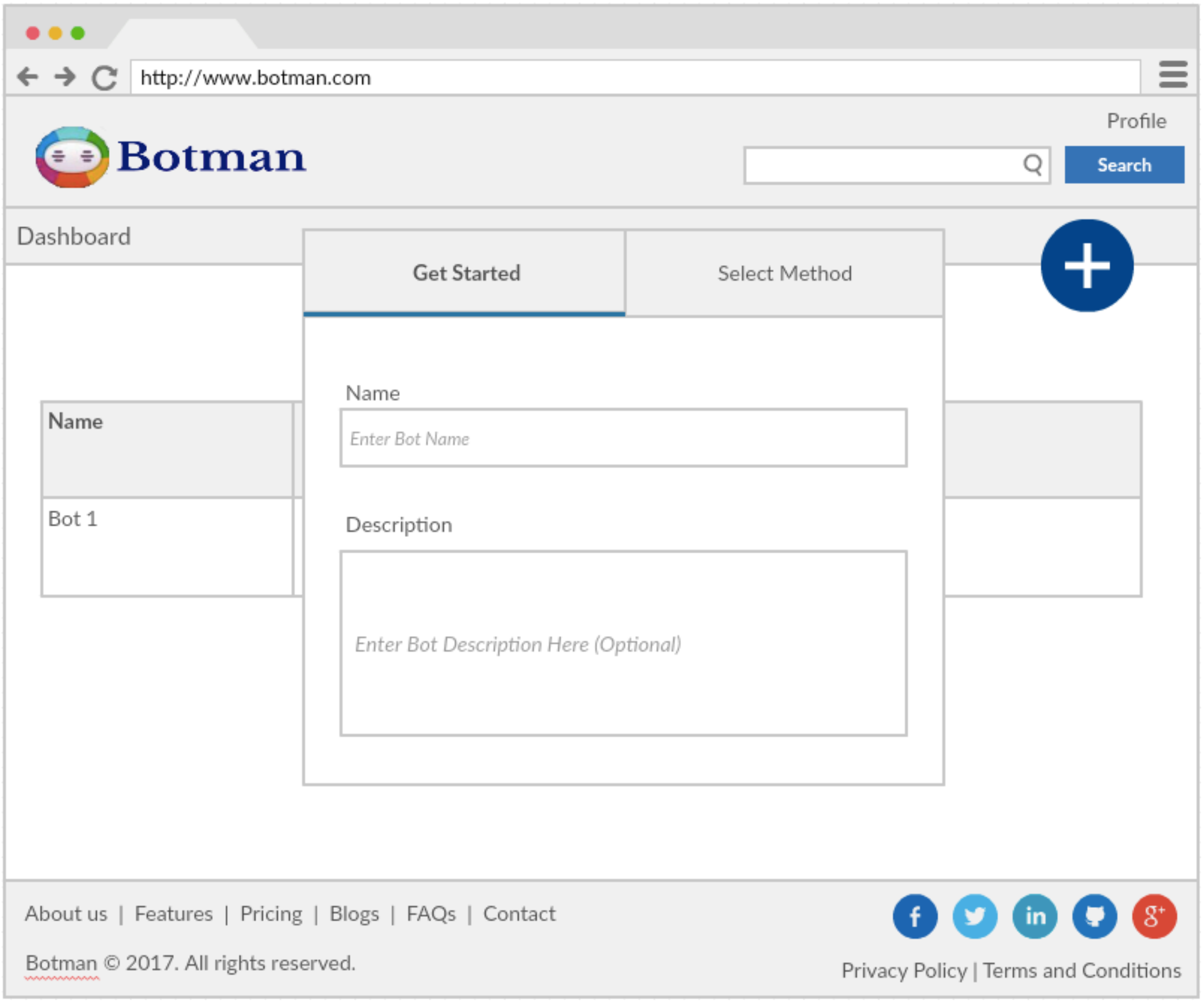


http://www.botman.com
Profile
Botman
Search
Dashboard
+

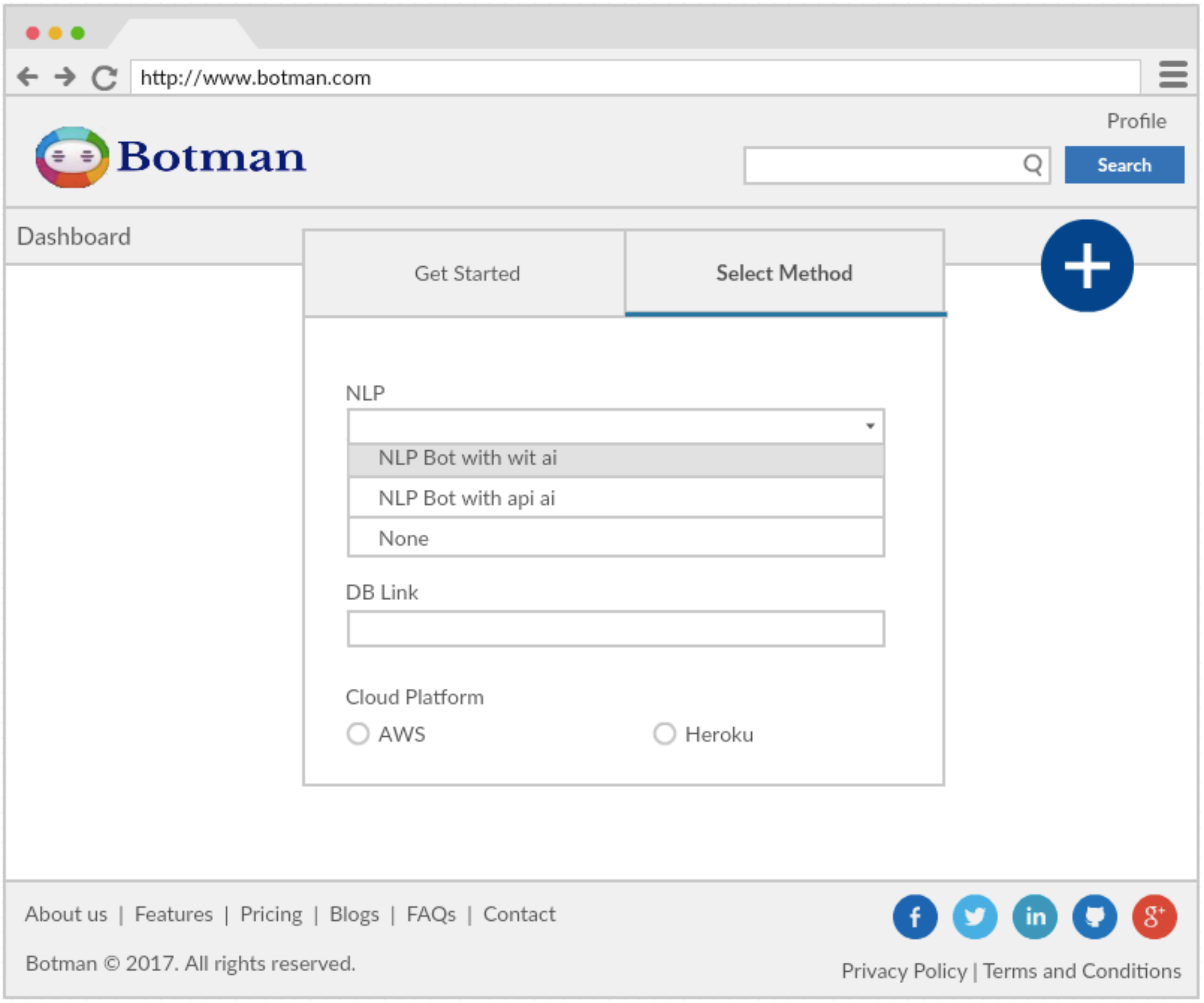
Name	Health	NLP	Settings
Bot 1			

About us | Features | Pricing | Blogs | FAQs | Contact
Botman © 2017. All rights reserved.
Privacy Policy | Terms and Conditions

3. Enter the bot name and its description.


http://www.botman.com
Profile
Botman
Search
Dashboard
Get Started
Select Method
Name
Enter Bot Name
Description
Enter Bot Description Here (Optional)
Name
Bot 1
About us | Features | Pricing | Blogs | FAQs | Contact
Botman © 2017. All rights reserved.
Privacy Policy | Terms and Conditions

4. Select NLP Bot with wit.ai or api.ai or none.



http://www.botman.com

Botman

Profile

Search

Dashboard

Get Started | Select Method

NLP

NLP Bot with wit ai

NLP Bot with api ai

None

DB Link

Cloud Platform

AWS

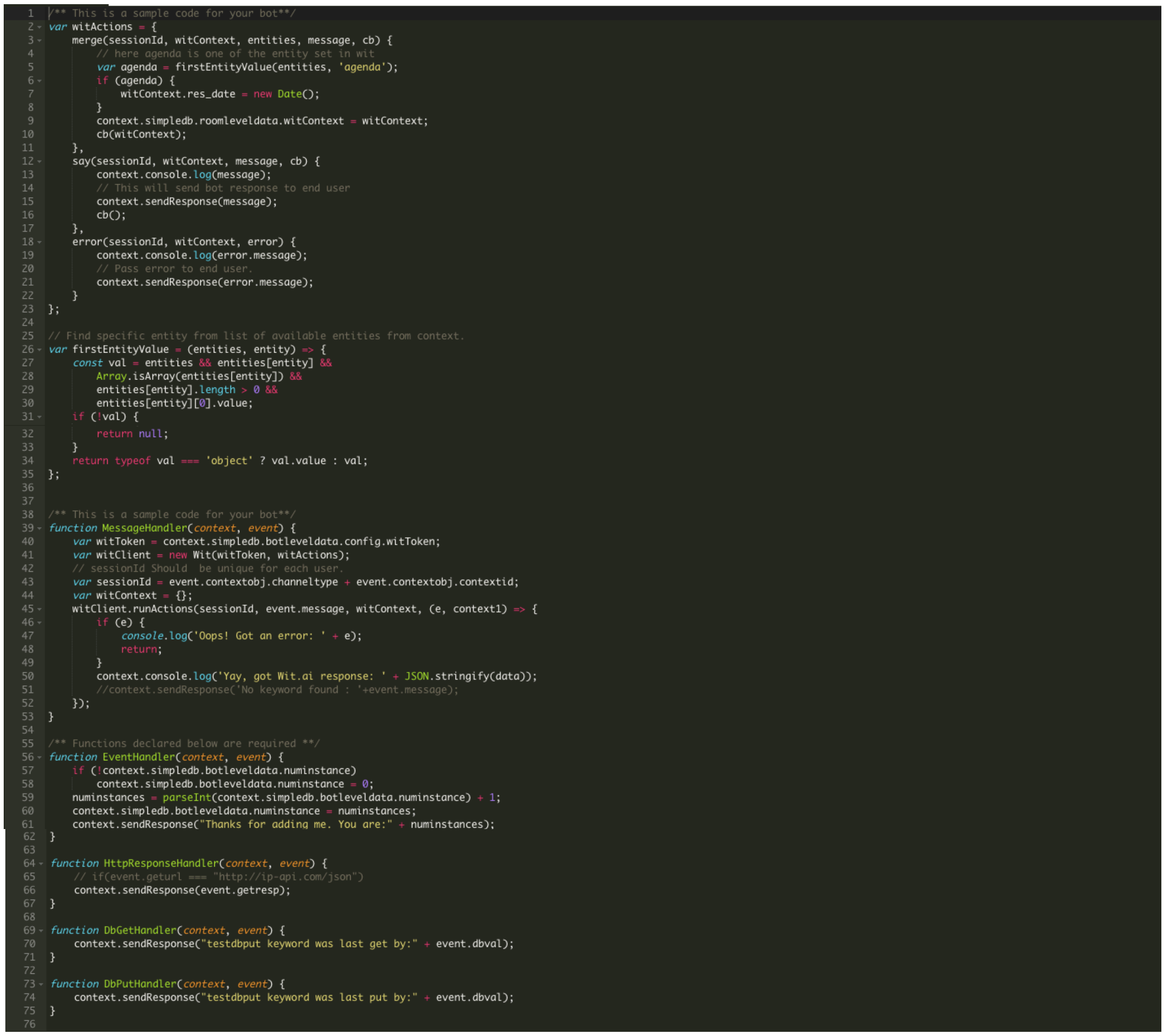
Heroku

About us | Features | Pricing | Blogs | FAQs | Contact

Botman © 2017. All rights reserved.

Privacy Policy | Terms and Conditions

5. Let’s assume the user selected the option NLP bot with ait.ai



```javascript
/** This is a sample code for your bot**/
var witActions = {
    merge(sessionId, witContext, entities, message, cb) {
        // here agenda is one of the entity set in wit
        var agenda = firstEntityValue(entities, 'agenda');
        if (agenda) {
            witContext.res_date = new Date();
        }
        context.simpledb.roomleveldata.witContext = witContext;
        cb(witContext);
    },
    say(sessionId, witContext, message, cb) {
        context.console.log(message);
        // This will send bot response to end user
        context.sendResponse(message);
        cb();
    },
    error(sessionId, witContext, error) {
        context.console.log(error.message);
        // Pass error to end user.
        context.sendResponse(error.message);
    }
};

// Find specific entity from list of available entities from context.
var firstEntityValue = (entities, entity) => {
    const val = entities && entities[entity] &&
        Array.isArray(entities[entity]) &&
        entities[entity].length > 0 &&
        entities[entity][0].value;
    if (!val) {
        return null;
    }
    return typeof val === 'object' ? val.value : val;
};


/** This is a sample code for your bot**/
function MessageHandler(context, event) {
    var witToken = context.simpledb.botleveldata.config.witToken;
    var witClient = new Wit(witToken, witActions);
    // sessionId Should  be unique for each user.
    var sessionId = event.contextobj.channeltype + event.contextobj.contextid;
    var witContext = {};
    witClient.runActions(sessionId, event.message, witContext, (e, context1) => {
        if (e) {
            console.log('Oops! Got an error: ' + e);
            return;
        }
        context.console.log('Yay, got Wit.ai response: ' + JSON.stringify(data));
        //context.sendResponse('No keyword found : '+event.message);
    });
}

/** Functions declared below are required **/
function EventHandler(context, event) {
    if (!context.simpledb.botleveldata.numinstance)
        context.simpledb.botleveldata.numinstance = 0;
    numinstances = parseInt(context.simpledb.botleveldata.numinstance) + 1;
    context.simpledb.botleveldata.numinstance = numinstances;
    context.sendResponse("Thanks for adding me. You are:" + numinstances);
}

function HttpResponseHandler(context, event) {
    // if(event.geturl === "http://ip-api.com/json")
    context.sendResponse(event.getresp);
}

function DbGetHandler(context, event) {
    context.sendResponse("testdbput keyword was last get by:" + event.dbval);
}

function DbPutHandler(context, event) {
    context.sendResponse("testdbput keyword was last put by:" + event.dbval);
}

```

6. The bot is ready for deployment.

Code Samples:

Code snippet for a basic chatbot

```cpp
#include <iostream>
#include <string>
#include <ctime>

int main()
{
    std::string Response[] = {
        "I HEARD YOU!",
        "SO, YOU ARE TALKING TO ME.",
        "CONTINUE, I'M LISTENING.",
        "VERY INTERESTING CONVERSATION.",
        "TELL ME MORE..."
    };

    srand((unsigned) time(NULL));

    std::string sInput = "";
    std::string sResponse = "";

    while(1) {
        std::cout << ">";
        std::getline(std::cin, sInput);
        int nSelection = rand() % 5;
        sResponse = Response[nSelection];
        std::cout << sResponse << std::endl;
    }

    return 0;
}
```

Testing

General Approach

Chatbots work in a different manner as compared to traditional software. The chatbot needs to understand the user's conversation and its context. The following items to be tested for a chatbot:

1. Chatbot speed.
2. Chatbot accuracy of responses.

3. Conversation stages.
4. Conversational flow.
5. Conversational logic
6. NLP score.
7. Chatbot response time.
8. Ease of usage.
9. Usability.
10. User experience.

Generally, chatbot testing can be done by asking basic questions to confirm that it performs properly. This testing is done to check if it functions properly when it receives expected inputs. After that, continuous testing needs to be done to find as many unexpected or incorrect inputs as possible.

1. Test if the dashboard displays accurate user information.
2. Test if the dashboard automatically updates user information.
3. Test if the chatbot is clear with the onboarding.
4. Test if the chatbot can provide the correct output for the expected input.
5. Test if the chatbot can understand idioms, slangs and emoji.
6. Test if the chatbot can handle a user request when it does not have the required output.
7. Test if the chatbot's responses are relevant and accurate.
8. Test if the chatbot can successfully mimic a human interaction.
9. Test if the chatbot can recover from an error.

Unit Tests

Test Case Number	Test Description	**E**xpected Result
1.	User logs in with correct credentials.	Log in is successful.
2.	User creates a bot.	Bot is created.
3.	User searched information.	Required information is provided.
4.	User deletes a bot.	Bot is deleted.
5.	A question is asked with the expected input.	Correct Answer is received.
6.	A question is asked with a spelling mistake.	Correct Answer is received.
7.	A question is asked with slangs, idioms and emoji.	Correct Answer is received.
8.	A question is asked which requires additional information.	Required information is asked.

Appendix

1. IBM Developer works – https://developer.ibm.com/code/topics/chatbot/
2. Debug bots emulator – https://docs.microsoft.com/en-us/bot-framework/debug-bots-emulator
3. Microsoft bot framework – https://docs.microsoft.com/en-us/bot-framework/
4. Bot framework – the https://dev.botframework.com/