**Introduction to bot scripting**

Gupshup's IDE Bot Builder is used by thousands of bot makers around the world to create advanced and powerful bots. We're constantly improving our bot platform to ensure that our creators have access to cutting-edge technology. We have just launched v2 of our IDE Bot Builder with a ton of new features to improve the bot-building experience. Read [our blogpost](http://blog.gupshup.io/index.php/2017/06/29/introducing-the-new-ide-bot-builder/) on some the new features that the new IDE provides.

A revolutionary feature of the new Bot Builder tool is the ability to create bot conversations using a scripting tool. This allows anyone to map out conversation paths between a bot and a user, without the need for any coding knowledge. The user's conversation has Auto-NLP enabled that matches it to the closest intent specified in the script file. The scripting tool uses Gupshup's own '[NLP on the fly]'(https://www.gupshup.io/developer/ent-apis) API. This guide will illustrate how you can use the bot scripting feature to create advanced bots with API calls, structured messages, entity extraction, slot filling and a whole lot more.

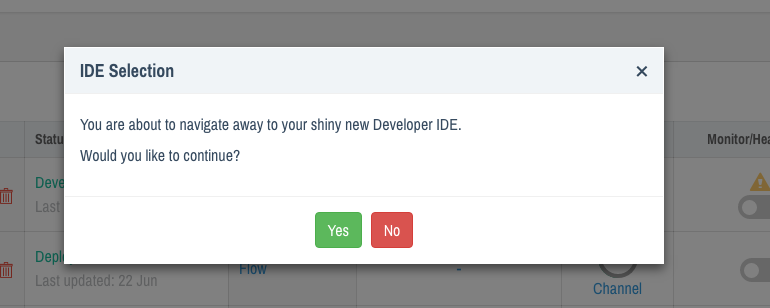
**Terminology**

* Script file: A script file is a text file with the extension .scr. Script file contains bot flows organized as modules.
* Module: A module is a unit of bot flow that performs some useful task. The module can be thought of as a tree with each branch representing a separate flow.
* Element: An element is a single node in the module tree. Conceptually, it represents a message that is either sent by the user or by the bot.
* Bot Output: A bot output is an element where the bot sends a message to the user.
* User Input: A user input is an element where the input by a user is parsed.

**Getting Started**

To familiarise yourself with the layout of the new IDE, read [this guide](https://www.gupshup.io/developer/docs/bot-platform/guide/new-ide-helper-guide).

To get started with scripting your Hello World bot, create a new project in your dashboard. You will see an option to select your IDE. Click on the 'Yes' button.



The new IDE will open up. In the explore section locate the file named 'default.scr' which contains some template script. This file is where you will create your bot script. Clear the existing script and type [main] in the first line to create the Main module.

**It is important to note that the bot scripting tool works on indentation.**

Set the welcome message as 'Welcome to the scripting tool'. This line is what the bot sends to the user. The script alternates between what the bot says and what the user says (unless specified). The following line is the expected response from the user. Ensure that this is indented more than the previous line. Put the bot response 'Hello World' in the next line and make sure it is indented more than the previous line. This is how your script will look like:

[main]

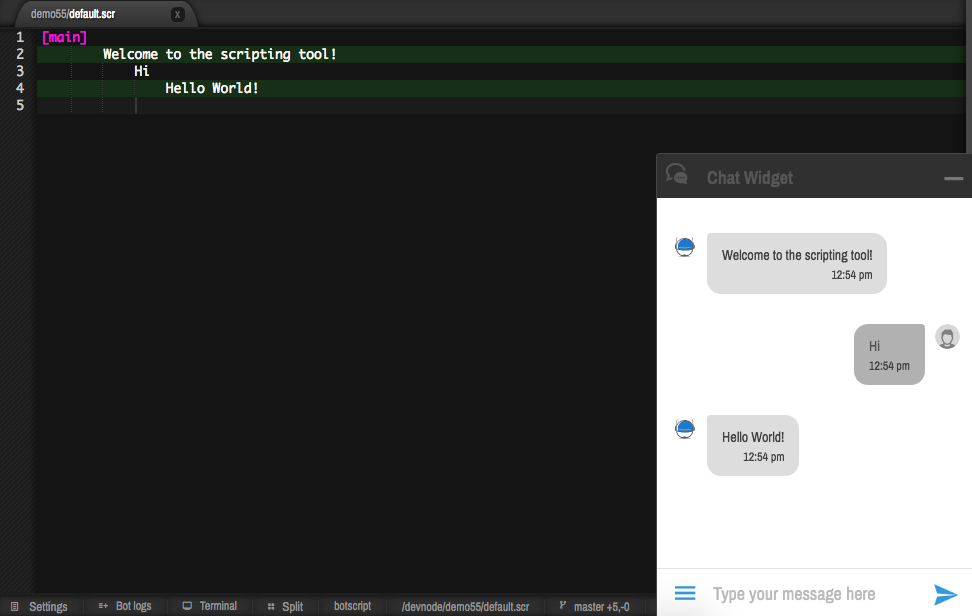
Welcome to the scripting tool!

Hi

Hello World!

The scripting tool automatically adds NLP to the user input, in this case 'Hi'.

To test the bot out, right click on your bot in the left Explore menu and select 'Start Server' from the menu that appears. Once the server is started, click on chat widget in the bottom bar. The chat widget will appear and in a few seconds, it will show the 'Welcome to the scripting tool!'. Type Hi and it will respond with Hello World.



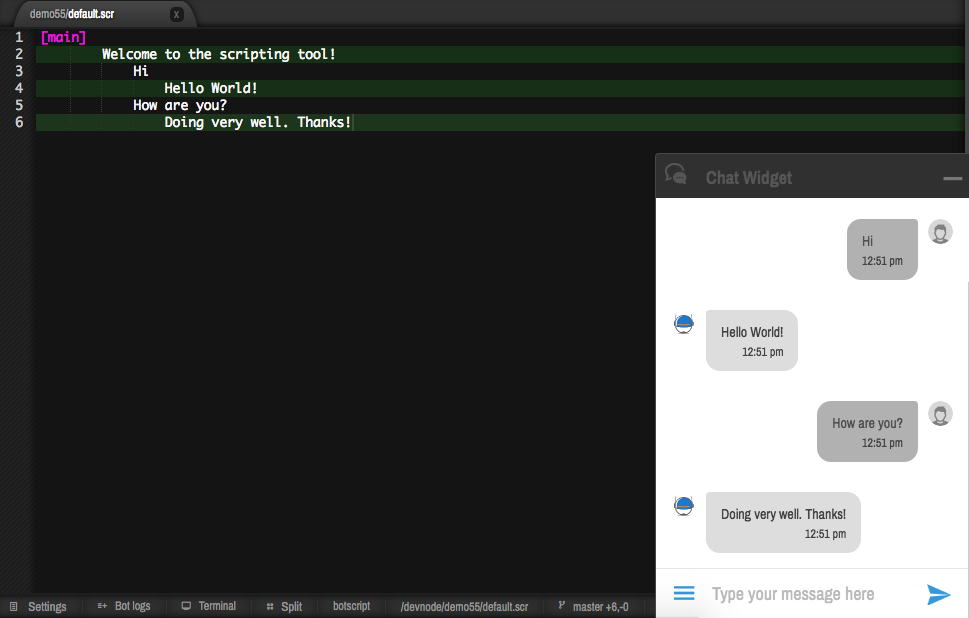
**Indentation**

As mentioned earlier, the scripting tool relies on indentation to differentiate between a 'user reply' and a 'bot reply'. Indentation rules are inspired by the natural rules for bullets and numbered list. Barring exceptions (see: Sending Multiple Messages) When a statement is indented it becomes a 'child' of the statement above it.

Please note that if you get the indentation wrong, the entire conversation flow will change.

**Rules for indentation**

1. In order to create a child, you need to indent it more than the parent
2. Elements that have the same parent and are at the same indentation levels are siblings.
3. If any element has more than one child, it is a decision point.
4. The conversation till an element consists of the element and each of its ancestors (i.e. parent, parent of parent and so on)
5. User inputs and bot outputs typically alternate. The same are highlighted in different colors in the IDE.
6. An exception to 5 is when we have multiple bot messages (next topic)
7. All children of an element must be of the same type. It is not possible to have some children of user input and some of bot output.



In the above example, 'Hi' and 'How are you' are siblings as they're in the same indentation level. They are both 'child' to 'Welcome to the Scripting Tool!'

# Sending Multiple Messages

We can send two consecutive messages to the user without waiting for user input in between. In order to do so type :continue after the first message. Make sure that the following message is indented more than the previous one. Here?s an example:

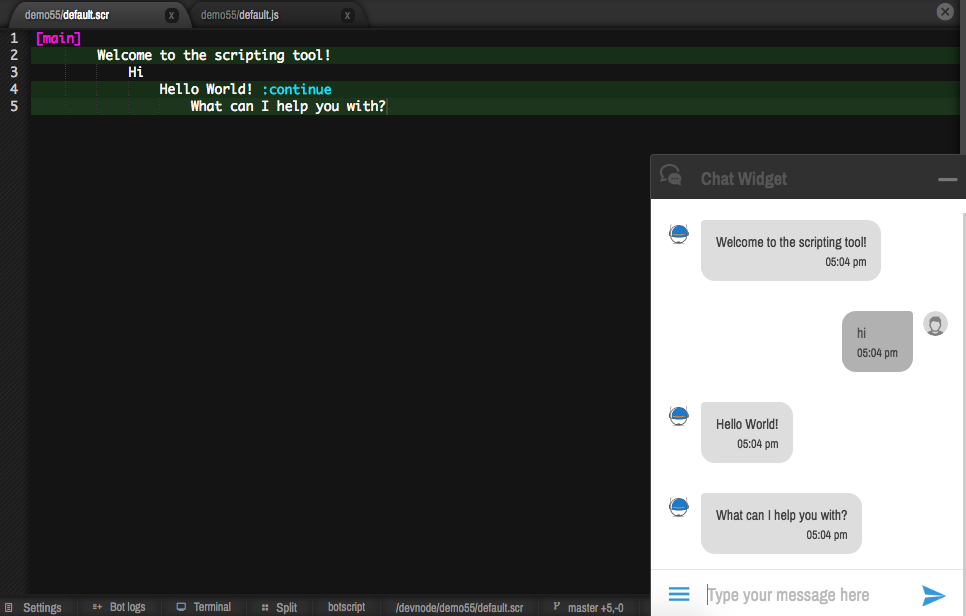
[main]

Welcome to the scripting tool!

Hi

Hello World! :continue

What can I help you with?



### Adding a delay

You can also choose to add a delay in between bot messages using the keyword 'delay'. Here's an example:

[main]

Welcome to the scripting tool!

Hi

Hello World! :continue

((delay 2000))

What can I help you with?

Use the 'delay' keyword in double brackets and specify the time in milliseconds. You also need to provide your apikey in index.js for the delay to work.   
In your scriptHandler() function in index.js add this line of code:

options.apikey = "YOUR-API-KEY";

# Labels and Handlers

Labels are used to refer to particular states. These can then be used to attach code snippets (handler) to these states.

You can assign a label just before starting a message. Labels can be attached to user messages and bot messages. The label is alphanumeric and is followed immediately by a colon. When you create a label, the editor will highlight it in a different color. Here's an example **bot1** is the label attached to the first bot message.

[main]

bot1: Welcome to the scripting tool!

Hi

Hello World! :continue

What can I help you with?

Do note that no spaces or special characters are allowed in a label. Also, the indentation is determined by the first character in a line. Hence adding a label means the indentation starts from the label name.

#### goto

In order to go from one step to another, prepend :goto [destination label] to the source message. You can only go to steps within the same module. A cross module goto is not allowed. Example:

[main]

bot1: Welcome to the scripting tool!

Hi

Hey there. Do you want to continue or start from the beginning?

Continue

Sure thing!

Start

No worries :goto bot1

Flow control directives such as ':goto' are applicable **after** the corresponding message is sent. Also, a message with a ':goto' cannot have children.

### Handlers

Labels are used to refer to particular states. This is useful when you have to attach code to a message. The code that the label refers to is called a **handler**.

In order to create a handler, we first need to create a JavaScript file with the same name as the script file but a .js extension in place of .scr. In this example we will create default.js. To create a new file, right-click on your project folder in the explorer pane and choose 'New File'.

In this file, you need to export an object for each module. The name of this exported object must be the same as the module name. Further, each exported object can have a handler for any labelled step. The function takes 4 arguments (options, event, context and callback) and must call callback with the other three (options, event, callback).

In the below example there are labels for all three statements. The labels are **bot1, user1 and bot2** respectively.

[main]

bot1: Welcome to the scripting tool!

user1: Hi

bot2: Hello World!

The default.js file contains the handler for each label. You also have to export each label to a handler.

function bot1handler (options, event, context, callback) {

options.next\_state = 'user1';

callback (options, event, context);

}

function user1handler (options, event, context, callback) {

options.next\_state = 'bot2';

callback (options, event, context);

}

function bot2handler (options, event, context, callback) {

callback (options, event, context);

}

module.exports.main = {

bot1: bot1handler,

user1: user1handler,

bot2: bot2handler

}

The module.export.main associates a label with a handler. You can give custom names to these handler functions. In function bot1handler and user1handler we explicitly mention the next state by giving a label name.

# Modules

Structured programming requires functionality to be broken into modules that can be called. The Bot Scripting tool supports modules which can reference each other. To create a module, just add a line that contains the module name in square brackets. Each bot that is created has a [main] module by default. The full module name consists of the entire folder hierarchy, the file name and the module name each separated by a period (.)

In order to call a module, append ':call [full module name]'' to the message from which it needs to be called. To return back from a module, append :return to the message from where it needs to return. Here we create a module named [profile] and call it from the [main] module.

[main]

Hey! Do you want to create your user profile first?

Yes

Lets create your profile :call default.profile

Your profile is now created!

[profile]

What is your name

name

Thanks! Where are you from?

Place

Super. We're all done :return

If you are calling a module from a message, it needs to have a child. This child will always be of type bot message and will be displayed when the called module returns. In the above example the child message is “Your profile is now created!“.

Flow control directives such as :call and:return are applicable **after** the corresponding message is sent. Also, a message with a ':return' directive cannot have children

# Dynamic Messages

Dynamic messages are messages whose contents can be changed programmatically.   
The bot messages in the scripting tool are really templates. You can create dynamic parts by enclosing the name of the part in double braces. Please note that part name must only be alphanumeric and without whitespace.

Consider a situation where a user message labelled as user1 is followed by a bot message labelled as bot2. If this user message is Hello {{name}}, and the value of name is set as Bruce Wayne, the message shown will be Hello Bruce Wayne.

In our script file, we shall add the {{name}} variable. Here is how the script file will look:

[main]

bot1: Welcome to the scripting tool!

user1: Hi

bot2: Hello {{name}}

In our default.js file we will have to set the value of name. Since the label associated with name is bot2, we shall set the value for name in the bot2handler function. Here?s how the default.js file will look:

function bot1handler (options, event, context, callback) {

options.next\_state = 'user1';

callback (options, event, context);

}

function user1handler (options, event, context, callback) {

options.next\_state = 'bot2';

options.data.name = "Bruce Wayne"; //name is set here

callback (options, event, context);

}

function bot2handler (options, event, context, callback) {

callback (options, event, context);

}

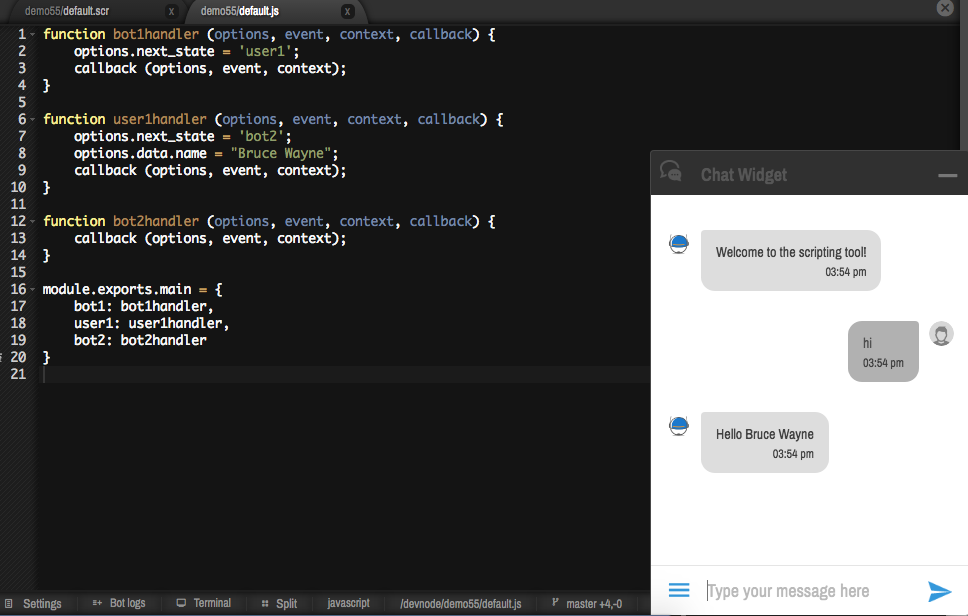
module.exports.main = {

bot1: bot1handler,

user1: user1handler,

bot2: bot2handler

}



# Making an API call

Making calls to external APIs is a simple process. We have already described the syntax and how to handle HTTP responses in [this document](https://www.gupshup.io/developer/docs/bot-platform/guide/making-http-calls). You can use the same syntax along with labels and handlers, to trigger an API call from the bot scripting tool. This guide will illustrate how to make API calls in conjunction with the bot scripting tool.

The earlier guide discussed the addition of [labels and handlers](https://www.gupshup.io/developer/docs/bot-platform/guide/labels-and-handlers) to user messages and bot states in the script file. These labels and handlers enable you to preprocess or postprocess data that the bot says. When you attach a label to a state, the handler associated with that label executes code. The code for the HTTP call is inside the handler. Here?s an example. Let us create a simple bot that tells us the time.

We shall create a new project and in the script file (default.scr), we can define the script as

[main]

Hello. I?m a bot and I can tell you the time!

timeLabel: What is the time

The time is {{time}}

Create a file named default.js and in that file we have to set the value of time. Since the label associated with time is timeLabel, we shall set the value for name in the timeHandler function. Here?s how the default.js file will look:

function timeHandler (options, event, context, callback) {

context.simplehttp.makeGet ("http://time.jsontest.com", {}, function (context, event) {

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

options.data.time = res.time;

callback(options, event, context);

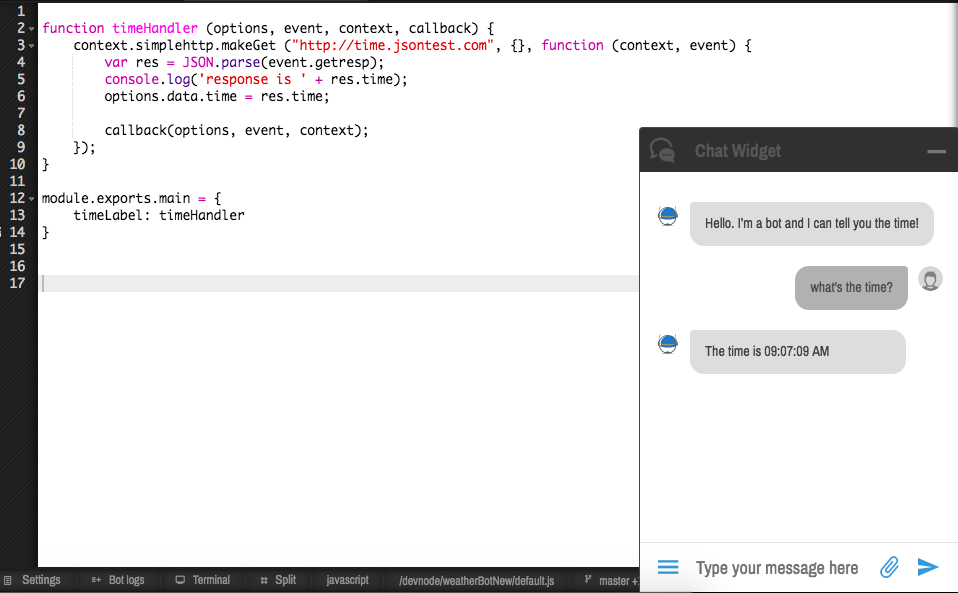
});

}

module.exports.main = {

timeLabel: timeHandler

}



The syntax to make GET, POST and PUT calls are all discussed in [this document](https://www.gupshup.io/developer/docs/bot-platform/guide/making-http-calls).

# Creating Branches

We can create messages that may be have multiple branches and then programmatically select a branch. In order to create a branch, a message element must have multiple children.   
This is done by writing two lines at the same indent level. Ensure the indent level is more than that of the parent element. For instance:

[main]

bot1: Welcome to the scripting tool!

user1: Hi

bot2: Hello {{name}}

bot2\_1: Hey there!

Here bot2 and bot2\_1 are sibling branches with the parent being user1. Now in the handler for user1 you can programmatically define which message should be displayed after user1.

function bot1handler (options, event, context, callback) {

options.next\_state = 'user1';

callback (options, event, context);

}

function user1handler (options, event, context, callback) {

// programmatically define which message should be displayed

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

options.next\_state = 'bot2';

options.data.name = "Bruce Wayne";

}

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

options.next\_state = 'bot2\_1';

}

callback (options, event, context);

}

function bot2handler (options, event, context, callback) {

callback (options, event, context);

}

module.exports.main = {

bot1: bot1handler,

user1: user1handler,

bot2: bot2handler

}

# Multiple Inputs

So far we have viewed examples which had just the one user response. You can create multiple input options as well. The scripting tool ships with a default NLP parser which matches what the user has said with all the input options at the same level and chooses the closest match. For instance an a bank bot a user might want to 'Enquire' or 'Transact' - these are the two input options. You can create a script file like this:

[main]

Welcome to the scripting tool!

Enquire

Fulfillling your enquiry

Transact

Completing your transaction

Enquire and Transact are indented at the same level. Whatever the user types out will be matched 'Enquire' and 'Transact'. The auto-NLP parser ensures that each input has NLP built-in already. Thus if a user says ?I want to enquire' or ?how do I enquire? it will match with ?Enquire?. Context management is built-in to the scripting tool. You can have as many input options as you'd like.

You can even have input options within input options. For instance if you want to present more options after entering ?Transact?, you can do so like this:

[main]

Welcome to the scripting tool!

Enquire

Fulfilling your enquiry

Transact

Would you like to send or receive money?

send

Enter the account number to send to

receive

Enter the account number to receive from

Here the options to 'Send' and 'Receive' are defined under the 'Transact' input.

#### Variations

We may want multiple variations of of user input to mean the same. In order to support different variations for user inputs, just separate them by a pipe - |   
Here's an example:

[main]

Welcome to the scripting tool!

Enquire | Balance

Fulfilling your enquiry

Transact

Would you like to send or receive money?

Here the NLP parser will check for variations of 'Enquire' as well as 'Balance'.

# Entity Extraction

The NLP parser also comes with an entity extraction mechanism. Entity extraction refers to parsing user inputs to get desired information.   
In order to extract entities, begin with a normal user input and enclose the entities in double braces. Then, prefix a label for each entity followed by a colon.   
Sample format: **I want a {{color:blue}} {{item:shirt}}**. You can then use these entities in a bot response. Example:

[main]

Welcome to the scripting tool!

I want a {{color:blue}} shirt

Here are your options for a {{color}} shirt

You can use multiple entities in a single statement as well

[main]

Welcome to the scripting tool!

I want a {{color:blue}} {{item:shirt}}

Here are your options for a {{color}} {{item}}

# Structured Messages

You can create structured messages such as quick replies, polls and carousels using the scripting tool.

#### Quick Reply

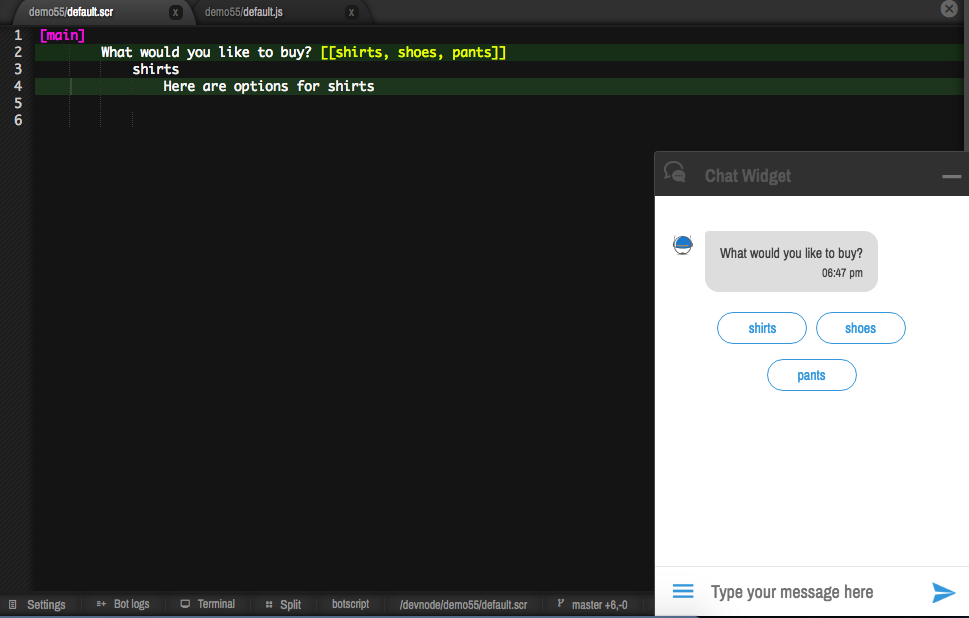
We create a bot message that has quick reply buttons to allow the user fast access to a button rather than typing. The quick replies are ephemeral. Ie. they disappear after the user answers. In order to show quick reply buttons, just type the bot message followed by the options (comma separated) in a double square bracket. Example:

[main]

What would you like to buy? [[shirts, shoes, pants]]

shirts

Here are options for shirts



#### Buttons

We send a message with buttons. Buttons are different from quick replies in that they are always visible even after the user answers to the message. In general, you can set structured messages via the scripting tool just by setting the appropriate JSON. The JSON structure is similar to what is described here

**Sample JSON**:

{

"type": "survey",

"question": "Hello World",

"options": ["Continue", "Main Menu"]

}

This will show a bot message as Hello World followed by two buttons, one with a Continue label and the other with a Main Menu label.

#### Carousel

A carousel can take an image, title, subtitle, description and buttons to request input from the user. This template can support multiple bubbles per message and display them in a horizontally scrollable form. Just like buttons, you can set a carousal JSON to get a carousal.

**Sample JSON**:

{

"type": "catalogue",

"items": [{

"title": "Vote in a Poll",

"subtitle": "Cast your vote now",

"imgurl": "https://s3.amazonaws.com/gs-bot-images/Pollbot/Vote+image.jpg",

"options": [{

"type": "text",

"title": "Vote now"

}]

}, {

"title": "Create a Poll",

"subtitle": "Create your own poll and share with friends",

"imgurl": "https://s3.amazonaws.com/gs-bot-images/Pollbot/create+a+poll.png",

"options": [{

"type": "text",

"title": "Start creating"

}]

}]

}

# Adding a fallback, NLP threshold and small talk components

The scripting tool gives you the option to add a default error message if what the user says cannot be parsed by the bot.

Every project has a file named index.js. Double-click the index.js file and and look at the ScriptHandler() method. The options.default\_message variable prints out the default fallback message. You can change it to your preferred message.

### NLP Threshold

The scripting tool makes use of the [NLP on the Fly](https://www.gupshup.io/developer/docs/bot-platform/guide/introduction-to-bot-scripting) API to match user messages with the intents that have been defined. Each time the intent is matched, a NLP Threshold is returned internally. The default value for this threshold is 0.2 but you can change that to your preferred value.   
In the ScriptHandler() method, add

options.nlp\_threshold = value;

Where value is a number between 0 and 1.

### Small Talk

To add small talk components to your bot, just add a separate module named '[common]'. Typically, small talk questions such as greetings and access-to-help menus are written in the [common] module. A powerful feature of the bot scripting tool is that you do not have to take care of state and context switching to reply to these small talk questions from users. At any point, a user can ask a question from this list and the bot will answer it and then switch the context back.

[common]

Hi

Hello

Who are you?

I'm a bot that can help you with the time

Help | menu

Type the word 'time' to know what time it is!

Since the [common] module is pre-defined and the initial message is a user message, ensure that there is no indentation for the user messages in the top-level messages.

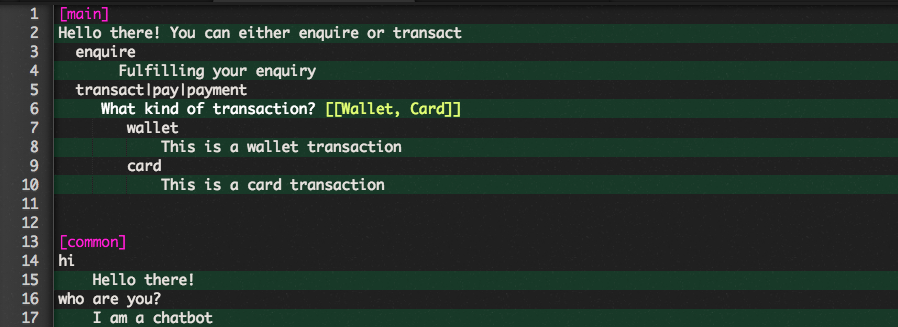
## State Management

State is auto-managed by the scripting tool. Scripting tool stores the last user state and then does the intent mapping based on the state user conversation it is at.

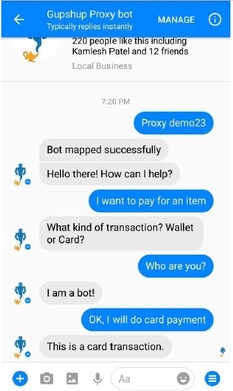
Although for the small talk module the state is not managed. Scripting tool retains the user state of the main module when small talk module is invoked. (To know about small talk please read this [guide](https://www.gupshup.io/developer/docs/bot-platform/guide/adding-a-fallback-nlp-threshold-n-small-talk) )

Example:

**Code:**



**Result:**



From the above example you can see that the scripting tool retained the user state at “transact” even when user invoked the small talk component.

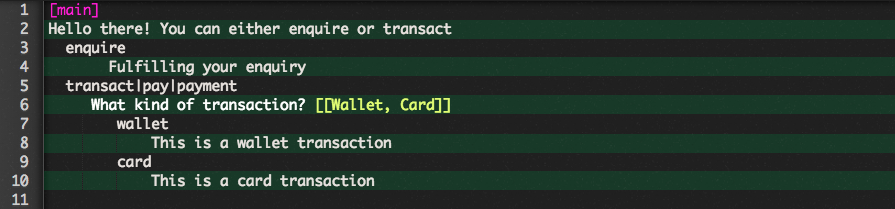
## Exception handling

The scripting tool has an auto exception handling feature. The exception that is being referred here occurs if the scripting tool doesn't find an intent mapping. There are two reasons for intent mapping exception to occur:

1. No intent mapped at a particular state of the user conversation.
2. No intent is mapped at all in the complete script.

For point 2 the scripting tool triggers the default fallback message which you can add/edit in the index.js file. Double-click the index.js file and and look at the **ScriptHandler()** method. The **options.default\_message** variable is the placeholder for the default fallback message. You can change it to your preferred message.

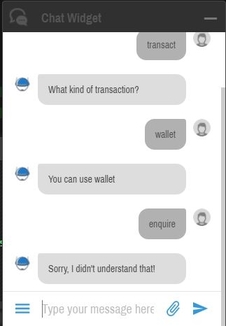
For point 1 lets look at the below example:



The “transact” user state has two child states “wallet” & “card”.

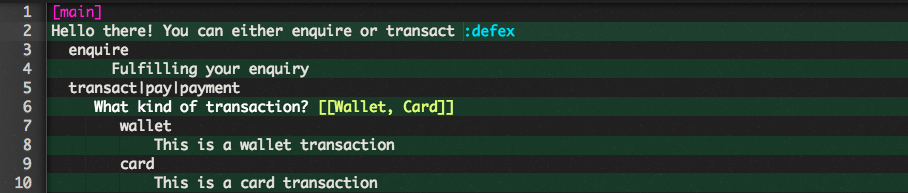
Now if a user sends a query related to transaction then the state will be set at the “transact” state because of the intent mapping found. For any query post that the scripting tool will try to do an intent mapping with the child states “wallet” or “card” and will trigger the default fallback message if the intent doesn't match.

i.e. if the user sends a query related to “enquire” it will result in an exception and the default message will be triggered because the scripting tool will try searching for “enquire” intent in the child states of “transact”.

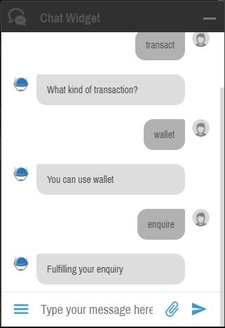


Hence you need to add the directive **:defex**at the end of a bot state that the scripting tool should restart the intent mapping process in case of the exception.

Example:

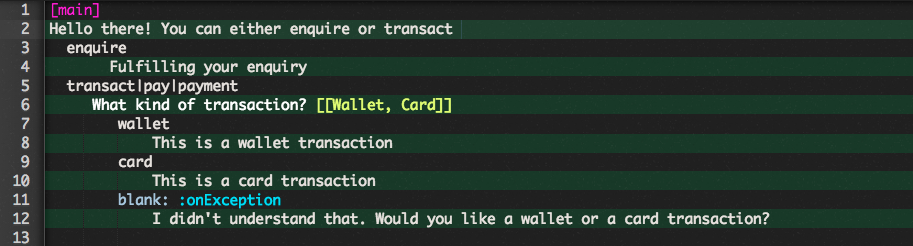


The result:



You can also choose to add a fallback within a state using the ':onException' directive. This directive denotes the state that will be executed if the NLP fails. Typically this can be used to restart a certain flow when the user has entered something unexpected.

Here's an example:



Here, 'blank:' is a label that refers to the ':onException' state.

It is important to note that you can use **either** the ':onException' directive or the ':defex' directive. The Bot Scripting tool will first check for a ':onException' fallback within the state. If it isn't found, it looks for a ':defex' to match an intent.

Here's the script in text form:

[main]

Hello there! You can either enquire or transact :defex

enquire

Fulfilling your enquiry

transact|pay|payment

What kind of transaction? [[Wallet, Card]]

wallet

This is a wallet transaction

card

This is a card transaction

blank: :onException

I didn't understand that. Would you like a wallet or a card transaction?

[common]

hi

Hello there!

who are you?

I am a chatbot

## Java SDK

Along with all the great tools we have for bot development, we now have created new Maven libraries to make it super easy for Java developers to develop bots on our platform.

Pre-requisites:

1. Maven 3.0 or above
2. JAVA 8

This can be used from any IDE which supports Maven such as Eclipse, IntelliJ IDEA, and NetBeans etc. If an IDE doesn't support Maven then you can generate bot projects using the archetypes from command line. Then edit the project in the IDE and run all other steps using command line. Check out the section below for [console commands](https://www.gupshup.io/developer/docs/bot-platform/guide/gupshup-bot-library-for-java#cm).

Let's look at the steps for using this Maven archetype plugin on Eclipse for creating a bot.

**Step 1: Installing and configuring Maven**

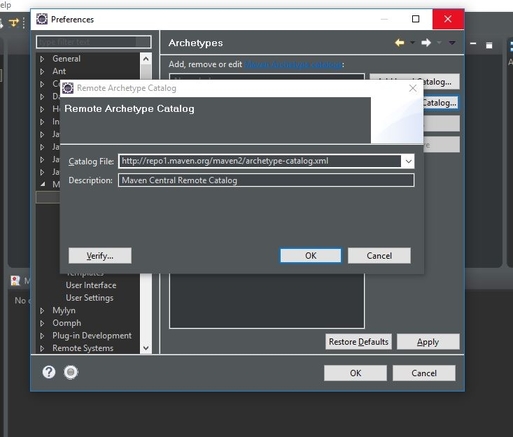
Install the Maven plugin on Eclipse and configure it. You can find information [here](https://stackoverflow.com/questions/8620127/maven-in-eclipse-step-by-step-installation).

**Step 2: Adding the archetype catalogue**

On Eclipse go to **Window->Preferences->Maven->Archetype** then click on the “Add Remote Catalog”. Once the pop up opens, fill in the details

 **Catalogue file:** http://repo1.maven.org/maven2/archetype-catalog.xml 

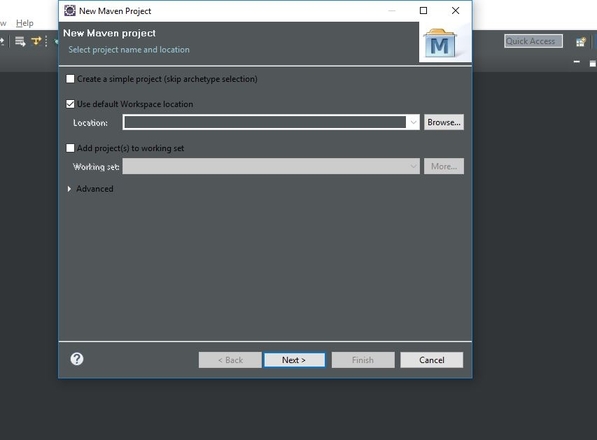
 **Description:** You can name it “Maven Central Remote Catalog” or any other name that you would prefer.



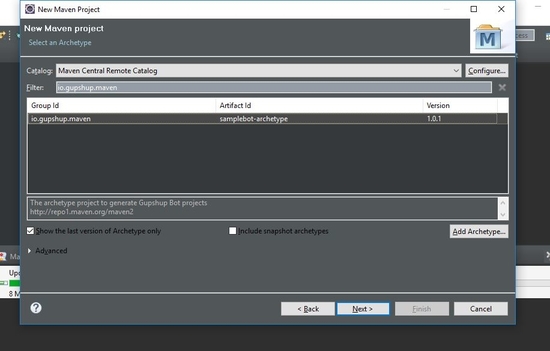
Once you apply this you can move to the next step.

**Step 3: Creating the bot project**

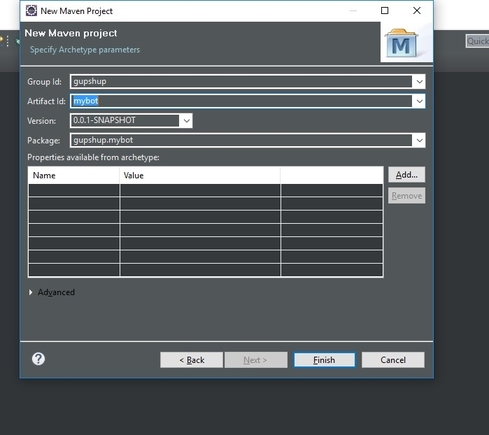
Start by creating a Maven project. You can either change the project location or click on 'Next' to move forward.



Then select the catalogue that you had configured in step 2 after which you will find the archetype displayed. Select the **samplebot-archetype**'s latest version under group id **io.gupshup.maven** and click “Next”.



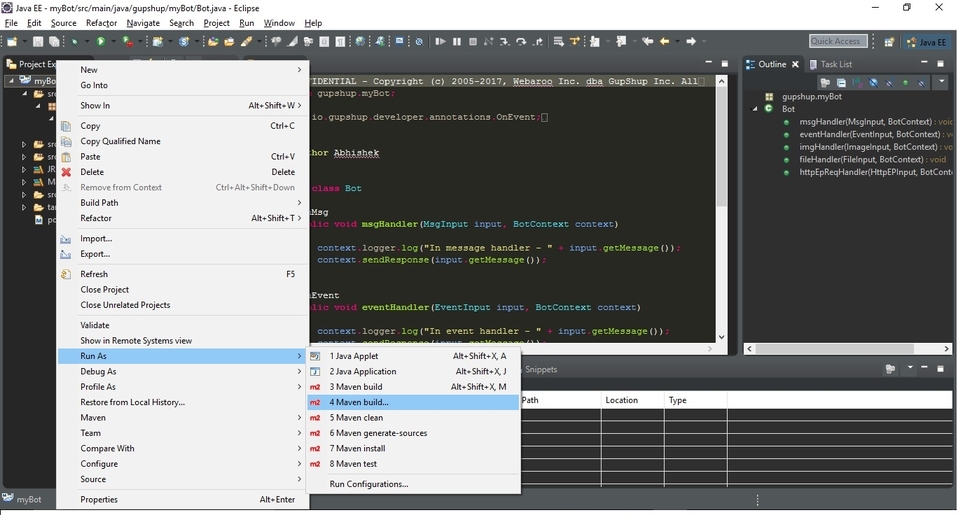
Then you will be asked to define few more names.



It will take some time to download the complete artifacts related to the project for the first time. Once done you are all set to start with your bot.

**Step 4: Creating your bot identity on Gupshup**

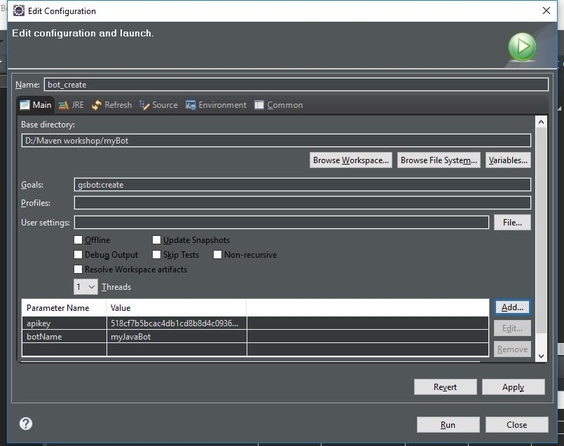
Right click on the project and go to run option then select “Maven build”



Once you get the edit configuration screen, give it a name and then in the 'Goals' field put **gsbot:create**. Then click on the 'Add' button near the parameter section to add these two parameters:

 **Name:** apiKey, **Value:** {your gupshup apikey} 

 **Name:** botName, **Value:** {any name of your choice}



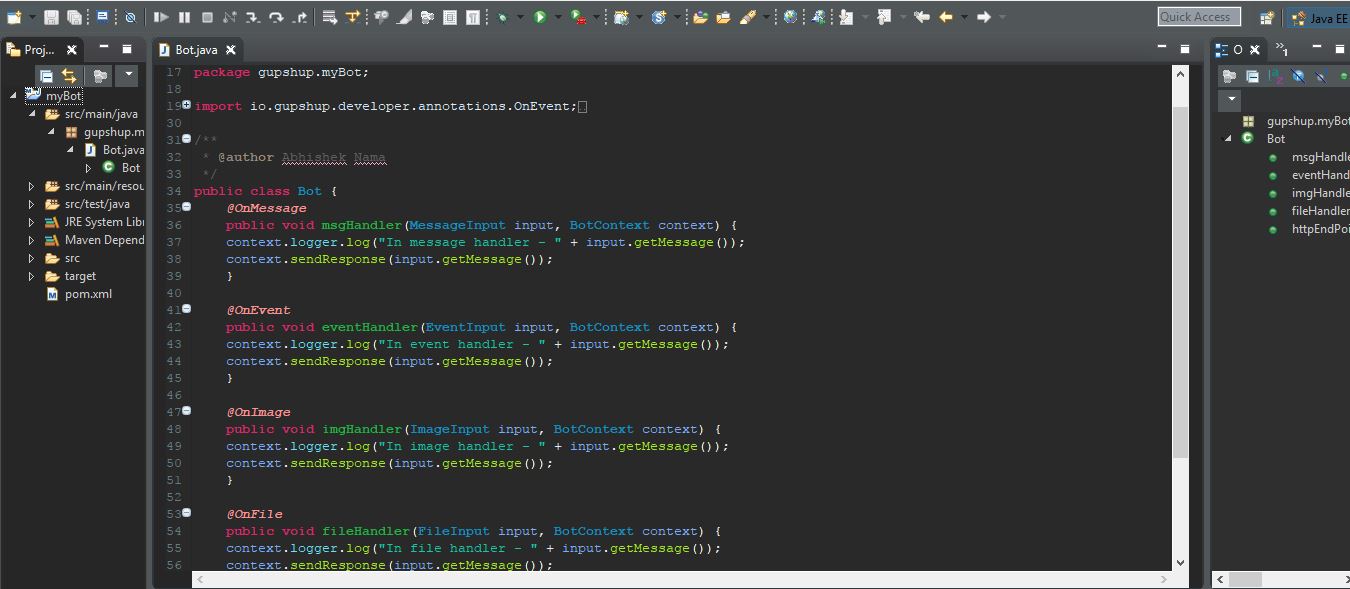
Once you have added all the details click on “Run” to create the bot. You can also validate the bot creation by going to the [dashboard](https://www.gupshup.io/developer/dashboard) on gupshup.io.

Once the bot is created the **gsbot-cfg.properties** file will be updated with the details. This file can be found at **src/main/resources/**  location in the project.

**NOTES:**  
1. You can also add an existing Gupshup bot, in that case the Goal in the configuration will change from **gsbot:create** to **gsbot:add**  
2. You might run in to an error like this  
**[ERROR] Failed to execute goal io.gupshup.maven:gsbot-maven-plugin:1.0.0:create (default-cli) on project mybot: MojoExecutionException: {"message": "Bot Name is already in use."}**  
  
To fix this go back to the run configuration and edit the parameter botName and this should go away.

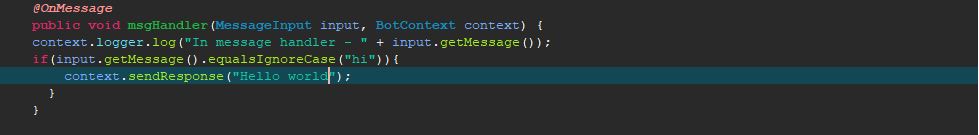
**Step 5: Coding the Bot**

Once your bot is created you can go ahead and start coding. The main bot file is located at **src/main/java/{package}/Bot.java** under your project.



You will find few pre-defined handlers that must be used. These handlers are similar to the IDE Bot Builder of Gupshup. Refer this [guide](https://www.gupshup.io/developer/docs/bot-platform/guide/intro-to-gupshup-bot-builder) section to understand more about these handlers.

The **msghandler** receives any message from the user. So let?s write a small hello world bot.

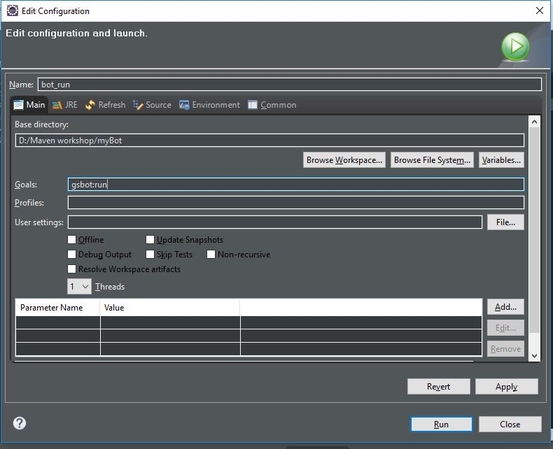


The input parameter contains all the required details like user message, user name, messaging channel etc and is similar to the IDE Bot Builder of Gupshup. You can check all the parameters returned [here](https://www.gupshup.io/developer/docs/bot-platform/guide/different-objects-sent-to-bot).

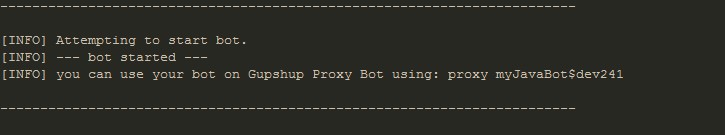
**NOTES:**  
1. If you want to change the package and class name then please go and change the same in **gupshup.bot.class** parameter in the **gsbot-cfg.properties** file as well.  
2. Please do not change the apikey in the properties file.

**Step 6: Running the bot for testing**

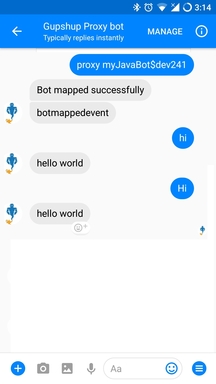
Once you are ready with your code it's time to run the bot for testing. Just like step 3 go to Maven build and name the configuration and set the Goals field as **gsbot:run**

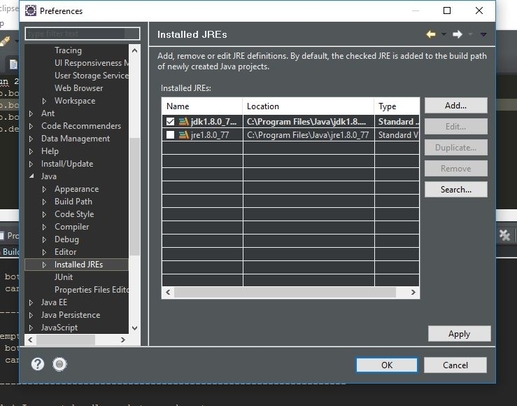


Then click the Run button to run this configuration. This will start a local server and provide you steps to test the bot and you will see this message in the console.



Go to [Gupshup proxy bot](https://www.messenger.com/t/gupshupproxybot) on Facebook messenger and use the command.



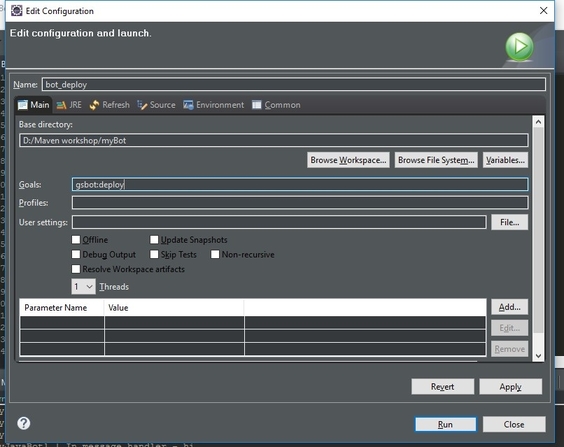
**NOTES:**  
1. If there are any DB operations in your bot, while running locally you can find the db file at following location:**{user home}/.gsbot/{bot name}.db**  
2. There are two more optional parameters to this goal, **ngrokUrl** and **port**. As the name suggests these parameters are used to set ngrok url as bot callback on specified port. The specified port will be used to start the server. Useful when 8080 port (default port) is not free.  
3. You might get the below error trying to run the bot  
**[ERROR] Failed to execute goal org.apache.maven.plugins:maven-compiler-plugin:3.1:compile (default-compile) on project mybot: Compilation failure [ERROR] No compiler is provided in this environment. Perhaps you are running on a JRE rather than a JDK?**  
To solve this go to **Window->Preferences->Java->Installed JREs** , then add the location of JDK 1.8. Check the JDK option, uncheck the JRE option and click Apply.  


**Step 7: Debugging the bot**

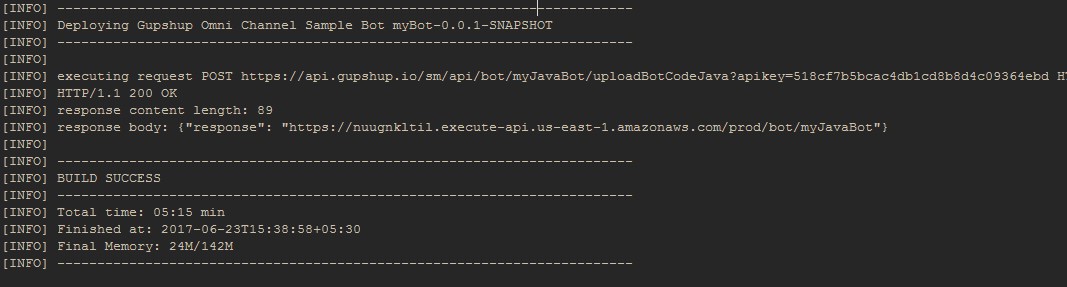
To run the bot in debugging mode, right click on the project and select **Debug As** and then **Maven Build**.   
The goals and other steps remains same as Step 5.

**Step 8: Deploying the Bot**

You should execute this step once you are done with your development and want to finally deploy your bot on production Just like step 3 go to Maven build and name the configuration and set the Goals field as **gsbot:deploy**



Then run this configuration to deploy the bot. This will take some time. This deploys the bot on AWS. A successful build will have a result like this.



Once deployed, execute the run configuration again like in step 5 and test the bot using the Gupshup proxy bot. But this time you can just run the bot using “Proxy myJavaBot” as it's deployed in production.

**NOTES:**  
1.   You might run into this error  
**[ERROR] Failed to execute goal io.gupshup.maven:gsbot-maven-plugin:1.0.0:deploy (default-cli) on project io: MojoExecutionException: build the project first -> [Help 1]**  
  
To solve this follow these two steps  
1. Stop the running server  
2. Use this command in the Maven build **clean package gsbot:deploy**

**Step 5, 6 and 8 has to be repeated for new code addition into the bot.**

#### Git Integration

This bot project comes with inbuilt git integration. This git repository is specific to Gupshup's platform and hosted by Gupshup. All the basic git features like commit, pull, push etc are available.

If you are using Eclipse, you need to have the EGit plugin installed. Checkout this [link](http://download.eclipse.org/egit/updates/) to install the plugin.

**Enabling git**:

1. Create the bot using step 4 from above.
2. Refresh your project or close and reopen the project.

**Committing changes**:   
Right click on the project then select **Teams->Commit changes**

**Adding an existing bot**:   
You can use the add bot goal to add an existing bot. There could be a case of file conflict while doing this and a warning will be displayed to you.   
You can then use the command `git checkout -f origin/master` to force the changes.

### Maven Command Reference

These are the Maven commands which you can run outside any IDE to setup the project. To execute them you need to follow the below steps.

1.   Download and install maven3.

2.   Use following command for generating bot project:

mvn archetype:generate

-DarchetypeGroupId=io.gupshup.maven

-DarchetypeArtifactId=samplebot-archetype

-DarchetypeVersion=<latest-version>

-DgroupId={my-groupid}

-DartifactId={my-artifactId}

-Dpackage={my-package}

-Dversion={version}

The new bot will be created with the sample template. It contains following files:

1.   **Bot.java:** The bot entry point. Location: src/main/java/{package}/Bot.java

2.   **gsbot-cfg.properties:** Contains bot configuration information. Location: src/main/resources/gsbot-cfg.properties

3.   **Test.java:** Can be used for local testing. Loc: src/test/java//Test.java.

**To create a bot**

mvn gsbot:create -DapiKey=<my-gupshup-apikey> -D botName=<my-bot-name>

**To add an existing bot**

mvn gsbot:add -DapiKey=<my-gupshup-apikey> -D botName=<my-bot-name>

**To run the bot locally**

mvn gsbot:run (optional parameters: -DngrokUrl=<ngrok-url> -Dport=<port-on-which-bot-server-will-run>)

**To deploy the bot**

mvn gsbot:deploy

Use "mvn clean package" to build and create package of your bot after any changes (config or code change).

## Node.js SDK

Gupshup has created a node package which will help you develop bot on Gupshup from your local environment. This package creates a replica of IDE Bot Builder in your local system allowing you to code, test and then deploy the bot on Gupshup platform.

**Pre-requisites:**

1. Node version 6 and above should be installed.

If you do not have node installed then go to this [website](http://nodejs.org/) to do so.

Now, lets look at the steps for using this SDK. 

**Step 1: Installing the supporting packages**

Create a new folder and open the terminal(command prompt) inside that folder. Then execute these two commands:

npm install -g yo

npm install -g generator-gupshup-bot

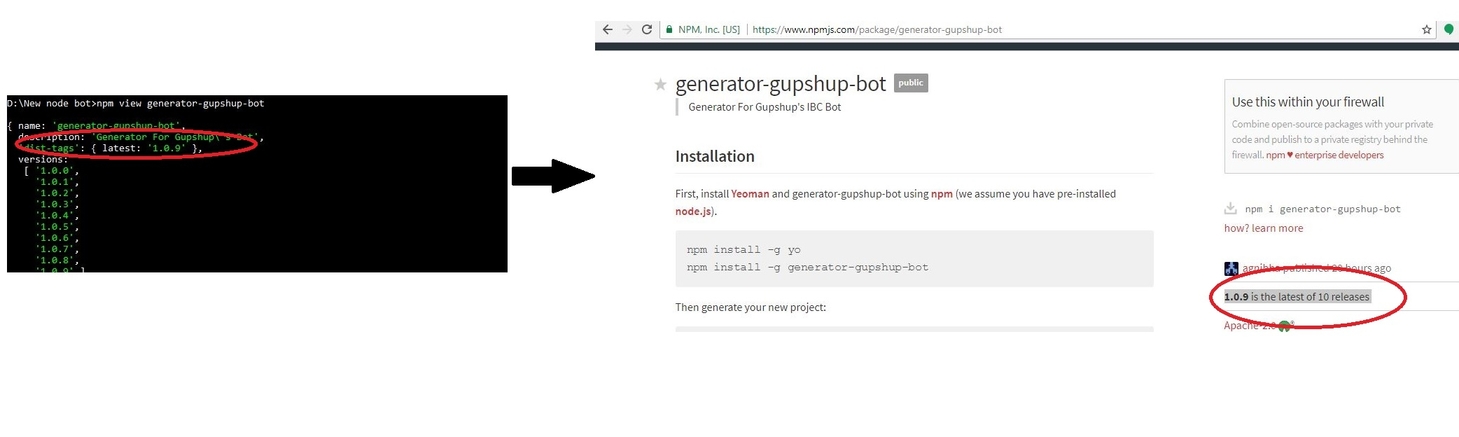
Reference: https://www.npmjs.com/package/generator-gupshup-bot

**Step 2: Verifying the installed version**

Once installed, run the below command to verify the version of **generator-gupshup-bot** package.

npm view generator-gupshup-bot

The version installed should match with the one displayed on the [package page](https://www.npmjs.com/package/generator-gupshup-bot)



For **windows** users there might be a problem where it may show a different older version hence you should uninstall and then force install the package.

To uninstall

npm uninstall generator-gupshup-bot

To force install

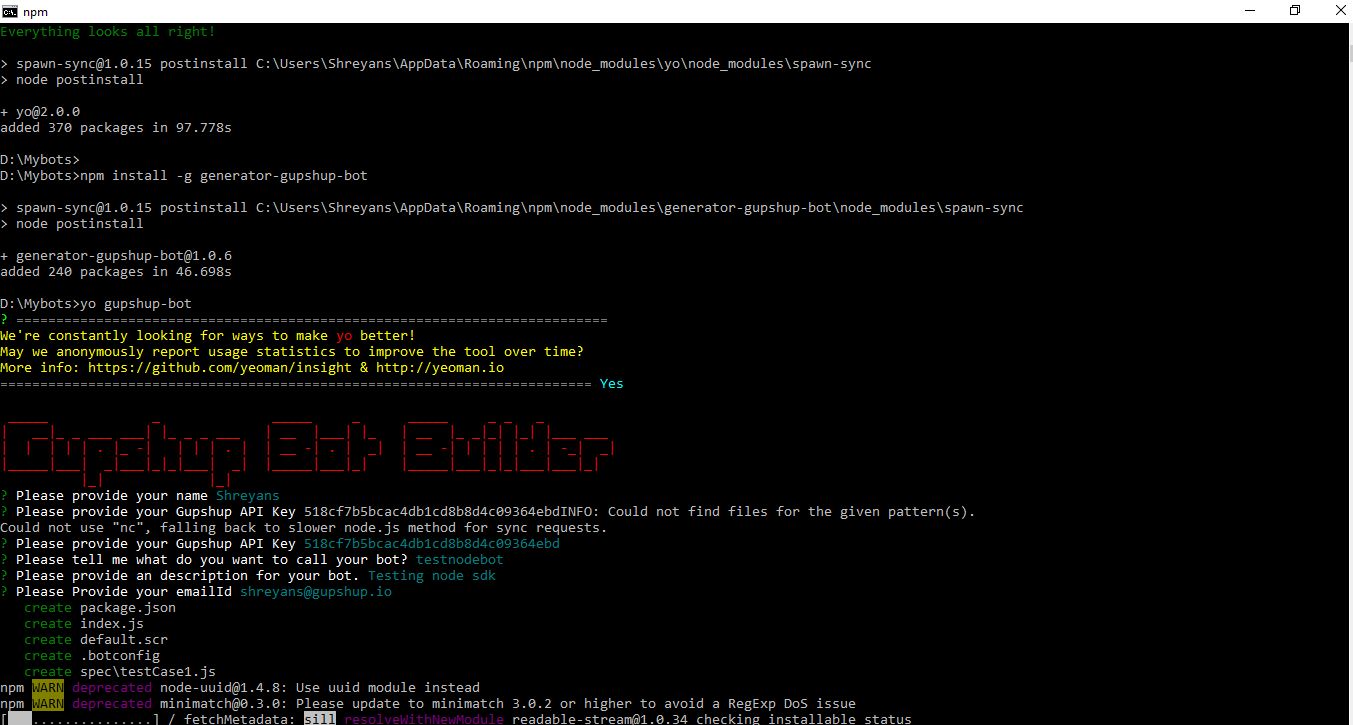
npm install generator-gupshup-bot@latest

**Step 3: Project generation**   
After the right packages are installed use the below command to generate your project.

yo gupshup-bot

When you run this command you will be asked series of inputs

1. Your name
2. Gupshup apikey(You need to login to gupshup.io to get the apikye)
3. Bot name
4. Description for the bot
5. Your email address

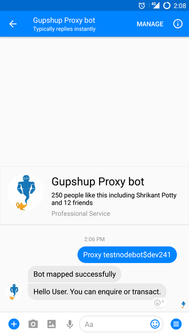


Once you have entered this information the project gets created and a bot is created on Gupshup platform. You can view the bot [dashboard](https://www.gupshup.io/developer/dashboard) to validate it.

It also starts a local server automatically using **ngrok** and provides you the command to test the bot out with the default code that comes with the project.



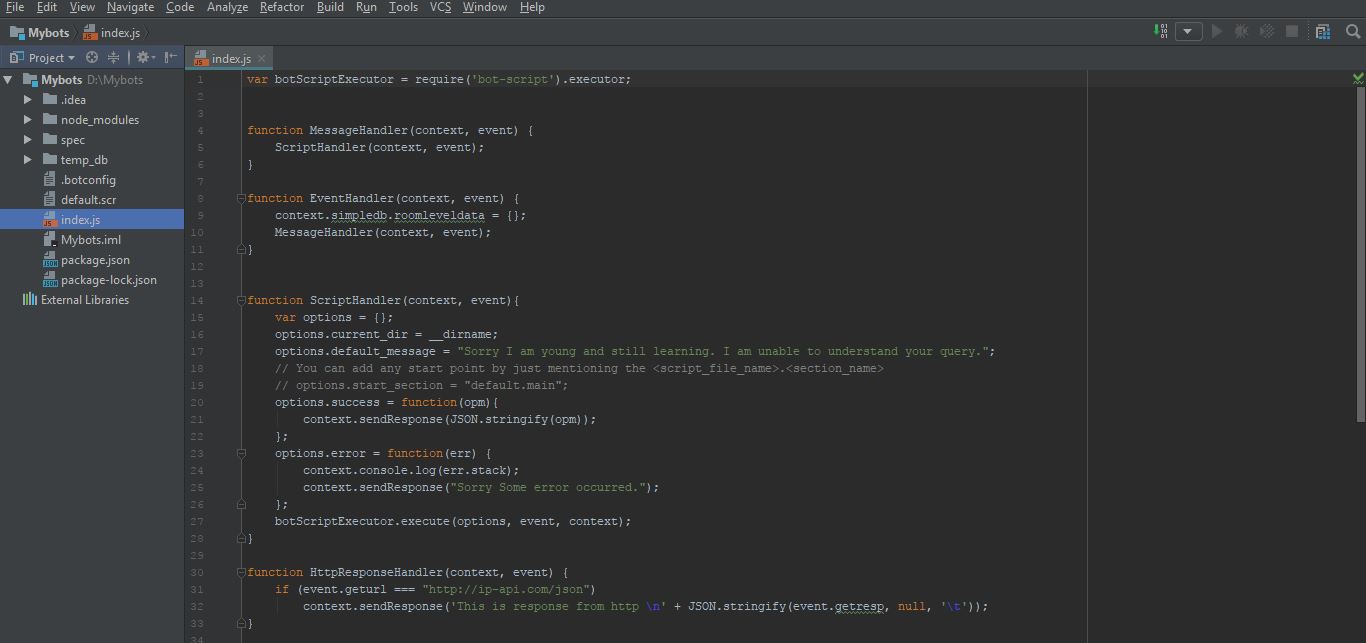
You can then go to [gupshup proxy bot](https://www.gupshup.io/developer/docs/bot-platform/guide/m.me/gupshupproxybot) on FBM and test it out.



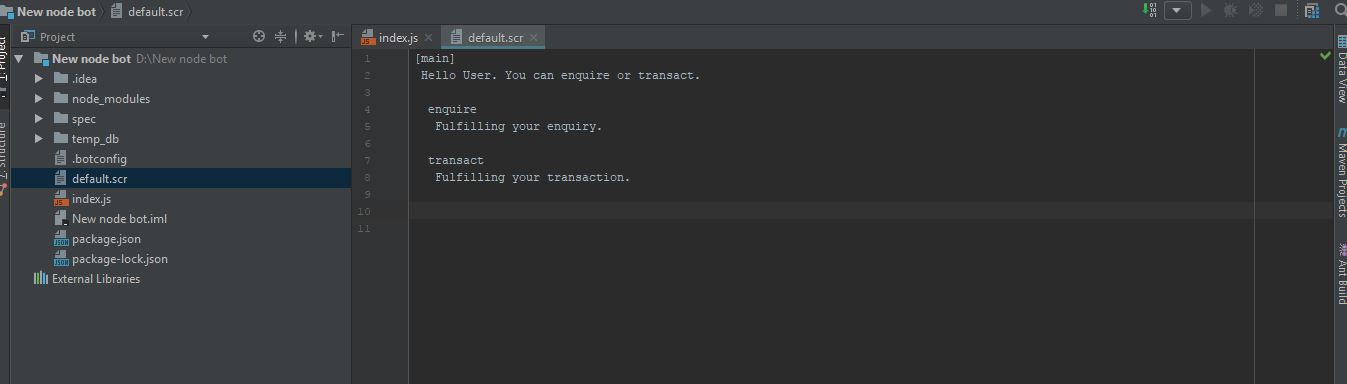
**Step 4: Changing the code**

This project is same as the Gupshup's IDE Bot Builder and supports the bot scripting tool features. Hence everything in this [guide](https://www.gupshup.io/developer/docs/bot-platform/guide/intro-to-gupshup-bot-builder) section applies to it.

The main files are **index.js** and **default.scr**.



**index.js** contains all the functions which are called when a user interacts with the bot. **default.scr** is the script file where you can write your bot script which is used by the bot.



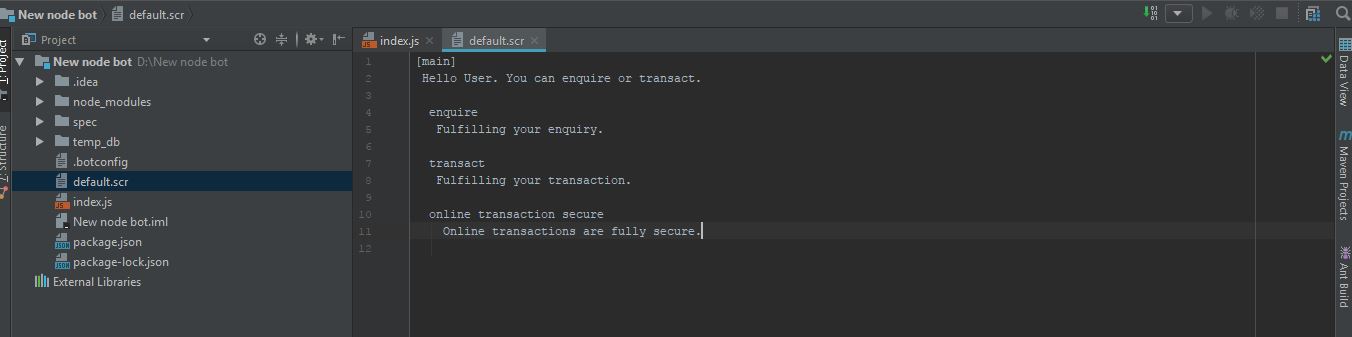
**Step 5: Testing**

The server starts running once you create the project. But after making changes you need to restart the server. Hence stop the server first by using **ctrl+c** or **cmd+c** and then run below command to start is again

npm start

This will start the local server and again provide you the command to test your bot using [gupshup proxy bot](https://www.gupshup.io/developer/docs/bot-platform/guide/m.me/gupshupproxybot).   
You have to repeat this step when you want to test out the new changes.   
Example:

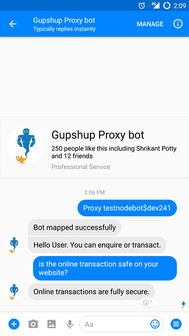
Lets make a change in the default code in **default.scr** file



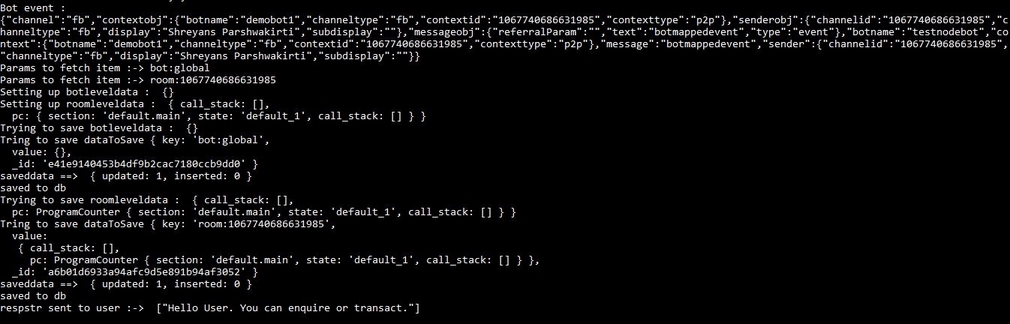
Then run the **npm start** command.



Then test the new change using the [gupshup proxy bot](https://www.gupshup.io/developer/docs/bot-platform/guide/m.me/gupshupproxybot).



You will also be able to view the logs in the terminal(command prompt)



**Step 6: Deployment**

After you have completed your bot development and testing you can deploy your bot in production using this command

npm run deploy

This will host the bot on Gupshup's AWS server in production.



**Step 7: Publishing your bot**

Once deployed on the production you can go to Gupshup's [dashboard](https://www.gupshup.io/developer/dashboard)and publish your bot on the desired channels using the publish tab.

# Serverless WebViews using Gupshup

Suppose you have a conversational shopping bot that allows your users to buy t-shirts. A typical user would click on the bot, choose size, color, design and other details. The bot then asks your user about delivery date and location and so on. Just before confirming the order, your user changes his mind and wants to select a different color. The user has to then cancel the current flow and restart the entire process via conversation again. This is cumbersome and highlights the a problem with having wholly conversational bots: Text isn't the best option when there are multiple inputs. Having [structured messages](https://www.gupshup.io/developer/docs/bot-platform/guide/structured-messages) to get each input doesn't solve the problem of having to re-enter data.

To help bot makers improve the experience for their users, Gupshup has developed Serverless WebViews which are webviews that you can customize and embed within your chatbot to collect data from the user. These Webviews do not explicitly need to be hosted on any server (hence the name “Serverless”) and they serve two main purposes:

1. Entry of multiple input fields
2. Provide a richer browsing experience

This guide will illustrate how you can create a custom Serverless WebView using the Gupshup Form Builder API and embed it within your chatbot. You can embed the webview within a chatbot on Facebook Messenger. These are the input fields that you can use in a Serverless WebView:

* Text box
* Drop down
* Radio button
* Check box
* Date
* Time
* TextArea

### Gupshup Form Builder API

First you'll have to create a form using the Gupshup Form Builder API.

Go to gupshup.io and click on the API tab and click on the 'Form Builder' section. You will use the /facebook/smartmsg/form/create API to build a form. This is a POST request and to create a form you will have to specify the fields of the form in a JSON.

Here's a description of the JSON:

### Serverless webviews using Facebook API

|  |  |  |  |
| --- | --- | --- | --- |
| **Property Name** | **Description** | **Type** | **Required** |
| title | Form title | String | Y |
| autoClose | Enable Facebook's autoClose feature. If 'true' the form closes as soon as the 'Submit' button is pressed. | Boolean | N |
| message | On form submit given message value will be displayed. | String | N, default is '**Success!** Your reply has been noted.' |
| callback-url | URL to pass the user input when submit button is clicked. If you'd like the user input data to be passed back to the bot use: https://www.gupshup.io/developer/bot/{botName}/public. Read [this](https://www.gupshup.io/developer/docs/bot-platform/guide/http-call-to-a-bot) for more info | String | Y |
| fields | 1. Type - The type of input element. Types supported are:  -**input** - Text Box - **radio** - Radio Button - **select** - Drop Down - **checkbox** - Check Box - **date** - Add a date picker. Date format MM/DD/YYYY - **Time** - Add a time field. Time format HH:mm (12 hours) - **textarea** - Text Area. -**Fbid**- Add this to enable the 'Add to Messenger' extension  Minimum 1 element is required in fields   2. Name: The name of the field   3. Label: The display name on the form   4. Validations: You can add an array of regex validations and corresponding messages. Format is:  - **regex**- Add a regular expression to validate the field  - **msg** - Message displayed when input is erroneous | String | Y |
| users | User ID or any unique value for user form.  The response contains unique forms for each user specified. | String | Y |

Here's a sample JSON:

{

"title": "This is a test.",

"autoClose": false,

"message": "Thank You",

"callback-url": "https://www.gupshup.io/developer/bot/{botName}/public",

"fields": [{

"type": "fbid",

"name": "fbname",

"label": "fbName"

}, {

"type": "input",

"name": "name",

"label": "Name",

"validations": [{

"regex": "^[A-Z a-z]+$",

"msg": "Only alphabets are allowed in this field"

}, {

"regex": "^[A-Z a-z]{6,}$",

"msg": "Minimum 6 characters required"

}]

}, {

"type": "radio",

"name": "gender",

"label": "Gender",

"options": [

"Male",

"Female"

],

"validations": [{

"regex": "",

"msg": ""

}]

}, {

"type": "select",

"name": "account",

"label": "AccountType",

"options": [

"current",

"savings"

],

"validations": [{

"regex": "",

"msg": ""

}]

}, {

"type": "checkbox",

"name": "interest",

"label": "Interests",

"options": [

"Cooking",

"Reading"

],

"validations": [{

"regex": "",

"msg": ""

}]

}],

"users": [

"Testing"

]

}

You can copy the sample JSON replacing {botName} with the name of your bot, in the callback URL field and use the Form Builder API to create a new form. You can also create your custom form JSON. If all goes well you will get a sample response like this:

[

{

"embedlink": "https://smapi.gupshup.io/sm/api/facebook/smartmsg/embed/786d48b1-xxx-4ebf-9fe1-998393fadab4",

"expired": false,

"fb-button": {

"messenger\_extensions": true,

"title": "This is a test.",

"type": "web\_url",

"url": "https://smapi.gupshup.io/sm/api/facebook/smartmsg/embed/786d48b1-xxxx-4ebf-9fe1-998393fadab4",

"webview\_height\_ratio": "tall"

},

"id": "786d48b1-xxxx-4ebf-9fe1-998393fadab4",

"signed-for": {

"display": "Testing",

"subdisplay": "Testing"

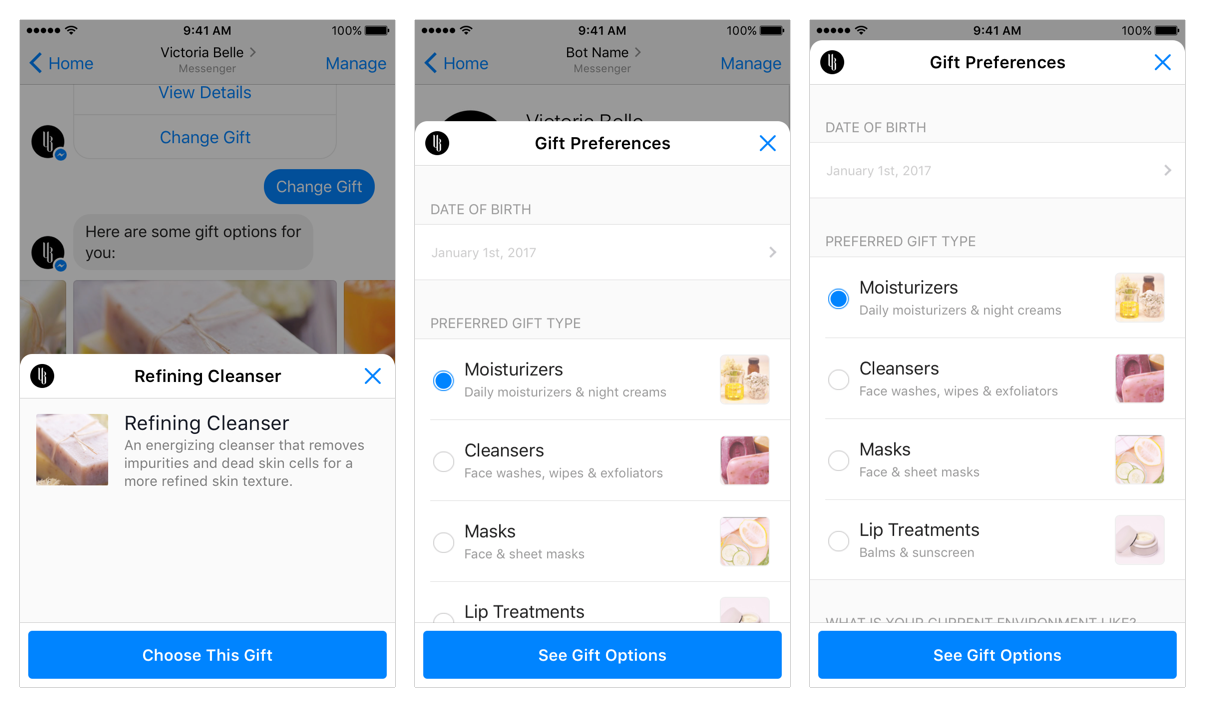
},

"smid": "4436"

}

]

Here 'embedlink' contains the link to the form that is created.   
Webview\_height\_ratio refers to the [size of the webview](https://developers.facebook.com/docs/messenger-platform/send-api-reference/url-button). It comes in three sizes - compact, tall and full.



(image courtesy Facebook Messenger documentation)

### Embedding the form

In our example above, we have specified the endpoint of the bot as the callback URL. For more details about handling the HTTP endpoints of a bot read [this](https://www.gupshup.io/developer/docs/bot-platform/guide/http-call-to-a-bot). Thus the user's response is handled in the HttpEndpointHandler() method. The user input is stored in the event.params.payload parameter. Use this parameter to extract out all the fields of the form that you have created and store it into a database.

In your HttpEndpointHandler() add this code snippet:

function HttpEndpointHandler(context, event) {

formId = event.params.linkId;

var userDetails = event.params.payload;

for(var i=0; i < userDetails.length; i++)

{

if(userDetails[i]["fieldname"]=='name')

{

name = userDetails[i]["fieldvalue"];

continue;

}else if(userDetails[i]["fieldname"]=='account')

{

account = userDetails[i]["fieldvalue"];

continue;

}else if(userDetails[i]["fieldname"]=='gender')

{

gender = userDetails[i]["fieldvalue"];

continue;

}else

{

interests = userDetails[i]["fieldvalue"];

}

}

context.simpledb.doGet(formId);

}

Where 'name', 'account' and 'gender' correspond to the 'name' field of the JSON that we sent as part of the POST request earlier. We use the doGet method to query the database against the 'formId' of the form that was introduced earlier.

Do note that you cannot use a context.sendResponse within the HttpEndpointHandler() method. This is because the sendResponse will be sent to the Serverless Webview created and not the Messenger session that is in use.

In your DbGetHandler() method, we will construct a message using the input from the form. We will then use the Gupshup Send Message API to send this message back to the user.

function DbGetHandler(context, event) {

var done = null;

var apikey = '<API KEY>';

botname = encodeURI(event.botname);

contextobj = event.dbval;

apikey = encodeURI(apikey);

message = encodeURIComponent("Your Details are:\n" +

"Name: "+ name + "\n" +

"Gender: "+ gender + "\n" +

"Account Type: "+ account + "\n" +

"Interests: "+interests);

var url = "https://api.gupshup.io/sm/api/bot/" + botname + "/msg";

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

var headers = "{\"Accept\": \"application/json\",\"apikey\": \"" + apikey + "\",\"Content-Type\": \"application/x-www-form-urlencoded\"}";

context.simplehttp.makePost(url, body, JSON.parse(headers), done);

}

In the above example, the callback URL was the bot itself i.e the user input was sent back to the bot code hosted on Gupshup. You can chose to have an external callback as well. For example, we can use a Request Bin as a callback URL. Just modify the JSON that you are sending as part of the POST call.

{

"title": "This is a test.",

"autoClose": false,

"message": "Thank You",

"callback-url": "requestb.in/1i5oasr1?,

"fields": [{

"type": "fbid",

"name": "fbname",

"label": "fbName"

}, {

"type": "input",

"name": "name",

"label": "Name",

"validations": [{

"regex": "^[A-Z a-z]+$",

"msg": "Only alphabets are allowed in this field"

}, {

"regex": "^[A-Z a-z]{6,}$",

"msg": "Minimum 6 characters required"

}]

}, {

"type": "radio",

"name": "gender",

"label": "Gender",

"options": [

"Male",

"Female"

],

"validations": [{

"regex": "",

"msg": ""

}]

}, {

"type": "select",

"name": "account",

"label": "AccountType",

"options": [

"current",

"savings"

],

"validations": [{

"regex": "",

"msg": ""

}]

}, {

"type": "checkbox",

"name": "interest",

"label": "Interests",

"options": [

"Cooking",

"Reading"

],

"validations": [{

"regex": "",

"msg": ""

}]

}],

"users": [

"Testing"

]

}

The sample URL for a callback looks like this:

{

"linkId": "f42414e2-ce1a-xxx-b40a-e88e4d4d9aee",

"payload": [{

"fieldname": "name",

"fieldvalue": "Alice"

},{

"fieldname": "gender",

"fieldvalue": "Male"

},{

"fieldname": "account",

"fieldvalue": "savings"

},{

"fieldname": "interest",

"fieldvalue": "Cooking"

}],

"time": 1479904354249,

"userid": "UserID"

}

### Messenger Extensions

Like the name suggests, Facebook Messenger Extensions allow you to add functionality to your user's Messenger experience. With Extensions you can get information on the user, accept payments, auto-close webViews, make your bot viral with sharing, and deeply integrate with Messenger's UI.

To use Messenger Extensions in your bot, you must first whitelist the domain the webView is served from. Gupshup has an API to help you do this:

POST /sm/api/facebook/smartmsg/messenger/extension/enable

pageAccessToken= {FB page access token}

You can view this API [here](https://www.gupshup.io/developer/ent-apis). The pageAccessToken can be obtained only once a FB App and FB page has been created.

### Enabling auto-close

Facebook has a feature which closes the webView after a transaction is complete or the form is filled. To enable this, first whitelist the domain using the API mentioned above.   
Then in the JSON used to construct your form, change the 'autoClose' parameter to 'true'.