Facebook Messenger Platform

Exercises

Introduction: According to Fig. 1, we require two parts for our chatbot: a conversation/canvas channel for connecting with users and audiences. and on the other hand, an artificial intelligence (AI)/natural language processing (NLP) for making our chatbot more intelligent and practical.

These services allow us using natural language understanding (NLU), dialogue managers and natural language understanding (NLG) methods. In general, we are interested in designing or leveraging an existed platform because of the following reasons:

- It is easy to use NLP tools and define our workflows and answers.
- We can add files, figures and emojies to our chatbots easily.
- Different buttons and sliders should be easily designed. Chatbot designing should create an omnichannel experience too.
- It is easy to train the chatbot and change settings of chatbot.
- It is easy to add a chatbot to our website.
- It is easy to integrate chatbot to an analytic tool.
- It increases the conversations with the clients.

  Among existed list of conversation channels, we will use the **Facebook messenger platform** (a toolbox for building bots)[1] and **wit.ai** as a platform for chatbot development. Wit.ai[2] is a free chatbot software

[1]The other existed platforms and apps are: Line, Skype, Telegram, Wechat and Slack

[2]The other existed platforms for chatbot developments are: Chatfuel, Botsify, Flow Xo, KITT.AI, IBM Watson, Reply.ai, ManyChat

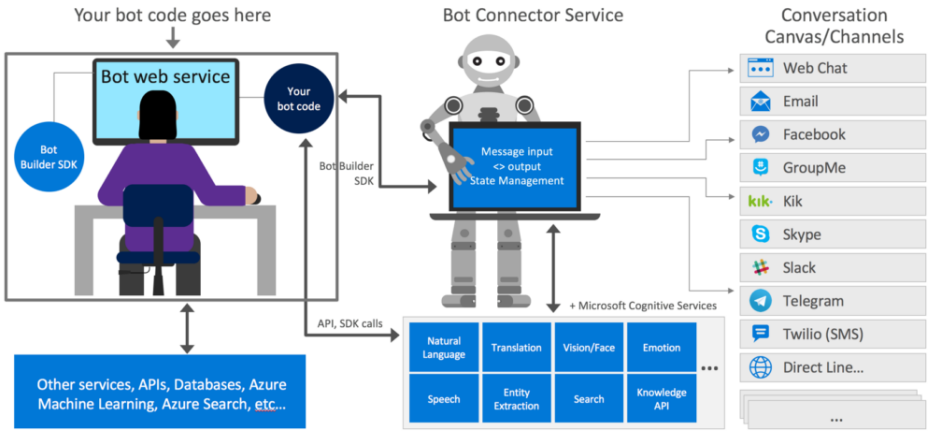


Figure 1: Chatbot in connection with the conversation/canvas channels and NLP/AI services

that lets you easily create text or voice based bots on your preferred messaging platform. Wit.ai learns human language from every interaction, and leverages the community: what is learned is shared across developers.

For Implementing a chatbot with a conversation channel and an AI api, we should follow the following workflow:

Create a Facebook page → create a Facebook app → add the messenger platform to the App → use a service[3] → link the chatbot app to the messenger platform (using webhooks[4]) → train and test → submit the chatbot

Let's start with the Facebook messenger platform.

Exe 1 Read this link `https://techcrunch.com/2016/04/12/agents-on-messenger/`, as an introduction on Facebook messenger platform.

Exe 2 Similar to the Vanilla project, design the interactive terminal interface with nodejs. Open a terminal in your computer. Make a directory

[3]for us it is wit.ai

[4]It will be explained later

namely movieBot[5]. Go to the related directory and run the following command:

```
> npm init
```

This command creates a package.json file for your app. Create another file in your folder namely app.js. Check your package.json file and change its "main" parameter to app.js

Exe 3 Similar to the vanilla project test your code using the following command line[6]:

```
node app.js
```

Define the app globally using:

```
nodemon app.js
```

Exe 4 **Connect our chatbot to the facebook messenger using a server** It means our chatbot connects to a server which enables it to be connected to the facebook messenger:

our chatbot $\leftrightarrow$ server (ex. Express) $\leftrightarrow$ Facebook messenger

Notice An existed server for Node.js is **Express**. Express is a web application framework for Node.js that allows you to spin up robust APIs and web servers in a much easier and cleaner way. It is a lightweight package that does not obscure the core Node.js features.

Install Express via npm such as using the following command line[7]

```
npm install express --save
```

check the movieBot/package.json file. You see that express has been added as a new dependency. In the movieBot folder make a js file namely server.js. Afterward change your main attribute of the package.json to

```
"main": "server.js",
```

[5]At the moment choose this name for your bot. We will explain later what is this chatbot and why this name.

[6]This part is just a reminder of what you have done in the Vanilla project

[7]For installing packages and software, google the installation. Because each computer has different characteristics.

this allows us to put the chatbot in contact with the messenger platform.

Exe 5 server.js: we should code the server.js to design an API for sending and getting messages to the facebook messenger. Now that Express is installed, here's what the most basic server looks like:

```
1 'use strict';
2
3 const express = require('express');
4
5 const server = express();
6 const PORT = process.env.PORT || 3000;
7
8 server.get('/', (req, res) => res.send("hello!!"));
9 server.listen(PORT, () => console.log(`The bot server is
     running on port ${PORT}`));
```

run this script with the nodemon from the inside movieBot folder and navigate to `localhost:3000` in your browser. You should see the message hello!! in your browser window.

Notice Node.js: In the previous example, line 3 is grabbing the main Express module from the package you installed. This module is a function, which we then run on the line 5 to create our server variable. You can create multiple apps this way, each with their own requests and responses.

The code in line 8 is where we tell our Express server how to handle a GET request to our server. Express includes similar functions for POST, PUT, etc. using server.post(...), server.put(...), etc.

These functions take two main parameters. The first is the URL for this function to act upon. In this case, we are targeting '/', which is the root of our website: in this case, localhost:3000.

The second parameter is a function with two arguments: `req`, and `res`. `req` represents the request that was sent to the server; We can use this object to read data about what the client is requesting to do. res represents the response that we will send back to the client. Here, we are calling a function on res to send back a response: 'hello!!'.

Finally, once we've set up our requests, we must start our server. So `process.env.PORT || 3000` means: whatever is in the environment variable `PORT`, or `3000` if there's nothing there. We are passing the PORT variable into the listen function, which tells the server which port to listen on. The

function passed-in as the second parameter is optional, and runs when the server starts up. This just gives us some feedback in the console to know that our application is running.

And there we have it, a basic web server. However, we definitely want to send more than just a single line of text back to the client.

Notice Node.js: The `process.env` global variable is injected by the Node at runtime for your application to use and it represents the state of the system environment your application is in when it starts. For example, if the system has a `PATH` variable set, this will be made accessible to you through `process.env.PATH` which you can use to check where binaries are located and make external calls to them if required.

Exe 6 create a Facebook app: Go to `https://developers.facebook.com/`. After signing in with your FB account create a new app id for instance namely Movie chatbot. After creating the app, you will acces to several api and services supported by Facebook. For our part, click on the **setup button for the Facebook messenger**[8]. You will be directed to the setting dashboard for your chatbot.

Our chatbot requires a unique code for communicating with the Facebook messenger. The page access token is the unique code that our chatbot uses for sending back every request that it makes to the messenger platform api. [9] To get a page access token, you should select a page assigned to your chatbot. For this example either choose an existed page of your FB account or make a new page FB for your chatbot.

The generated token will NOT be saved in the user interface. Each time you select a Page from the drop-down, a new token will be generated. If a new token is generated, previously created tokens will continue to function.

Before counting to the rest of this course, we need to know about the **webhook**.

[8] it is just close to the messenger logo among the list of existed services in the page

[9] Graph API requires Page access tokens to manage Facebook Pages. They are unique to each Page, admin, and app and have an expiration time. You must own or have a role on the Page to get a Page access token.

WebHook: Webhook is a user-defined HTTP callback which retrieves and stores data from an event, usually from outside of your software application. As opposed to an API which requires you to be constantly polling, webhooks will let you know when information has been received, saving you valuable time. In really simple terms a Webhook is a messenger between your chatbot and a source of information.

How do you install a Webhook? A Webhook is essentially a POST request which is sent to a URL. You will register an http:// or https:// URL which will store this POST request in JSON or XML format before processing it.

notice: JSON is short for JavaScript Object Notation, and is a way to store information in an organized, easy-to-access manner. In a nutshell, it gives us a human-readable collection of data that we can access in a really logical manner.

As a simple example, information about an imaginary person can be written in a example.json file as follows:

```
{
  {
    "name": "Walter White"
    "age" : "50",
    "hometown" : "New Mexico, US",
    "gender" : "male"
  }
}
```

This creates an object. By enclosing the values in curly braces, we're indicating that the value is an object. Inside the object, we can declare any number of properties using a "name": "value" pairing, separated by commas.

How is a Webhook different from APIs? In an Application Programming Interface (API), the user must make a direct request to the program for a response. For example, if you create an application which is built on top of Facebook, the user would need a Facebook account to use it. When your application is downloaded the burden is on you to ask the customer to sign in with their Facebook account or create an account.

What can the Webhook be used for? Webhooks are incredibly flexible and can be used for many functions such as:

. Placing orders

. Changing prices

. Invoices

. Payments

. Collecting data for data-warehousing

. Integrating accounting software Disputes

. Removing customer data from your database when they end their subscription or uninstall an app

. Filtering order items and informing shippers about the order

Exe 7 manage tokens To register our movieBot on the messenger platform we should configure the access tokens to our bot. First create a folder namely 'config' in movieBot folder and then make a 'development.json' file in the new folder. This file will hold the tokens while we develop the app.

It is recommended that you keep sensitive information like your page access token secure by not hardcoding it into your webhook. To do this, code the development.json as the following:

```
1
2 {
3   "FB": {
4     "pageAccessToken": "page-access-token",
5     "VerifyToken": "verify-token"
6   }
7 }
```

`page-access-token` should be replaced by your page access token coming from the previous exercise.

We create a verify token as a random string of our choosing, hardcoded into the webhook. We will provide the verify token to the Messenger Platform when we subscribe our webhook to receive webhook events for an app[10].

[10]Necessity of defining this random string will be clear in the rest of this course.

Now `verify-token` should be replaced with a random string. To do so, generate a random verify token using the `randomBytes` in java script. Copy your random verify key and paste it in the code.

——— *Try to find a solution by your self* ———

If you can not find any solution try the following code directly in a terminal. :

```
$ node
> crypto.randomBytes(64).toString('hex')
```

`randomBytes(64)` gets 64 random bytes. This number can be different according to your preferences.

Exe 8 For the sake of the security, create an index.js in the config folder for accessing the tokens from the development.json file. As writing this code is not obvious, the code is given completely here:

```
'use strict';

if(process.env.NODE_ENV === 'production') {
  module.exports = {
    FB: {
      pageAccessToken: process.env.pageAccessToken,
      VerifyToken: process.env.VerifyToken
    }
  }
} else {
  module.exports = require('./development.json');
}
```

Question Import the ./config in the server.js file to make the tokens accessible for your chatbot.

Notice Node.js: Working with environment variables is a great way to configure different aspects of your Node.js application. Many cloud hosts (Heroku, Azure, AWS, now.sh, etc.) and Node.js modules use environment variables. Hosts, for example, will set a PORT variable that specifies on which port the server should listen to properly work. Modules might have different behaviors depending on the value of `NODE_ENV` variable.

According to Fig. 2, we require to define a webhook for receiving messages and callbacks from the FB messenger in our chatbot and sending back new callbacks and messages to the FB messenger.

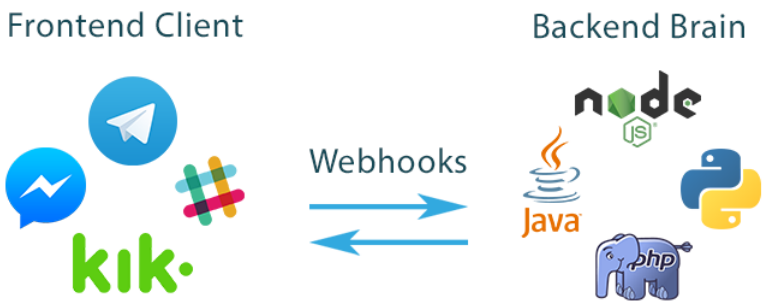


Figure 2: Frontend and backend communicate with the Webhook.

Exe 9 register webhook: For defining a webhook, create a folder namely fbeamer in the movieBot folder. Create an index.js file inside the new folder as well. To define the webhook as a reusable model, define a class namely 'FBeamer' with a constructor containing a parameter containing pageAccessToken, VerifyToken[11]. Notice to verify if the two parameters are given or not. Adding a try catch is appreciated too. Complete the class constructor.

```
'use strict'

class FBeamer{

  constructor({pageAccessToken, VerifyToken}){

    //complete the constructor here

  }

}

module.exports = FBeamer;
```

Notice Node.js: To understand the class definition in javascript, here we present two examples. The class syntax in javascript has two forms:

```
function User(name) {
  this.name = name;
```

[11]Notice that the input has the same format as the FB of the development.json file.

```
}

User.prototype.sayHi = function() {
  alert(this.name);
}

let user = new User("John");
user.sayHi();
```

And that's the same using class syntax:

```
class User {

  constructor(name) {
    this.name = name;
  }

  sayHi() {
    alert(this.name);
  }
}

let user = new User("John");
user.sayHi();
```

The two examples are alike. Just note that methods in a class do not have a comma between them. Novice developers sometimes forget it and put a comma between class methods.

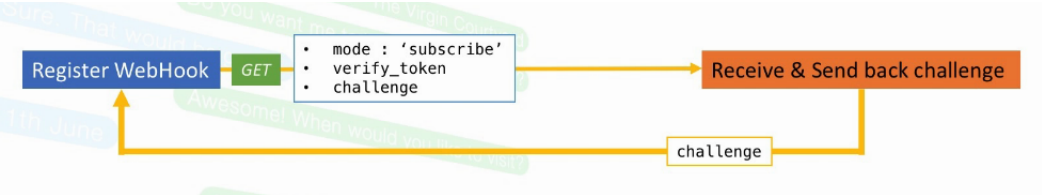
The `class User ...` here actually does two things:

- Declares a variable `User` that references the function named "constructor".
- Puts methods listed in the definition into `User.prototype`. Here, it includes `sayHi` and the `constructor`.

Exe 10 Import the fbeamer folder in the server.js and get the fb parameter by initiating of the FBeamer class. *hint:* you can get the FB of the development.json by importing the `./config` as conf and using `conf.FB`.

WATCH THIS SHORT VIDEO ON THE WEBHOOK:
`https://www.youtube.com/watch?time_continue=6&v=wUPGUD9_41w&feature=emb_title`

Exe 11 `registerHook` function (to be defined later), is a function that allows



us to register the webhook for our chatbot. For registering a webhook for our chatbot, we have to provide Facebook messenger with a secure url. After registering the webhook, Facebook sends an object to our chatbot. Facebook makes a GET call to the chatbot and sends an **object** to our chatbot 'see the above figure):

```
mode : 'subscribe'
verify_token
challenge
```

According to the following diagram, facebook sends an object to the chatbot by calling the GET function and finally the chatbot returns back the challenge to the facebook.

Facebook Messenger
webhook $\longrightarrow$ GET $\longrightarrow$ mode : 'subscribe', verify_token, challenge $\longrightarrow$ receive and send back challenge for the chatbot $\longrightarrow$ chatbot

Facebook messenger $\longleftarrow$ challenge $\longleftarrow$ chatbot

There is a key calling hub containing the object for the process as :

```
{
  hub:
   {
  mode : 'subscribe'
  verify_token : <token example>
  challenge: <challenge example>
  }
}
```

First modify the get function in server.js by calling the `registerHook` function (attention the function has not been implemented yet.):

```
const config = require('./config');
const f = new FBeamer(config.FB);

```

```
4 server.get('/', (req, res)) => f.registerHook(req, res));
```

Here the request (`req`) refers to the object coming from facebook and response (`res`) is referred to the response sending from the chatbot to facebook.

Once the chatbot receives the verify_token, it should checks if this token is the same as the one that we have defined in our setting and it will send back the received challenge code to the facebook.

Exe 12 verify webhook Write a function (registerHook) in fbeamer/index.js, FBeamer class to verify if the webhook has been registered or not. Complete the commented parts of the code by yourself.

```
1
2
3  registerHook(req, res) {
4    const params = req.query;
5    const mode = params['hub.mode'],
6    token = params['hub.verify_token'],
7    challenge = params['hub.challenge'];
8      // if mode === 'subscribe' and token ===
          verifytoken, then send back challenge
9      try {
10       if () { //condition should be written in the
            parentheses
11         //print here that webhook is registered
12         return res.send(challenge);
13       } else {
14         console.log("Could not register webhook!");
15         return res.sendStatus(200);
16       }
17     } catch(e) {
18       console.log(e);
19     }
20 }
```

Notice that mode, token and challenge are the the webhook parameters and they are retrievable using the code written in lines 7 to 10. The parameter descriptions are as follow:

parameter	sample value	description

hub.mode subscribe This value is always set to subscribe.

hub.challenge 1158201444 An int must pass back to FBM[12].

hub.verify_token meatyhamhock A string that FBM grab from the Verify Token field in your app's App Dashboard. You will set this string when you complete the Webhooks configuration settings steps.

For better understanding of webhooks read:
`https://developers.facebook.com/docs/messenger-platform/webhook/`

***Notice**: to continue with the following exercise, check your Internet connection. Some secured wifi connections may block the ngrok.*

Exe 13 make webhook accessible using an HTTPS connection: In order to make the webhook (as a connection between FBM and our chatbot) accessible, we need an HTTPS. The HTTPS is located between FBM and the webhook as:
FB messenger ↔ HTTPS ↔ webhook ↔ chatbot

In fact this unique and https, recieve and tunnel data with our local running node.js app. For generating a private https, you can use one of the following services based on your operating system (windows, linux or mac)[13]

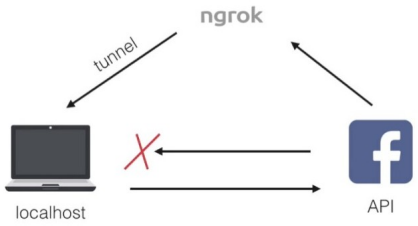
* Glitch: `https://glitch.com/`
* PageKite: `http://pagekite.net/`
* TeleConsole `https://www.teleconsole.com/`
* GoTTy `https://github.com/yudai/gotty`
* ngrok `https://ngrok.com/`

Some notes on ngrok: `https://ngrok.com/`
What is Ngrok? Ngrok is multiplatform tunneling, reverse proxy software that establishes secure tunnels from a public endpoint such as internet to a locally running network service while

[12]Facebook messenger

[13]Notice that most of these services are free and they will expire your https after a few days. Remind to update them if you want to use your chatbot in the future.

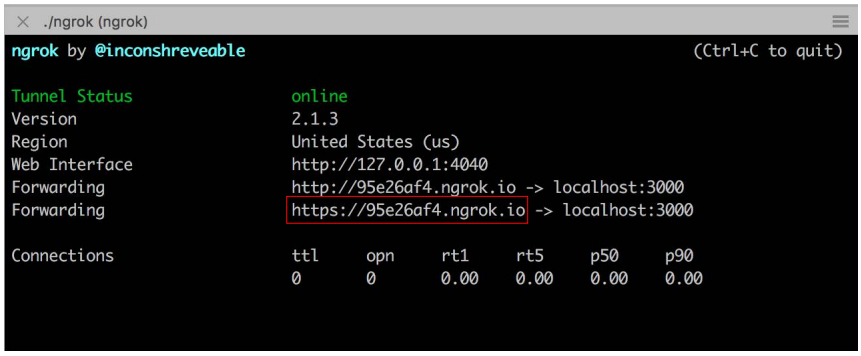
capturing all traffic for detailed inspection and replay (check the following figure).



How to Use it?

* install: `npm i -g ngrok`
* usage: `ngrok http [-subdomain myapp] [dstport]`
  If you don't have access to the ngrok link using your wifi, switch to your cellphone wifi just for installing and generating an https with ngrok.

How generate a private https with ngrok:
go to the terminal where the ngrok is installed. Write
`ngrok http 3000`
in the terminal. You will receive a list of results as:



The marked https by red is the https generated by ngrok.
How to register your WebHook on Facebook? After generating a private https, go to your app dashboard in the facebook developer page. Find the webhooks option and click on the setup webhooks button. Fill the callback URL with your HTTPS coming from one of the above services (for instance, with the URL

provided from ngrok) and copy paste your verify token from the development.json file. And click only messages in the subscription fields at the moment. By this way, we authorize the messenger platform to only send messages to our chatbot. If the webhook is valid you will see a green logo in your page.
More explanation on Webhook: For getting more details on the webhook registration and characteristics check this link `https://developers.facebook.com/docs/graph-api/webhooks/getting-started/` // Finally test the registerHook() function in your server.js to see if it has been registered for the chatbot or not.

Exe 14 Securing webhook (optional) Once your webhook is registered, your chatbot is ready to recieve messages from facebook. The facebook messenger can send the messages to your chatbot by calling the `POST`. In this step your chatbot can verify the payload signature sent by facebook. For security reasons, you probably want to limit requests to those coming from facebook. This is optional but recommended for the webhook security.

facebook ⟶ POST ⟶ FBeamer ?verify payload signature?

every incoming payload message from Facebook comes with a header namely `x-hub-signature`. This header contain SHA1 equivalent of the payload. SHA1 is a cryptographic hashing algorithm where creating one requires a key. In our case this key used is a unique string key provided by facebook. In the same way as it provides in page access token. According to figure 3, if we take the payload data with the sha1 key generated by appsecret provided by facebook, we should generate the exact same key in our chatbot. If this happens the signature matches the facebook key which verifies the message validation.

To do so, first install the bodyparser package using:
`npm i body-parser`

Get an app secret key: Go to your app dashboard on developers.facebook.com, click on settings, go to the basic section, here you

will find the app secret key. Copy it and add it to `development.json` as `appSecret`. Consequently add it in the config/index.js. Add it in FBeamer constructor as well.

import body-parser inside `server.js`. Write a server.post function in `server.js`. This is where the incoming messages come first.

```
1 const f = new FBeamer(config.FB);
2 server.post('/', bodyparser.json({
3   verify: f.verifySignature.call(f);
4 }));
```

Notice: The reason of defining line 3 as this is because the verify property internally invokes the verifySignature's returned function within its class. For this reason, the 'this' context is lost. By binding it to the FBeamer instance, we insure the context is preserved. Also, the `.call()` here runs the verify method.
where `verifySignature` will be implemented later in the fbeamer. We write another `post` function for receiving messages after passing through the signature. Just write the general part of function in your code in server.js. The code will be completed later.

```
1 server.post('/', (req, res, next) =>{
2   // message will be received here if the signature is
       verified
3   //the message will be passed to FBeamer for parsing
4 });
```

Write a function namely verifySignature in your fbeamer/index.js file for verifying if the received payload from the FB is valid or not. For completing this function try to check if the `x-hub-signature` coming from heading request is the same as sha1 hash using the given appSecret (we got appSecret from facebook messenger). If it is not the case `throw` errors. *Hint: you need crypto package for this part.*

The `verifySignature()` can be defined as:

```
1 const crypto = require('crypto');
2   verifySignature(req, res, buf) {
3     return (req, res, buf) => {
4       if(req.method === 'POST') {
5         try {
6           // get x-hub-signature here
7           /*this code generates a hash using the given
               appSecret*/
```

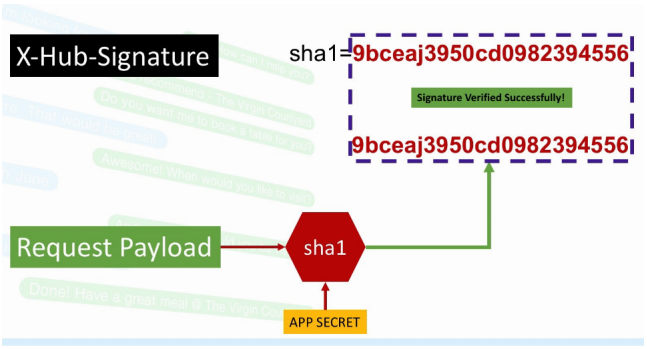


Figure 3:

```
8           let tempo_hash = crypto.createHmac('sha1', <
                your appSecret>).update(buf, 'utf-8');
9           let hash = hash.digest('hex');
10        // complete the rest of code by yourself
11
12          } catch (e) {
13            console.log(e);
14          }
15        }
16      }
17    }
```