**Introduction to IDE Bot Builder**

Gupshup.io has a tool called the Bot Builder that allows you to create bots in a jiffy. The Bot Builder includes a simple code editor, a publishing mechanism and a diagnostics program amongst other features, that significantly simplifies the process of building a chatbot.

#### Product Features

Usually, the process to build a bot would include setting up a dev. environment, installing libraries & packages, and setting up your own server space, to name a few. Gupshup has taken care of all these time-consuming processes to allow you to focus on building the bot logic. There are a few features of the IDE Bot Builder that enable this. First is the pre-installed libraries such as a Javascript async library and node-wit that is packaged with the code. Upon creation of a new bot you will also see some template code for common processes. These include helper methods that are briefly described below and in more detail in the following guides.

The IDE Bot Builder also provides single-click secure server deployment for your chatbot eliminating the need for your own server. The IDE Bot Builder is built on top of Amazon AWS Lambda and provides automated hosting. Deployment enables you to test out your chatbot using the Gupshup Proxy Bot. You can read a detailed guide about the Proxy Bot [here](https://www.gupshup.io/developer/docs/bot-platform/guide/testing-using-the-proxy-bot). You can also test out the conversational aspects of your chatbot using the built-in emulator given in the IDE Bot Builder.

#### Benefits

There are multiple benefits to using the IDE Bot Builder tool. It is more efficient and time-saving since a lot of the grunt-work involved in creating a functioning chatbot is automated. The IDE Bot Builder also reduces development time since it gives you the option of instant testing and deployment. You can also deploy to various messaging apps using a single click, thereby reducing the need to write separate API calls for each messaging app's integration. You can also integrate with third party services such as an NLP tool, through the IDE Bot Builder. As an advanced user, you can have control for custom hosting of your bot using the 'Callback URL' option to specify the endpoint where your bot logic resides.

#### Methods provided by the IDE Bot Builder

There are five methods given to a user and they are all required functions. We'll briefly discuss what each method does.

* MessageHandler(): This is the method that executes when your bot is communicated with. It has two parameters: 'context' and 'event'. You can parse what message was sent to your bot using 'event.message'. Your bot can then reply using the context.sendResponse property.
* EventHandler(): This function is invoked when an event associated with a bot is triggered. Examples of events that are triggered are: someone joining a Facebook group or someone invoking a Proxy Bot. When you type in 'proxy {botName}' to map your bot to the Proxy Bot, the EventHandler function is invoked.
* httpResponseHandler(): You can use http GET and POST requests in your chatbot. The responses to any such http call will be handled by the httpResponseHandler function.
* DbGetHandler(): A method for database persistence. Use this method to query your database for any records
* DbPutHandler(): A method for database persistence. Use this method to insert records into your database.

The next few guides will illustrate the process of building chatbots using the IDE Bot Builder.

# Different objects sent to the bot

Everytime a message is sent from the user to the bot, these objects are sent in the JSON format to the ***event*** parameter of handler methods inside the IDE Bot Builder.

|  |  |  |  |
| --- | --- | --- | --- |
| **Object Name** | **Sample Value** | **Description** | **Remarks** |
| contextobj | {"botname":"demobot", "channeltype":"twitter", "contextid":"john", "contexttype":"p2p"} | Json object defining various parameter and way of conversation. | **channeltype** - the messaging channel name from where the user interacts with the bot. For list of channels refer [appendix](https://www.gupshup.io/developer/docs/bot-platform/guide/appendix).  **contextid / channelid** - The user id of a user on the particular messaging channel. **contexttype** - The type of conversation. For the list of probable values refer [appendix](https://www.gupshup.io/developer/docs/bot-platform/guide/appendix). **display** - The display name of a user corresponding to the user id on the messaging channel **botname** - The bot name. **text** - The message from the user. **type** - The type of message sent by the user. For the list of probable values refer [appendix](https://www.gupshup.io/developer/docs/bot-platform/guide/appendix). |
| senderobj | {"channelid":"john", "channeltype":"twitter", "display":"John Doe"} | JSON object identifying the user details. This is useful in case of a group on a messaging channel. |
| messageobj | {"text":"Hi", "type":"msg"} | JSON object for messages from the user. |
| channel | twitter | The channel from where the user is interacting with the bot. |  |
| botname | demobot | The name of the bot. |  |
| sender | 195173 | This is the user id of the user on the messaging channel who has started the conversation with the bot. |  |
| message | Hi | The message sent by the user to the bot. |  |

***These are the properties to access the objects in the IDE Bot Builder*** .

|  |  |
| --- | --- |
| **Property Name** | **Description** |
| event.message | Returns the message from the user to the bot. This can be text or image URL or location URL etc. |
| event.sender | Returns the user id of the user from the messaging channel. |
| event.channel | Returns the messaging channel name from where the user is conversing with the bot. |
| event.botname | Returns the name of the bot. |
| event.senderobj.channelid | Returns the user id of the user from the messaging channel. |
| event.senderobj.channeltype | Returns the messaging channel name from where the user is conversing with the bot. |
| event.senderobj.display | Returns the display name of the user on a messaging channel from where he/she is conversing with the bot |
| event.messageobj.text | Returns the message from the user to the bot. This can be text or image URL or location URL etc. |
| event.messageobj.type | Returns the message type from the user. It can be msg or image or location etc. |
| event.messageobj.latitude | Returns the latitude of the location sent by the user to the bot. |
| event.messageobj.longitude | Returns the longitude of the location sent by the user to the bot. |
| event.messageobj.address | Returns the address of the location sent by the user to the bot. |
| event.messageobj.url | Returns the image URL of the image sent by the user to the bot. |
| event.contextobj.botname | Returns the name of the bot. |
| event.contextobj.channeltype | Returns the messaging channel name from where the user is conversing with the bot. |
| event.contextobj.contextid | Returns the user id of the user from the messaging channel. |
| event.contextobj.contexttype | Returns the type of conversation. |
| event.getresp | Returns the response of an HTTP/HTTPs call into the HttpResponseHandler function. |
| event.dbval | Returns the response of the database call into the DbGetHandler or DbPutHandler function. |
| event.params | Returns the URL parameters sent to the bot through [HTTP call to the bot](https://www.gupshup.io/developer/docs/bot-platform/guide/http-call-to-a-bot). Example - if one of the parameter is "message" then to access it use 'event.params.message' in the HttpEndpointHandler function. |

***Sample JSON for text message*** -

{

"botname": "demobot",

"contextobj": {

"botname": "demobot",

"channeltype": "fb",

"contextid": "1951021",

"contexttype": "p2p"

},

"messageobj": {

"text": "Hi",

"type": "msg"

},

"senderobj": {

"channelid": "1951021",

"channeltype": "fb",

"display": "John"

},

"channel": "fb",

"sender": "1951021",

"message": "Hi"

}

***Sample JSON for location from the user***

{

"botname": "demobot",

"contextobj": {

"botname": "demobot",

"channeltype": "telegram",

"contextid": "195173611",

"contexttype": "p2p"

},

"messageobj": {

"address": "",

"latitude": "19.145313",

"longitude": "72.855748",

"text": "https://www.google.com/maps/?q=19.145313,72.855748",

"type": "location"

},

"senderobj": {

"channelid": "195173611",

"channeltype": "telegram",

"display": "Shreyans"

},

"channel": "telegram",

"sender": "195173611",

"message": "https://www.google.com/maps/?q=19.145313,72.855748"

}

***Sample JSON for image from the user***

{

"botname": "rildemo",

"contextobj": {

"botname": "demobot2",

"channeltype": "telegram",

"contextid": "195173611",

"contexttype": "p2p"

},

"messageobj": {

"imgData": "{\"botname\":\"rildemo\",\"contextObj\":{\"botname\":\"demobot2\",\"channeltype\":\"telegram\",\"contextid\":\"195173611\",\"contexttype\":\"p2p\"},\"url\":\"https://api.telegram.org/file/bot183230543:AAHr9nQwmXVsGZzS8zk6gd-bd0pBC\_kl5N4/photo/file\_119.jpg\"}",

"text": "https://api.telegram.org/file/bot183230543:AAHr9nQwmXVsGZzS8zk6gd-bd0pBC\_kl5N4/photo/file\_119.jpg",

"type": "image",

"url": "https://api.telegram.org/file/bot183230543:AAHr9nQwmXVsGZzS8zk6gd-bd0pBC\_kl5N4/photo/file\_119.jpg"

},

"senderobj": {

"channelid": "195173611",

"channeltype": "telegram",

"display": "Shreyans"

},

"channel": "telegram",

"sender": "195173611",

"message": "https://api.telegram.org/file/bot183230543:AAHr9nQwmXVsGZzS8zk6gd-bd0pBC\_kl5N4/photo/file\_119.jpg"

}

# A 'Hello World' bot

Creating your first chatbot on the Gupshup platform is really simple. This guide will show you the steps to create a cross-platform bot in under two minutes. We will use the IDE Bot Builder tool. Read the Introduction to IDE Bot Builder [here](https://www.gupshup.io/developer/docs/bot-platform/guide/intro-to-gupshup-bot-builder) before proceeding.

#### Overview:

Perform the following steps to create your first chatbot:

1.    Login to Gupshup using the Github or Facebook login.

2.    Use the IDE Bot Builder tool to create a bot.

3.    Say Hello World

4.    Add the Proxy Bot to your messaging app.

5.    Test out your bot.

#### Create a bot in under two minutes

1.    Go to gupshup.io and login using your Github or Facebook login.

2.    Click on the 'My Bots' tab and click on the plus button to create a new bot. Give it a unique name. Choose the IDE option and hit the 'Create Bot' button.

# Handling conversations

Once you've created your first bot using [Gupshup](https://www.gupshup.io/developer/docs/bot-platform/guide/build-a-simple-bot-in-2-minutes), you can move on to handling conversations. A chat conversation is the interface on which your bot operates so handling conversations with your user is vital.

#### Overview:

This article will cover:

1.    Responding to a basic conversation

2.    Getting a user's message and user's name

3.    Adding text processing

4.    Emulate, deploy and test.

#### Handling conversations

1.    Follow the [guide](https://www.gupshup.io/developer/docs/bot-platform/guide/a-hello-world-bot) to create a Hello World bot.

2.    Use the event.message parameter to get the user's message to the bot.

context.sendResponse(event.message);

3.    Once you have the user's message you can use conditional statements to process the user's input. Chose your replies based on what the user has messaged.

function MessageHandler(context, event) {

if(event.message == "hi") {

context.sendResponse("Hello!");

}

else if (event.message == "hello") {

context.sendResponse("How are you doing " + event.sender + " ?");

}

else {

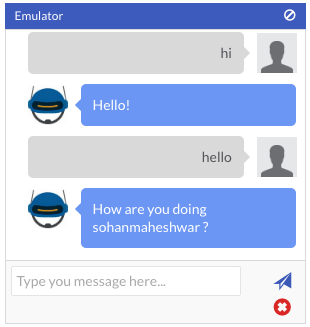
context.sendResponse('No keyword found : '+event.message);

}

}

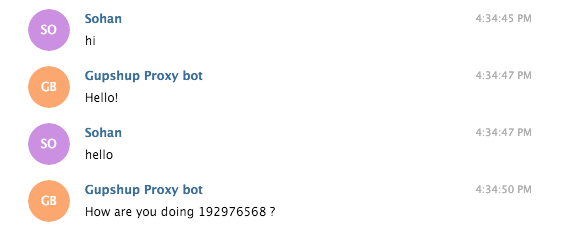
#### Testing your bot

1.    Try typing out “hi” and “hello” in the emulator provided.



2.    Hit the 'Deploy' button to deploy your bot. Like the previous guide, we're going to test this bot on Telegram by using the Proxy Bot.

3.    Type “hi” and “hello” and you'll see different responses.



That's the basic guide to handling conversations. Of course as the scope of the bot becomes more complex, the more advanced the conversation handling becomes. We'll touch upon some of this later on in the series.

# Using the Emulator, Console and Logs

The IDE Bot Builder tool takes care of the complete bot lifecycle. The bot lifecycle also includes local testing and debugging.

The IDE Bot Builder provides an option to test the bot locally before deploying it onto the server and it also provides an option to add specific logs to debug the code. Let's look at both these one by one to understand how they work and how they can be utilized in the bot building process.

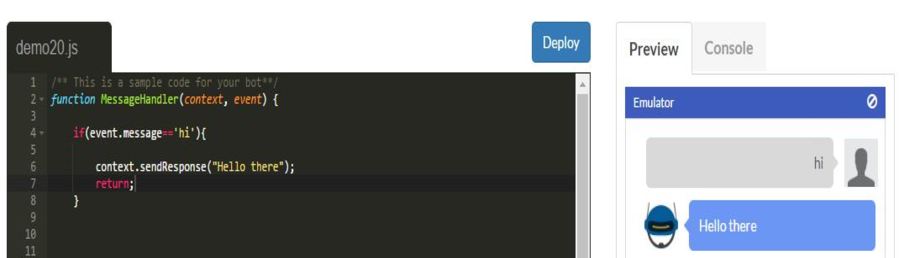
1. Emulator - Tool for local testing.
2. Console & Logs - Tools for logging and debugging.

#### Emulator

The emulator is a tool which is located just beside the hosted code under the 'Develop' tab. You can test your hosted code using the emulator.

**Testing Conversation** -

You can test out the conversation handling where you respond to user's conversation using the conditional statements. Let's see an example where we will respond to a 'hi' with 'hello there' and test it out in the emulator.



The **event** object in the case of emulator contains -

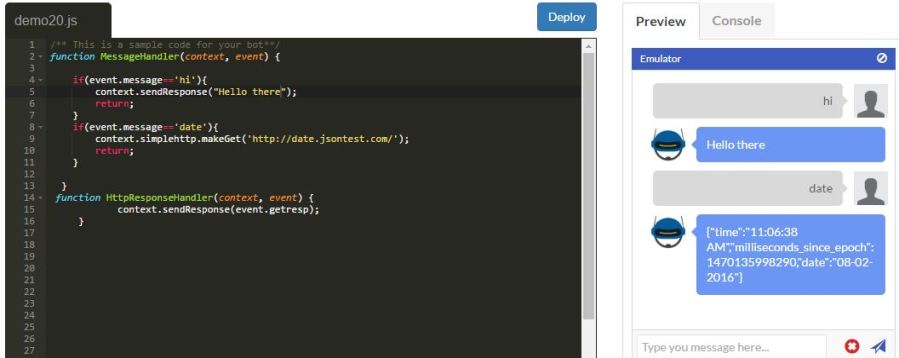
{"message":"hi","sender":"john"}

Since it's not a messaging channel,it's different from the objects sent while running the bot from a messaging channel.

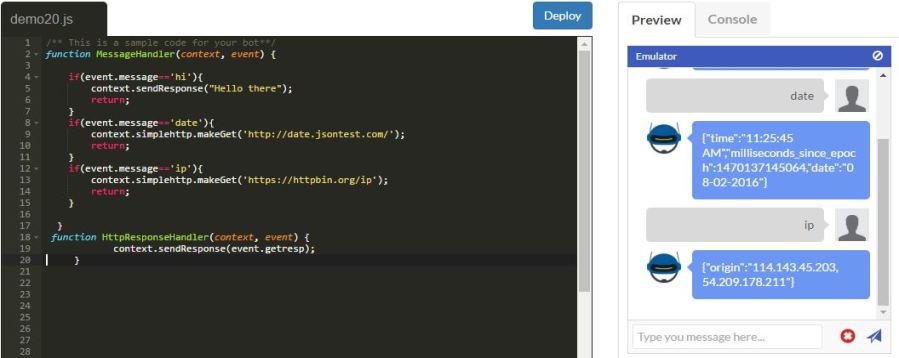
**Testing HTTP calls** -

In the above example, you saw how to use the emulator to test out the conversations. Let's look at testing HTTP/HTTPS calls.

First, we will look at making an HTTP GET call. The response of the HTTP call is sent to the respective handler using which the response can be handled.



Now let's look at making an HTTPS GET call. Same way as above the response will be handled by the HttpResponseHandler.

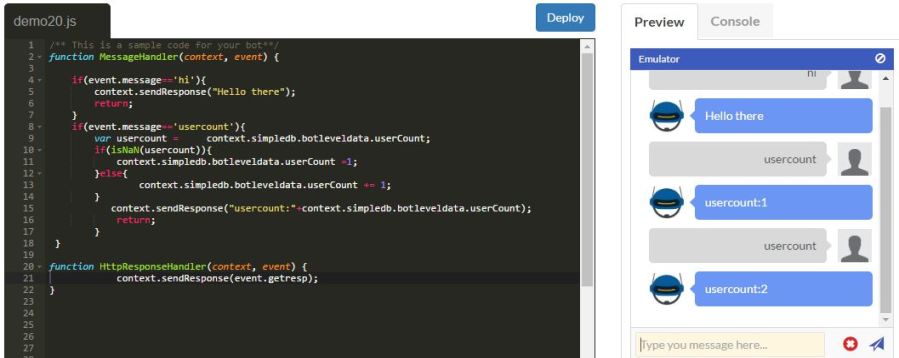


Please note -  
1. HTTPs GET calls may not work all the time in the emulator and only certain request may work. Hence test HTTPs calls using a messaging channel  
2. GET/POST calls with headers or parameters will not work in the emulator.Hence test using a messaging channel

**Testing basic data persistence** -

You can test both **botleveldata** and **roomleveldata** using the emulator.The emulator stores the data in its temporary memory.

To test how this works let's define a variable called 'usercount' which is incremented every time a user says usercount and stored at the **botleveldata**

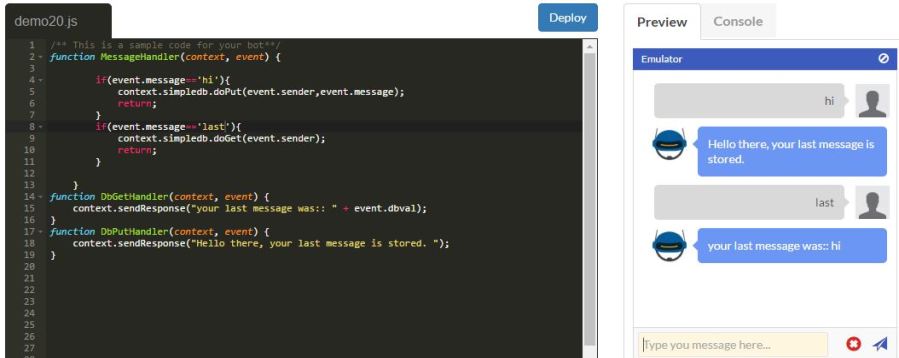


The **roomleveldata** can also be tested in the same way as the example above.

**Testing advanced data persistence** -

Now that you have seen how basic data persistence will work on the emulator , let's look at advance data persistence like **doPut** and **doGet**.

To test this we will store the message sent by the user using **doPut** and using the sender name as the key in and then when the user says 'last' we will retrieve the stored message using **doGet**.

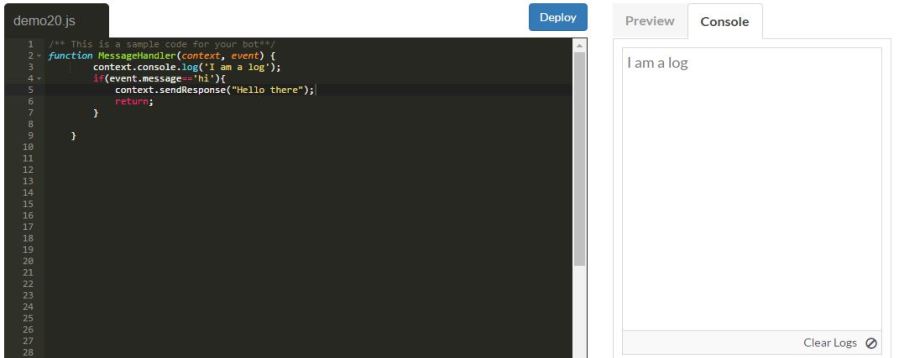


Please note that all the data stored by the emulator is temporary and valid only for the active instance of your testing.

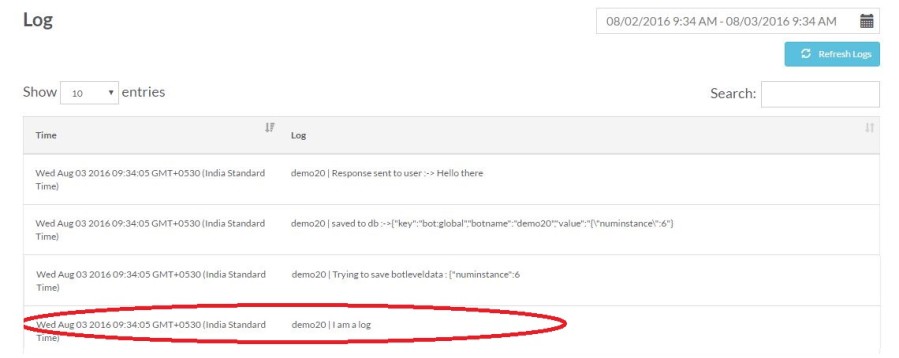
#### Console

The console is very useful. You can debug your code by adding some debug statement which will help you in debugging any functionality or error of your bot in the hosted code.

To log any statements you can use **context.console.log**. When you are testing in the emulator the log statements will appear in the Console tab next to the emulator.

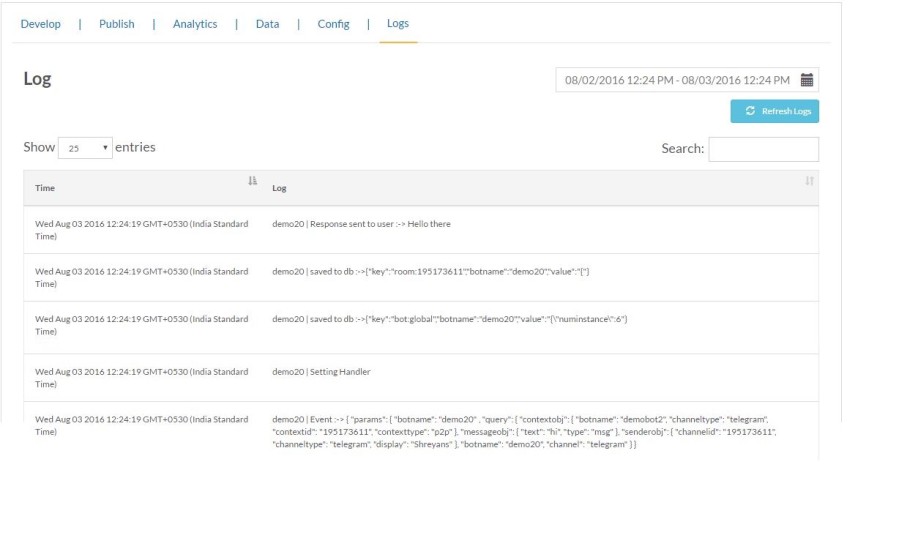
Example -   


If you are testing from a messaging channel then this log statement will appear in the 'Logs' tab of the bot builder.

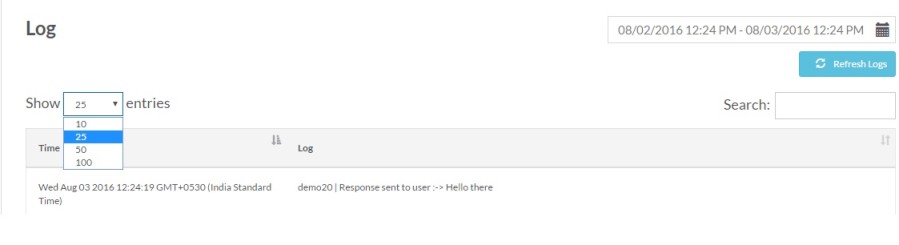


#### Logs

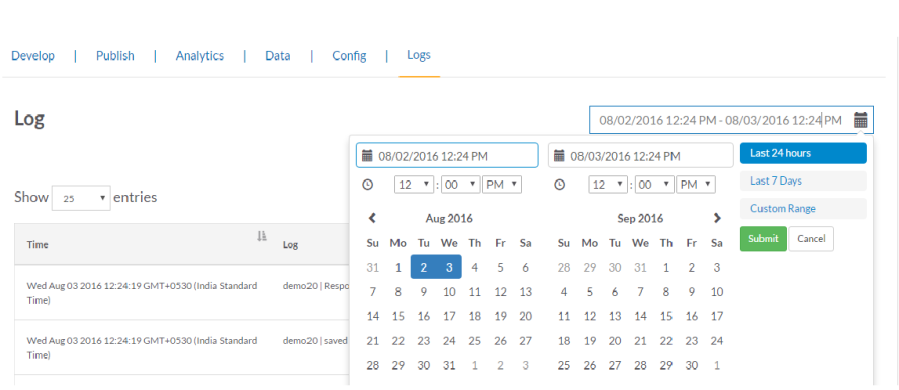
The 'Logs' tab contains the logs of your bot. These logs are very helpful in understanding the user and bot conversation as everything gets logged. Even DB operations or HTTP calls gets logged, thus making it easier for developers to find any flaw in the hosted code.   
These logs are only created when the bot is accessed from one of the supported messaging channels.



You can choose the number of results to display from the dropdown on the left-hand side.



You can also choose the duration between which you need the logs for by selecting the option on the right-hand side but only 500 logs can be displayed at a given point of time.



# Adding basic data persistence

Before you start reading this, please read the guide on [handling conversations](https://www.gupshup.io/developer/docs/bot-platform/guide/handling-conversations).

Now that you have understood how to handle conversations let's take bot building to the next level by adding basic data persistence. You can enhance the behavior of your bot using data persistence like you can store the user preference and then tune your bot accordingly or you can add a configuration file for your bot hiding away important information from the code. There are two levels of basic persistence offered - **botleveldata** and **roomleveldata**. All the data is stored as string in the database.

#### What is botleveldata and how to use it?

Botleveldata is the global data for a particular bot for all instances of its usage across all messaging channels and users. This data is accessible to the bot for all the conversations with all the users from the different messaging platforms.

**For example** -

Say you want to capture the total count of usage of your bot. You can assign a variable to the botleveldata and then increment it every time a new user access it. This data will be updated in the database and can be used by the bot in any chat.

Now let's see how to achieve this -   
Create a variable 'usagecount' which will then be incremented every time the bot is accessed by a new user and assign the variable to the botleveldata.

if (firstTimeUser)

{

context.simpledb.botleveldata.usagecount = 0;

totalcount= parseInt(context.simpledb.botleveldata.usagecount) + 1;

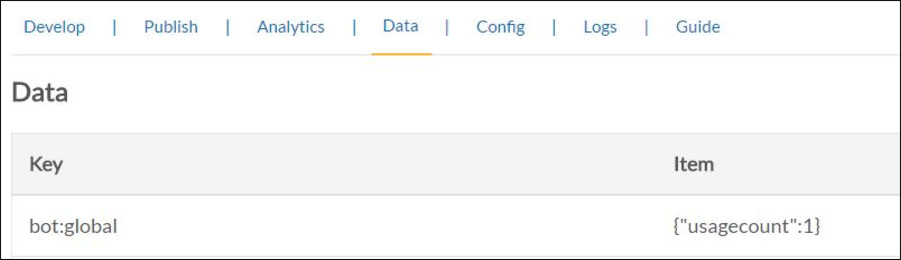
context.simpledb.botleveldata.usagecount = totalcount;

context.sendResponse("You are user number" +context.simpledb.botleveldata.usagecount);

}

For botleveldata to be stored the last statement should always be context.sendResponse.

This is how it looks in the data base



#### What is roomleveldata and how to use it?

Roomleveldata corresponds to the user level data. This data belongs to one single conversation between a bot and a user from one particular messaging channel. If the same user starts the conversation with the bot from a different messaging channel then that creates a separate roomleveldata.

**For example** -

Say you want to keep adding the numbers which the user provides to your bot and provide him with the total when he asks for it. You can achieve that by storing the number into roomleveldata and then keep adding the new number to the stored number and so on. Then pull out the latest number when the user asks for the total.

Now let's see how to achieve this -

Check if the message is a number or not then check for a variable 'totalcount', if it's not found then create it by assigning it to roomleveldata else just add the new number to the old number.

if (!isNaN(event.message))

{

var totalvalue = context.simpledb.roomleveldata.totalcount;

if (!totalvalue)

context.simpledb.roomleveldata.totalcount= parseInt(event.message,10);

else

context.simpledb.roomleveldata.totalcount = totalvalue + parseInt(event.message,10);

context.sendResponse("");

}

else if (event.message.toLowerCase() == 'total')

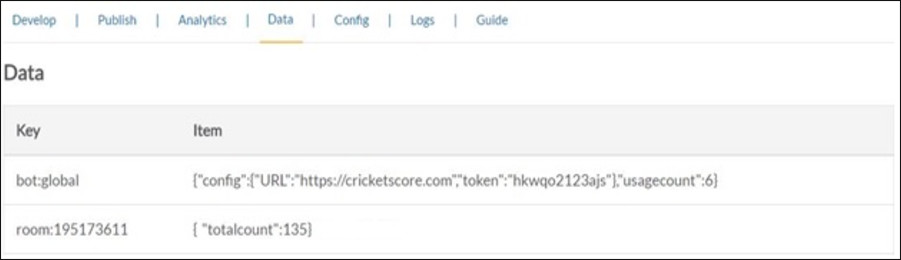
{

context.sendResponse(context.simpledb.roomleveldata.totalcount)

}

For roomleveldata to be stored with the variable the last statement should always be context.sendResponse.

This is how it looks in the database -



Let's look at another example for the use of roomleveldata -

Say you have a pizza ordering bot and want to store the user selection of pizza. Hence you can do that by setting up a variable 'usual' and setting the selection as value to the variable. When a user asks for the 'usual' then you can display it and order the same pizza.

How do you achieve this -

Say user selected combo #3 for pizza which is a medium size pizza with veggie toppings and a diet coke. You can ask him about making this his default choice and if he says 'yes' store combo #3 for the variable 'usual' under roomleveldata.

if (event.message == "yes")

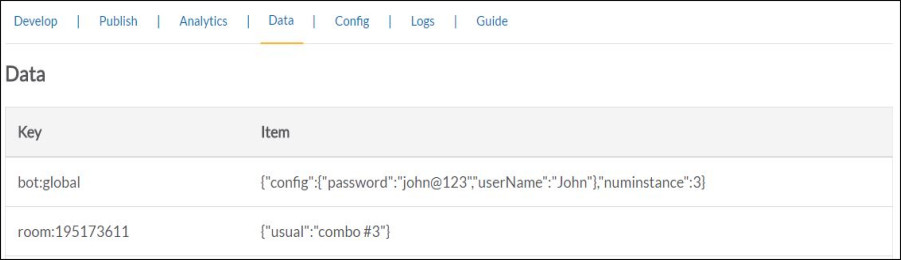
{

context.simpledb.roomleveldata.usual='combo #3';

context.sendResponse("I have stored your preference");

}

This is how it looks in the database -



So next time when the user says 'usual', you just have to get the value inside usual and proceed.

if (event.message == "usual")

{

var order = context.simpledb.roomleveldata.usual;

//placing an order code goes here

context.sendResponse("I have placed your order");

}

To delete any of this data, just set the variable's value to null like

context.simpledb.roomleveldata.usual ="";

or

context.simpledb.botleveldata.usagecount = "";

and this will update the variable to null and next time you can check for null for the same variable and add any value to it.

That is all for this tutorial. Do watch the related video for more understanding through live code example.

We will cover advanced persistence in our next tutorial where you will learn how you can store your own data structure in the database.

**Adding advanced data persistence**

In the previous guide we leaned about [basic data persistence](https://www.gupshup.io/developer/docs/bot-platform/guide/adding-basic-data-persistence), now let's understand how to add advanced persistence into your bot.

Say you have a certain data structure and you want to add that complete structure for the bot to access through the hosted code. Hence you can add the data structure into the hosted DB which is the NoSQL database provided by Amazon called DynamoDB. You can define a key and store the data corresponding to that key and to fetch the data you need to use the same key. There are two methods provided for this activity '***doGet***' and '***doPut***' and both these methods are non-blocking calls, hence the response of these methods are passed to their respective handlers in the code '***DbGetHandler***' and '***DbPutHandler***'. All the data is stored as a string.

Let's look at an example to understand how this will be helpful.

Say you have a team and everyone is using a JIRA(a bug filing tool) bot that shows the status of a bug, last modified data, the name of the person who modified it and the comment made if a bug ID is passed to it. This is pretty helpful for the team as they can just type in the JIRA bug ID and know the latest details on what is happening. To achieve this you need to store these 4 details of each bug every time it gets updated by someone.

* Last modified date.
* Person who modified.
* Comment made.
* Bug status

This can be achieved using the 'doPut' method by storing all these data in the form of JSON object and the bug ID as the key.

Let's look at the code:

function MessageHandler(context, event) {

if(event.message=='update bug - 1452') {

jiraUpdate(context);

}

}

function jiraUpdate(context){

//connect to Jira and check for latest update and values

if(true){

context.simpledb.doPut("1452" ,"{\"status\":\"QA pending\",\"lastUpdated\":\"06\/05\/2016\",\"userName\":\"John\",\"comment\":\"Dependent on builds team to provide right build\"}");

} else{

context.sendResponse('No new updates');

}

}

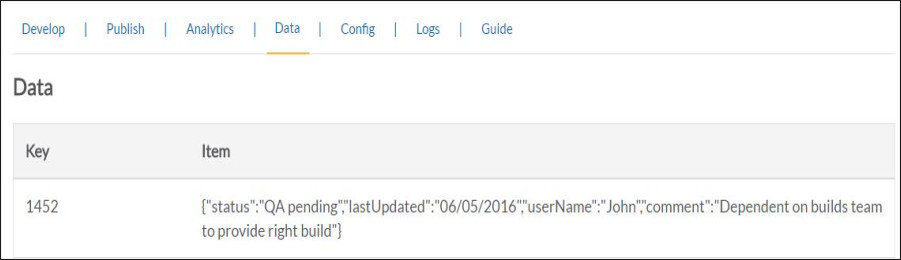
function DbPutHandler(context, event) {

context.sendResponse("New update in the bug, type in the bug ID to see the update");

}

In the code you can see that if someone is asking for the update on the bug ID 1452 then it's checking for update and then setting them in the data base.

This is how the data base entry looks like -



Now say the same user or any other user from another messaging channel enters the bug ID to see the update then use 'doGet' to get the values from the database and display it to the user.

Let's look at the code -

function MessageHandler(context, event) {

if(event.message=='1452'){

context.simpledb.doGet("1452");

}

}

function DbGetHandler(context, event) {

var bugObj = JSON.parse(event.dbval);

var user = bugObj.userName;

var lastupdate = bugObj.lastUpdated;

var status = bugObj.status;

var comment = bugObj.comment;

context.sendResponse(user+" "+lastupdate+" "+comment+" "+status);

}

In the code you can see that the data is fetched, parsed and displayed once a user provides the bug ID.

That is all for this tutorial. Do watch the related video for more understanding through a live code example.

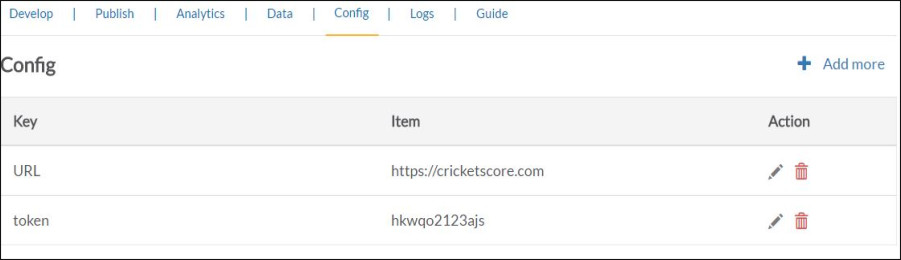
# Using Config and Data

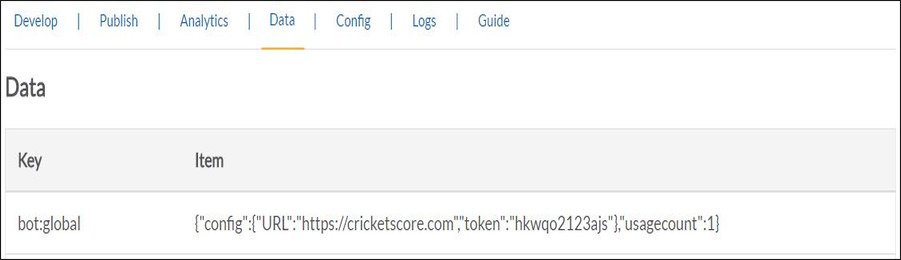
In this guide, we will understand the Config and Data tabs.

Let's say you are building a bot which provides cricket score and for this you need to make an API call to a specific URL and have a token to access the API. Now you can choose to store it in a variable inside your code. But what if tomorrow the URL or the token changes? You will have to make changes to the code at all the places where you have used them.

There is a better way to do this, you can store the URL and the token as configuration into the botleveldata and access it through the code.

Now let's see how to achieve this example:

First, use the '**Config**' tab from the IDE Bot Builder and set up URL and token into the configurations.   


This will reflect in the database like:   


Now to access this your code should look like:

var url = context.simpledb.botleveldata.config.URL;

var token = context.simpledb.botleveldata.config.token;

That is all for this tutorial. Do watch the related video for more understanding through live code example.

# Making HTTP calls from the bot

While developing the bot using IDE bot builder you may need to make HTTP calls in cases where your bot needs to fetch certain data from external servers or need to post certain data to external server e.g. - Your bot delivers news to the user or your bot needs to validate a user against your backend etc.

The response of any HTTP call is passed on to a particular response handler. The default handler is **HttpResponseHandler** but you can also define your own callback function.

Let's look at the two types of HTTP calls which you can make

1. [**GET**](https://www.gupshup.io/developer/docs/bot-platform/guide/making-http-calls#get)
2. [**POST**](https://www.gupshup.io/developer/docs/bot-platform/guide/making-http-calls#post)

## GET Call

**GET call example**: Here we are making a call to a date API and parsing the response in the HttpResponseHandler.

function MessageHandler(context, event) {

if(event.message== "date") {

context.simplehttp.makeGet('http://date.jsontest.com/');

}

}

function HttpResponseHandler(context, event) {

var dateJson = JSON.parse(event.getresp);

var date = dateJson.date;

context.sendResponse("Today's date is : "+date);

}

By default, the response is sent to 'HttpResponseHandler' function and then parsed.

**GET call with header example**: Here we are making a GET call to a weather API but it takes API key in the header.

function MessageHandler(context, event) {

if(event.message == "mumbai") {

context.simplehttp.makeGet('http://weatherapi.com/json/weather.json?city=mumbai', {"apikey":"ca99833c8c16a68d94"});

}

}

function HttpResponseHandler(context, event) {

var weatherJson = JSON.parse(event.getresp);

var temp = weatherJson.currentTemp;

context.sendResponse("Today's temperature is : "+temp);

}

The header is always passed as a JSON object.

**GET call with a callback function example**: Here we are making a call to a date API and parsing the response in our own callback function.

function MessageHandler(context, event) {

if(event.message== "date") {

context.simplehttp.makeGet('http://date.jsontest.com/',null,parser);

}

}

function parser(context, event) {

var dateJson = JSON.parse(event.getresp);

var date = dateJson.date;

context.sendResponse("Today's date is : "+date);

}

In the above example response of the Get call will be sent to **parser** which is defined as the callback function while making the API call. Also, since there is no header required, we need to pass it as null for this to work.

The order of fields in this GET call is:

1.    URL

2.    Headers

3.    callback function (optional)

Hence the complete syntax would be - **context.simplehttp.makeGet(url, headers, callback function)**

Syntax sequence has to be maintained at all time. While using the callback function make sure you pass "null" if no header is passed

## POST Call

**POST call example**: Here the example talks about sending name and number to a server.

function MessageHandler(context, event) {

if(event.message. == "postdata") {

var url = "https://abcserver.com/sm/postData";

var header = {"token":"ca916a68d94","Content-Type": "application/x-www-form-urlencoded"};

var param = "userName=John&phoneNumber=8431792747";

context.simplehttp.makePost(url,param,header);

}

function HttpResponseHandler(context, event) {

var result= JSON.parse(event.getresp);

if(result=="success")

context.sendResponse("We have successfully stored your data");

}

By default, the response form the POST call is also sent to the 'HttpResponseHandler'.

**POST Call with a different content type example**: Here we are making a POST call with “content-type:application/json”.

function MessageHandler(context, event) {

if(event.message=='hi'){

var contextParam = {

"User": {

"userName": "sbCobxxxx",

"Password": "xxxxxxx-9f-4307-9d9a-451f3xxxx075",

"Pin": "16776"

}

};

var url = "https://developer.api.abc.com/restserver/v1/user/login";

var param = JSON.stringify(contextParam);

var header = {"Content-Type": "application/json"};

context.simplehttp.makePost(url, param, header);

return;

}

}

**Making a POST call with callback function**: In the above example just define the callback function at the end.

function MessageHandler(context, event) {

if(event.message. == "postdata") {

var url = "https://abcserver.com/sm/postData";

var header = {"token":"ca916a68d94","Content-Type": "application/x-www-form-urlencoded"};

var param = "userName=John&phoneNumber=8431792747";

context.simplehttp.makePost(url,param,header,parser);

}

function parser(context, event) {

var result= JSON.parse(event.getresp);

if(result=="success"){

context.sendResponse("We have successfully stored your data");

}

}

The order of fields in this POST call is:

1.    URL

2.    Parameters

3.    Headers

4.    callback function (optional)

Hence the complete syntax would be - **context.simplehttp.makePost(url, formParams, headers, callback function)** .

Syntax sequence has to be maintained at all time. While using the callback function make sure you pass "null" if no header/parameters are passed

That is all for this tutorial. Do watch the related video for more understanding through live code example.

# Adding Image, Audio, Video and File Attachment.

Most of the messaging platforms like Facebook, Viber, MS teams, Slack etc allows bots to send these media files to users which gets displayed directly inside the messenger. Bots can now send pdf files and doc files as an attachment .

Let's understand how to send these attachments from a bot built using Gupshup.

1. [Image Attachment](https://www.gupshup.io/developer/docs/bot-platform/guide/audio-video-file#image)
2. [Audio Attachment](https://www.gupshup.io/developer/docs/bot-platform/guide/audio-video-file#audio)
3. [Video Attachment](https://www.gupshup.io/developer/docs/bot-platform/guide/audio-video-file#video)
4. [File Attachment](https://www.gupshup.io/developer/docs/bot-platform/guide/audio-video-file#file)

## Image Attachment

Given that all messaging apps support sending and receiving images, your bot can also send and receive images from the user. You have to specify the URL of the original image as well as the URL of the thumbnail.

#### Sending images to the user

Below is the payload of an image file:

Type: Image   
originalUrl: image URL here   
previewUrl: thumbnail URL here

Example:

if(event.message.toLowerCase() == "batman") {

var image = {"type":"image","originalUrl":"image url here","previewUrl":"thumbnail url here"};

context.sendResponse(JSON.stringify(image));

}

Here's how the image will look on Facebook Messenger



#### Receiving Images from the user

When a user sends an image to the bot, the message is in this format:

channel: fb

contextobj: {

"botname": "demobot1",

"channeltype": "fb",

"contextid": "1067740686631985",

"contexttype": "p2p"

}

senderobj: {

"channelid": "1067740686631985",

"channeltype": "fb",

"display": "John Doe",

"subdisplay": ""

}

botname: testBot

messageobj: {

"text": "someTextHere",

"type": "image",

"url": "imageUrlHere"

}

You can access this message using event.contextobj, event.senderobj and event.messageobj. Here's a screencast that illustrates the above example.

## Audio Attachment

To send an audio clip, upload that clip on a server in mp3 format and then send it to the user using the JSON mentioned below.

{

"type": "audio",

"url": "https://a.clyp.it/v3rprye4.mp3"

}

Change the value of “url” as per your file location.

You need to send this JSON as a string and Gupshup will make it work behind the scene.

As per Facebook,the maximum allowed attachment size for audio is 10 MB.

If you were using the hosted code on the bot builder then the code to send this audio attachment is :

if(event.message=='audio'){

var audioObj = {

"type":"audio",

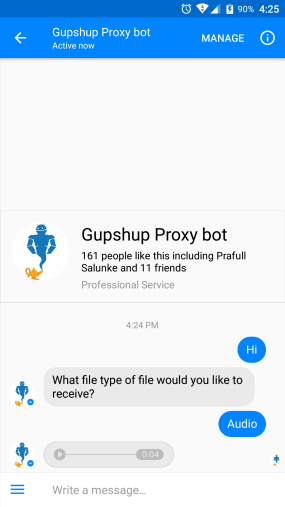
"url":"https://a.clyp.it/v3rprye4.mp3"

};

context.sendResponse(JSON.stringify(audioObj));

}

This is how it will render on Facebook messenger :



## Video Attachment

To send a video clip, upload that clip on a server and then send it to the user using the JSON mentioned below.

{

"type":"video",

"url":"http://clips.vorwaerts-gmbh.de/big\_buck\_bunny.mp4"

}

Change the value of “url” as per your file location.

You need to send this JSON as a string and Gupshup will make it work behind the scene.

If you were using the hosted code on the bot builder then the code to send this video attachment is :

if(event.message=='video'){

var videoObj = {

"type":"video",

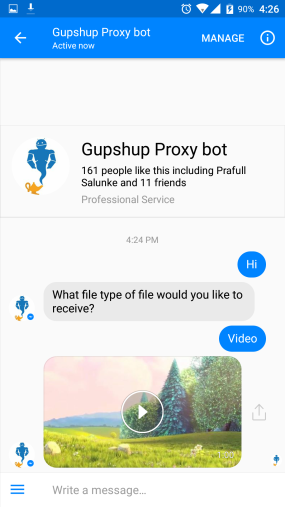
"url":"http://clips.vorwaerts-gmbh.de/big\_buck\_bunny.mp4"

};

context.sendResponse(JSON.stringify(videoObj));

}

This is how it will render on Facebook messenger :



## File Attachment

To send a file attachment, upload that file on a server and then send it to the user using the JSON mentioned below.

{

"type":"file",

"url":"http://unec.edu.az/application/uploads/2014/12/pdf-sample.pdf"

}

Change the value of “url” as per your file location.

You need to send this JSON as a string and Gupshup will make it work behind the scene.

If you were using the hosted code on the bot builder then the code to send this video attachment is :

if(event.message=='file'){

var fileObj = {

"type":"file",

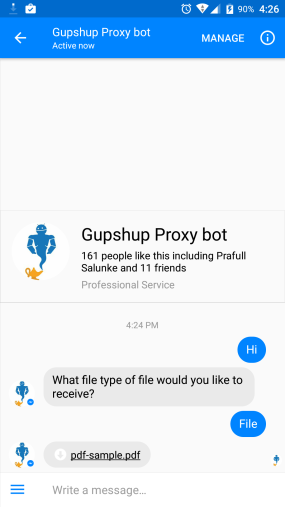
"url":"http://unec.edu.az/application/uploads/2014/12/pdf-sample.pdf"

};

context.sendResponse(JSON.stringify(fileObj));

}

This is how it will render on Facebook messenger :

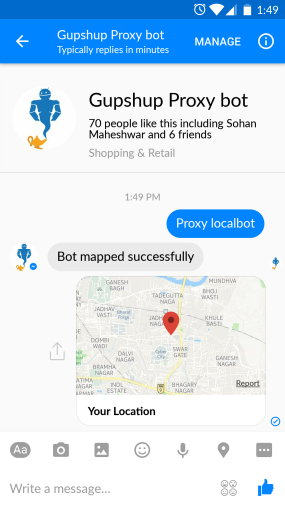


# Handling location from a user

Given that most messaging apps support sending location, users can send their location to your bot. You will receive their latitude and longitude along with a link to Google Maps.

#### Receiving location from a user

When a user sends his/her location to the bot,



the message JSON sent to the bot is as below:

{

"botname": "demobot2",

"contextobj": {

"botname": "demobot2",

"channeltype": "telegram",

"contextid": "195173611",

"contexttype": "p2p"

},

"messageobj": {

"address": "",

"latitude": "18.519825",

"longitude": "73.929449",

"text": "https://www.google.com/maps/?q=18.519825,73.929449",

"type": "location"

},

"senderobj": {

"channelid": "195173611",

"channeltype": "telegram",

"display": "Shreyans"

},

"channel": "telegram",

"sender": "195173611",

"message": "https://www.google.com/maps/?q=18.519825,73.929449"

}

There are two ways of handling the location into your bot code:

1.   **Inside the MessageHandler function**: You can handle the location in the MessageHandler by parsing out the event object.

        Sample code to achieve this:

function MessageHandler(context, event) {

if(event.messageobj.type=='location'){

var lat = event.messageobj.latitude;

var lang = event.messageobj.longitude;

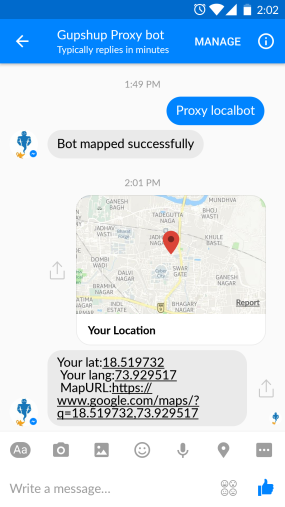
var url = event.message;

context.sendResponse("Your lat:"+lat+"\n Your lang:"+lang+"\n MapURL:"+url);

}

}

        Here is how the user will recieve your message -



2.   **Inside the LocationHandler function** - If you do not want to handle the location into the MessageHandler then define a function called **LocationHandler** into your code and any location sent from the user to the bot will be forwarded directly to this handler.

        Sample code for this -

function LocationHandler(context,event){

var lat = event.messageobj.latitude;

var lang = event.messageobj.longitude;

var url = event.message;

context.sendResponse("Your lat:"+lat+"\n Your lang:"+lang+"\n MapURL:"+url);

}

This will be displayed in the same way as above.

# Making HTTP call to a Bot

We have provided an option to make an HTTP call to a bot and trigger any function inside that bot. The bot has control over what values will be sent as a response to that HTTP call. The handler in the bot code which gets invoked upon this HTTP call is **HttpEndpointHandler**. There are many usages of this HTTP call to the bot such as - sending a scheduled message to a user or getting data from the bot for storing at your own server and much more.

In this guide we will learn two things:

1. How use HTTP call to a bot and configure bot's response
2. Using the above we will learn how to send scheduled messages.

#### How to use HTTP call to a bot

First, let's see how to respond to the HTTP call to the bot.   
You need to define the below function into the bot code and send out a response when the HTTP call is made.

function HttpEndpointHandler(context, event){

context.sendResponse("Hello there");

}

The URL to make this HTTP call is

https://www.gupshup.io/developer/bot/{botname}/public

You will need to replace the {botname} with the actual bot name to which HTTP call will be made.

Sample response of the above code -   


Now let's see how you can send some parameters through the HTTP call to the bot and read it through bot and utilize them.   
Say you will send an additional parameter named **text** which will contain a message, the URL will look like

https://www.gupshup.io/developer/bot/{botname}/public?text=Hello

Code to handle this will be:

function HttpEndpointHandler(context, event) {

var message = event.params.text;

context.sendResponse("You said : "+message);

}

Response of the above code to the HTTP call:   


The method to extract parameters sent is:

event.params.parameterName

The complete JSON of sent to the **event** parameter is

{

"botname": "demobot10",

"context": {

"botname": "demobot10",

"channel": "http",

"type": "http"

},

"params": {

"text": "Hello"

},

"request": {

"ip": "59.144.141.242",

"method": "GET",

"path": "/public"

}

}

#### Sending scheduled message through the bot

Let's take an example to understand this.

Say, you have built a newsletter bot to which users can subscribe. Once a user subscribes, you will need to send messages at certain intervals to the user as notification. Hence, you will need to utilize two features -

* The bot send message API.
* HTTP call to the bot to trigger the send message API.

Here are the complete steps which have to be followed by a developer:

1.   Store the **contextobj** using **dbPut** method into the hosted database when user messages your bot for the first time and use the key for db storage as **event.sender**.**contextobj** is available in the **event** object and can be accessed as **event.contextobj**.

Refer this [guide](https://www.gupshup.io/developer/docs/bot-platform/guide/different-objects-sent-to-bot) for ***contextobj*** and other objects. Refer to the [advanced persistence guide](https://dev-smapi.gupshup.io/developer/docs/bot-platform/guide/adding-advanced-persistence) to understand the***dbPut*** method

2.   Make an API call to your webservice by passing the **event.sender** and store that at your end as the key to getting **contextobj**. 

3.   To send a message to the user, just make a GET call to the bot URL and pass the key and the message to the URL.

Example - <https://www.gupshup.io/developer/bot/botname/public?key=106740686985&message=how%20are%20you?>

Botname has to be replaced with your bot's name to which you are making the call. Key is the ID that you had stored in step 2

4.   This HTTP call to the bot in #3 invokes the **HttpEndpointHandler** function in the bot builder code, from where you can make a API call to **/bot/{botname}/msg** API by pulling out the **contextobj** from the hosted database.

For bot send message API, Go to this [link](https://www.gupshup.io/developer/sm/apis.jsp?source=ent&apikey=undefined&username=undefined&userId=undefined#!/IP_Messaging/postMessage)

5.   Setup a scheduler to make the HTTP call to the bot at desired intervals.

**Sample code of the bot builder**:

/\*\* This is a sample code for your bot\*\*/

var message = null;

function MessageHandler(context, event) {

context.console.log("test")

if(event.message=='hi') {

context.simpledb.doPut(event.sender,event.contextobj);

}

}

function HttpEndpointHandler(context,event){

var dbkey= event.params.key

message = event.params.message;

context.simpledb.doGet(dbkey);

}

function DbGetHandler(context, event) {

var url = "https://api.gupshup.io/sm/api/bot/botname/msg"; //The bot which is sending the message to the user.

var header = {"apikey":"your api key","Content-Type": "application/x-www-form-urlencoded"}; // you can get the API key from https://www.gupshup.io/developer/docs/bot-api/bot-api-ref

var contextobj = event.dbval;

var param = "context="+contextobj+"&message="+message;

context.simplehttp.makePost(url,param,header);

}

function DbPutHandler(context, event) {

context.sendResponse("Hello "+event.sender);

}

function HttpResponseHandler(context,event){

context.sendResponse(event.getresp);

}

You will have to set the scheduler for each user and make the corresponding HTTP call to the bot.

# Sending multiple messages

Before reading this guide please make sure you have gone through the previous IDE Bot Builder guides

To send multiple messages through the IDE Bot Builder you need to create an array which contains all the messages that you want to send and then use **context.sendResponse()** method to send that array. Our platform will then send all the messages from the array one by one.

Let's say, if a user says “hi” to the bot then you want to send a response like “hello username” followed by an image. The code to achieve this would be:

function MessageHandler(context, event) {

if(event.message=="hi"){

var message = [

"Hello "+event.senderobj.display,

{"type":"image","originalUrl":"http://wallpaper-gallery.net/images/smile-image/smile-image-11.jpg","previewUrl":"http://wallpaper-gallery.net/images/smile-image/smile-image-11.jpg"}

];

context.sendResponse(JSON.stringify(message));

return;

}

In case you want to send 3 messages then just add one more message into the array.

If you are using the **Link your bot** option on our platform then you can use the **/bot/{botname}/msg** API. You need to pass the **contextobj** into this API which is sent to the bot everytime the user sends a message.

Here is the curl snippet for the API call :-

curl -X POST --header 'Content-Type: application/x-www-form-urlencoded' --header 'Accept: text/plain' --header 'apikey: Your API key goes here' -d 'context= "your url encoded contextobj goes here"&message=Hello%20there' 'https://api.gupshup.io/sm/api/bot/demobot/msg'

The API key can be found by clicking on your profile picture on the top right. Also, You must store the **contextobj** for future communications as it is required by this API.

This API can also be used to send **scheduled notification** to users.

# Putting it all together

In our earlier guides you have seen how to handle conversation, how to persist data and how you can make HTTP calls to APIs. In this guide, we'll put it all together by creating a simple chat bot that displays the day's top news story from the RSS feed of a tech publication, based on the user's choice.

#### Handling Conversations

Lets ask the user what his/her reading preference is. Here's how the messageHandler() function will look:

function MessageHandler(context, event) {

context.console.log("test")

if(event.message.toLowerCase() == "hi") {

context.sendResponse("hello");

}

else if(event.message.toLowerCase() == "hello") {

context.sendResponse("Hey there " + event.sender + " Do you prefer reading Wired or Techcrunch?");

}

else {

context.sendResponse('No keyword found : '+event.message);

}

}

#### Adding Persistence

function MessageHandler(context, event) {

context.console.log("test")

if(event.message.toLowerCase() == "hi") {

context.sendResponse("hello");

}

else if(event.message.toLowerCase() == "hello") {

context.sendResponse("Hey there " + event.sender + " Do you prefer reading Wired or Techcrunch?");

}

else if((event.message == "wired") || (event.message == "techcrunch")) {

setPreference(event.message);

}

else {

context.sendResponse('No keyword found : '+event.message);

}

}

When you ask the user his/her preference, make a call to the setPreference() method to set the roomleveldata to this preference. Roomleveldata allows you to set different preferences for each user on each messaging channel.

function setPreference (pref) {

context.simpledb.roomleveldata.publication = pref;

context.sendResponse("Type 'news' to get latest headlines on " + context.simpledb.roomleveldata.publication);

}

#### Making the HTTP call

Now anytime the user enters a message with the word 'news', make a HTTP call to the RSS feed of the publication the user prefers. We'll use a Google API to convert RSS to JSON.

else if (event.message.indexOf("news") > -1 ) {

context.simplehttp.makeGet('https://ajax.googleapis.com/ajax/services/feed/load?v=2.0&q=http://www.' + context.simpledb.roomleveldata.publication + '.com/feed/&num=15');

}

You can handle the response to this GET call in the HttpResponseHandler() method. Here, we have initially queried for 15 news articles and will display a single random article.

function HttpResponseHandler(context, event) {

var respJson = JSON.parse(event.getresp);

var stories = respJson.responseData.feed.entries;

var resp = "";

//generate a random number

var randomnumber = Math.floor(Math.random() \* (stories.length - 1 + 1)) + 1;

resp = resp + stories [randomnumber].title + "\n" + stories[randomnumber].link + "\n";

resp = resp.replace("&nbsp", "");

context.sendResponse(resp);

}

That's it. You now have a fully functioning bot that can read from RSS feeds. You can check out [our blogpost](http://blog.gupshup.io/index.php/2016/05/23/five-tips-to-create-a-well-behaved-chat-bot/) for guidelines to creating better bots.

Happy bot building!

[**Here is the Sample code**](https://github.com/GupshupBot/sampleCode/blob/master/Sample%20Code%20-%20Putting%20it%20all%20together.js)