ferdinand-muetsch.de

How to make Telegram Bots

Recently Telegram has introduced a new feature, the bots. Basically the bots enable human Telegram users to talk to machines, i.e. customly written little programs. A bot could be a daily helper, be it a bot that you can ask for the current temperature, one that googles something for you, one to manage your todo's or even a little text based game – everything within the Telegram chat. The nice thing about them is that they're really simple to create. You can read more about the bots in general here: https://telegram.org/blog/bot-revolution

This article shouldn't cover how to create and publish a bot (which actually is the same step), but how to write its actual functionality – its backend. On how to initially set up one, please refer to this little guide: https://core.telegram.org/bots. It's very easy, trust me. Everything you need for that is your Telegram app.

What you need further to program your bot basically is a favorite programming language, an IDE or at least a text editor, this page to be open and again your app. In this tutorial we will use Node.js as programming language (or more precisely as programming platform), but you really could use any other language, as well. Probably Python or PHP would do the job well, but you could even write a Java application.

First some basic things to understand. For the most parts your bot is your own application which we will try to create here, running on your own local PC or server. Telegram won't host any code. All functionality and data storage is kept in your program on your machine. Basically Telegram doesn't provide more than kind of an interface between your users' Telegram client (app) and your bot application. The flow would be like:

1. You create a new bot with @BotFather and set its description and commands (the commands you set there are – strictly speaking – completely independent of which commands your program will actually accept – they are just strings which the user gets suggested in a chat with your bot) so that the bot gets publicly accessible as any other (human) Telegram user is via her @ nickname.

2. You write the backend and run it to be listening.

3. A user sends a message to your bot.

4. The (JSON formatted) Telegram message object gets passed to your backend via HTTPS by Telegram.

5. Your backend parses the message text, extract the commands and arguments, processes some logic, possibly does some database actions and so on.

6. Your backend passes a response message object to the Telegram API via HTTPS (the other way round now).

7. Telegram shows the message to your user’s chat window.

Alright, now let’s get into the code. I require you to already have set up a bot and got its authentication token (all using @BotFather). As I said, we will make a little Node.js program here, but for those who don’t understand JavaScript too well, I’ll try to explain everything as clear as you need to re-do this in your programming language.

First, you need any library that can do HTTP requests, because basically your backend will be busy doing POSTs and GETs for the most time. For Node.js we’ll use the Unirest library, which probably is a little bit overkill, but very simple for our purpose. The library is available for Python, PHP, Ruby, Java and many others as well.

So first step is setting up a Node application and requiring Unirest (i assume you know how to set up a Node application and you’re familiar with npm).

```

var unirest = require('unirest');

```

Next is to receive messages from your users. Telegram offers two ways to do this. The first way would be to use Webhooks, which basically means that you’re backend runs on a webserver (or for Node it actually is one itself), Telegram knows your HTTPS endpoint and does a request to your backend every time a message is sent by a user. I would consider this the more elegant way. But the minus about this is that you would need to have a valid SSL certificate (which costs monthly charge) to provide a secure HTTPS connection. Telegram won't send any data to an unsecure or unverified endpoint. This is why we will take the other approach, which works kind of the other way round. Your backend will continuously request the Telegram API if there are new messages available. In detail your backend won't do request after request after request (because this would be so inefficient!) but use long polling. To put it simple long polling means that a request won't be answered instantly, but kept open until there is some data available.

We need to specify some constants. As i said, everything between your backend and Telegram happens via their HTTPS interface. We set up constants for the base API url, containing your bot token, the URL for the getUpdates method endpoint and the URL for the sendMessage method endpoint. The *:offset:* within the URL string will get replaced by a number, specifying, which new messages to fetch from Telegram later.

Now we'll introduce a function called *poll* (you can choose any other name), which basically is kind of the main loop of our program. Here's the code for this method, explanation follows.

```
function poll(offset) {

    var url = POLLING_URL.replace(":offset:", offset);

    unirest.get(url)

    .end(function(response) {
        var body = response.raw_body;
        if (response.status == 200) {
            var jsonData = JSON.parse(body);
            var result = jsonData.result;
            if (result.length > 0) {
                for (i in result) {
                    if (runCommand(result[i].message)) continue;
                }
                max_offset = parseInt(result[result.length - 1].update_id) + 1;
            }
            poll(max_offset);
        }

        });
};

```

Alright. The function is recursive, meaning it will call itself – namely each time, a request was answered (which is, when a new message was fetched). The http request to Telegram API to get updates will be pending until a message arrives. Then it gets answered, but has to be re-opened again instantly now to continue listening for message updates again. As a parameter it takes an offset number, which replaces the *:offset:* placeholder in the url string. To read more about this parameter, go to https://core.telegram.org/bots/api#getupdates. Unirest opens an http request to the specified url and executes the callback function given to end(), if the request was answered. First, we extract the response body. Afterwards, we check if our request was successful. If this is the case, we parse the body (which is a JSON object, consisting of an *ok* field and a *result* array. The result array contains one or more message objects. These are the ones that are relevant for us. For each message object, we try to parse it as a command (i'll give you the runCommand() function is a second…) and depending on which command we got, execute the respective method. Afterwards the offset gets updates to the id of the latest message plus one (to not receive it again next time) and a new request gets opened.

```
var dosth = function(message) {
}
var COMMANDS = {
    "dosth" : dosth
};
```

Now we specify a map, which maps strings (representing the users' command input – in this case */dosth* to actual functions.

```
function runCommand(message) {
    var msgtext = message.text;
    if (msgtext.indexOf("/") != 0) return false;
    var command = msgtext.substring(1,
msgtext.indexOf(" "));
    if (COMMANDS[command] == null) return false;
    COMMANDS[command](message);
    return true;
}
```

And this is the runCommand method. It takes the entire Telegram message object, which we got as a response from the Telegram API above and tries to parse its text as a command. A command string always starts with a slash. So if there is not slash in the beginning, we can be sure that we didn’t get a valid command. We simply return here, but we also could send a message to the user telling him “Hey, please enter a valid command.”. But let’s keep it simple. In the following line we extract everything after the slash and before the first blank space (assuming a command mustn’t contain a blank space) as the command. Afterwards we look into our map if the command actually is a key for a method and if so, we just run this method, passing it the message object as a parameter as the function will need information out of it.

What have we done so far? We wrote a program that requests the Telegram API for new messages, parses potentially contained commands and runs functions depending on these commands.

All we still need is to implement the method belonging to the */dosth* command.

```
var dosth = function(message) {
    var caps = message.text.toUpperCase();
    var answer = {
        chat_id : message.chat.id,
        text : "You told be to do something, so I took your input and made it all caps. Look: " + caps
    };
    unirest.post(SEND_MESSAGE_URL)
        .send(answer)
        .end(function (response) {
            if (response.status == 200) console.log("Successfully sent message to " + message.chat.id);
        });
}
```

Most times you’ll want to send a message as response to your user. You could also send an image, an audio, a location, … (see https://core.telegram.org/bots/api#available-methods). Every message object needs a *chat_id* field, containing a Telegram user id, so that Telegram knows which user to deliver your message to. We simply extract this id out of the message object's chat object we received from the user. The second mandatory field in a message object is the *text*. This is up to you. After having set up the new message object (you could add other fields, e.g. to show bot-buttons to the user – see https://core.telegram.org/bots/api#message for this), we just need to HTTP POST it to Telegram using Unirest again. Finished.

This example was kept veeeery simple. Of course you could implement your bot to do really fancy things. You could process media, do location-specific operations, include a database (MongoDB suits really well!) and many, many other things. You could write any complex application you can imagine – with a Telegram chat as the text i/o interface. Please tell be your ideas – what would be a great bot?

If you like to try by bot, simply write a message to **@FavoriteBot** and share it to your friends, if you like it 😃

If you have any questions, contact me via mail to *mail(at)ferdinand-muetsch.de.*