Nov 6 · 9 min read

# Introduction to the Telegram Bot API, Part 2

Make an inline Telegram bot in 100 lines of code

This article is a continuation from Introduction to the Telegram Bot API, Part 1, which explains what the Telegram Bot API is and how it works. In this series, we will build a Telegram bot for quickly looking up Pokémon.

In Part 2, we get hands-on and make a working Telegram bot with minimal functionality to ensure all the pieces we need are in place. Later in Part 3, we'll add the Pokédex data and search features.

## What the finished bot will do (recap)

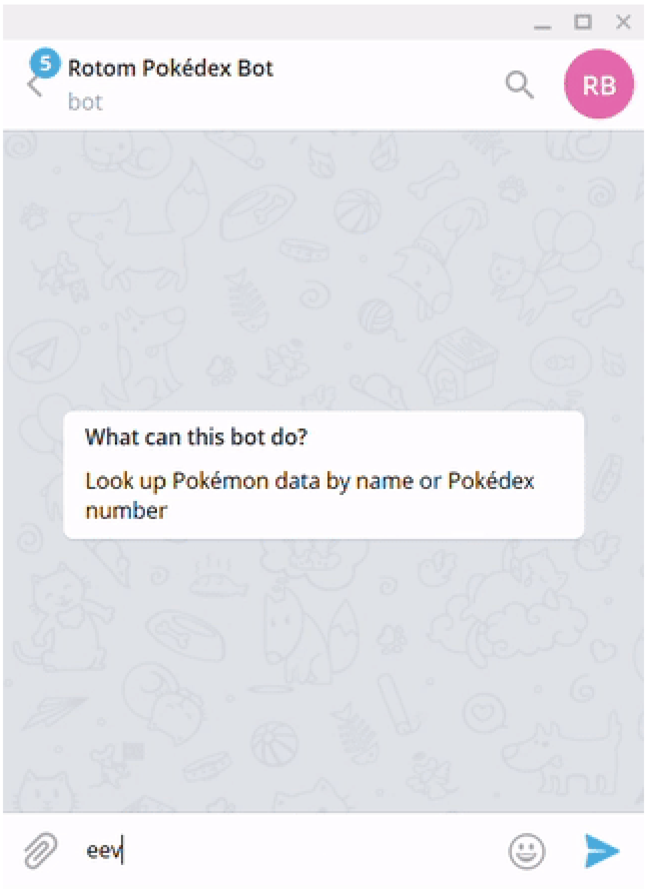
You can try out the finished bot on Telegram right now:

Rotom Pokédex Bot

Rotom Pokédex Bot is a bot to quickly look up Pokémon info on Telegram.

t.me

If you send the bot a text message directly, it tries to find a Pokémon with a matching name or Pokédex number:

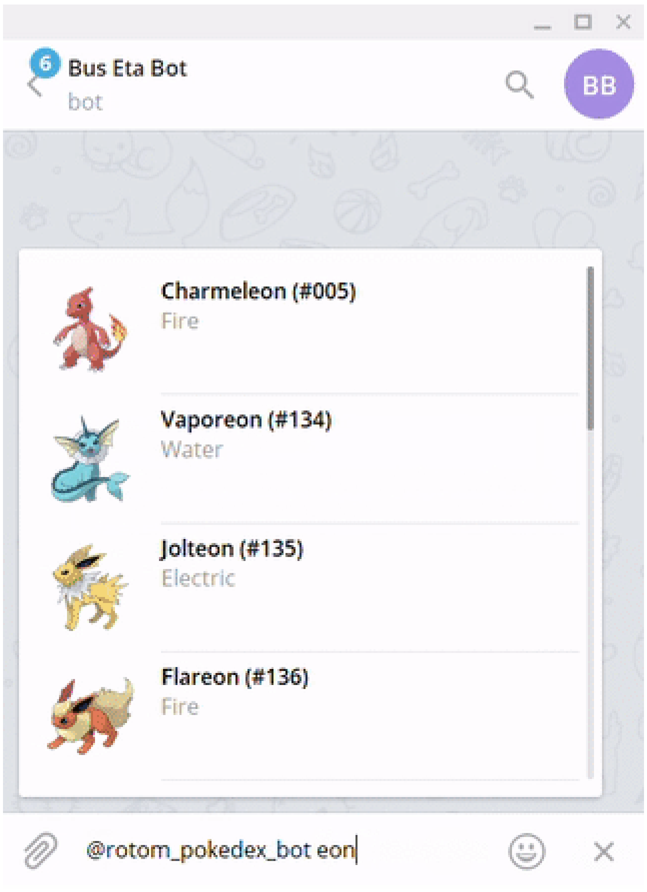


What can this bot do?

Look up Pokémon data by name or Pokédex number

eev

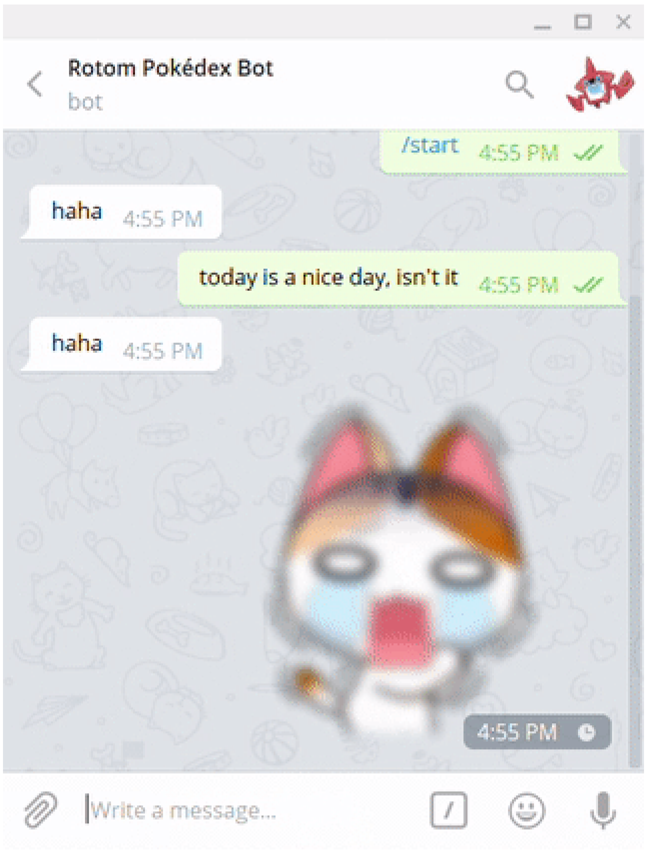
The bot also works in inline mode. You can send an inline query to search for Pokémon from any chat:



Do check out Bus Eta Bot too

## Getting started

In this part, we’ll be registering a new bot with Telegram and creating a new Cloud Function to run it. To make sure everything is connected together properly, we’ll create a simple bot first. I call it Haha Bot, after people who reply to every message with “haha”:



Rotom Pokédex Bot
bot

/start 4:55 PM

haha 4:55 PM

today is a nice day, isn't it 4:55 PM

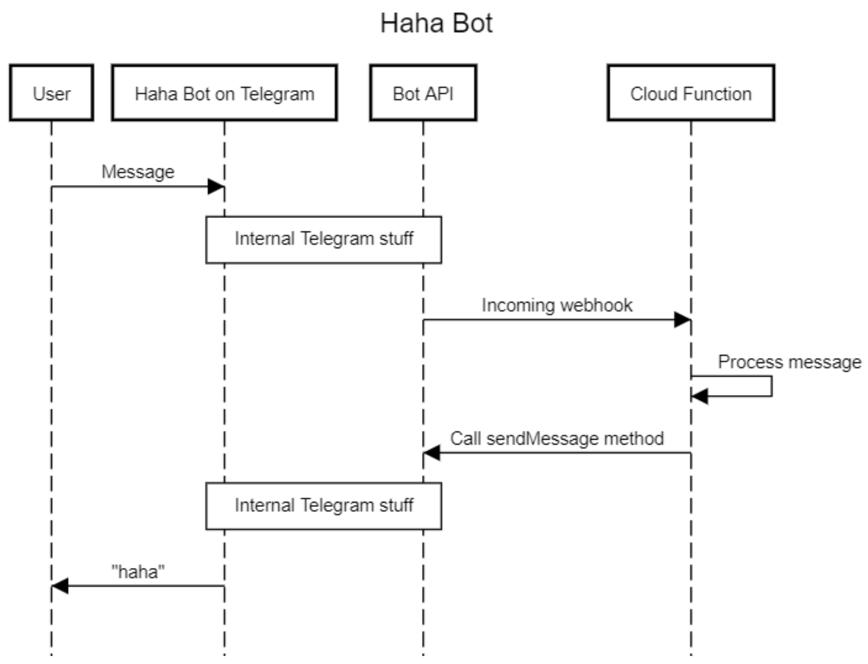
haha 4:55 PM

4:55 PM

Write a message...

Tag a friend

## How Haha Bot works: the stuff that happens every time you message it



We need to register a new Telegram bot with Telegram to obtain a bot token, then we'll create a new HTTP-triggered Cloud Function and set our function’s endpoint as our bot’s webhook URL.

As mentioned in Part 1, a Cloud Function is just some JavaScript code which runs when triggered, in this case by an incoming HTTP request. When a user interacts with our bot, the Bot API will make a HTTP request containing an update to our Cloud Function, which run and process the update. If the update represents a message, the function will invoke the `sendMessage` method on the Bot API and respond with “haha”.

## Obtaining a bot token

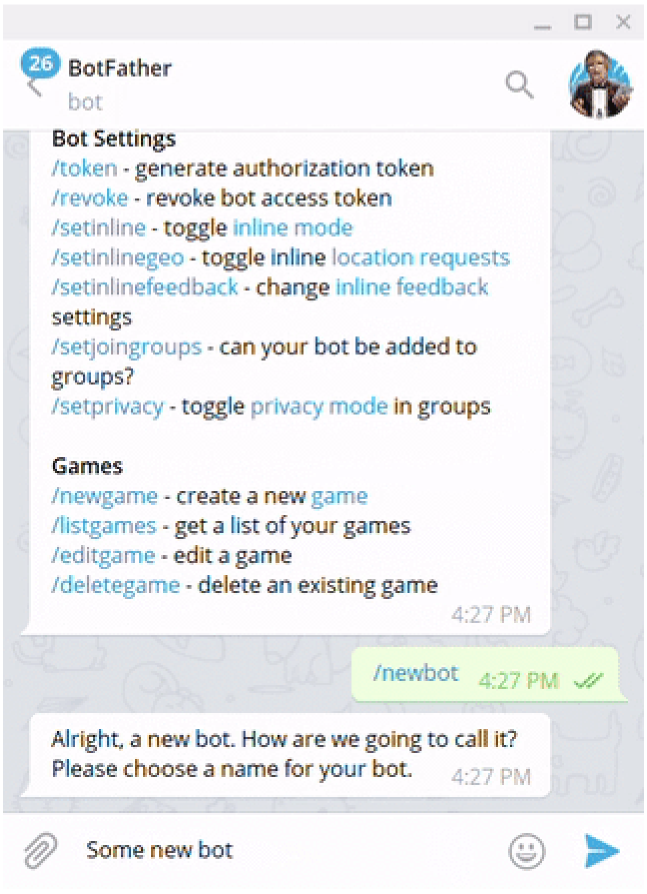
In order to access the Bot API, you must first obtain a bot token by creating a new bot with the Botfather:

BotFather

BotFather is the one bot to rule them all. Use it to create new bot accounts and manage your existi...

telegram.me

The Botfather itself is a Telegram bot which will guide you through choosing a name and username for your new bot and generate a bot token for you. Just send the `/newbot` command and follow its instructions:



BotFather
bot

Bot Settings
/token - generate authorization token
/revoke - revoke bot access token
/setinline - toggle inline mode
/setinlinegeo - toggle inline location requests
/setinlinefeedback - change inline feedback settings
/setjoingroups - can your bot be added to groups?
/setprivacy - toggle privacy mode in groups

Games
/newgame - create a new game
/listgames - get a list of your games
/editgame - edit a game
/deletegame - delete an existing game
4:27 PM

/newbot 4:27 PM

Alright, a new bot. How are we going to call it? Please choose a name for your bot. 4:27 PM

Some new bot

This bot was promptly deleted after the gif was recorded.

## Creating a Cloud Function

Before you can create a Cloud Function, you have to first create a Google Cloud Platform project to associate it with. Follow the GCP documentation on creating a new project and the quickstart for creating a new Cloud Function:

Creating and Managing Projects | Google Cloud Resource Manager Documentation | Google…

Google Cloud Platform projects form the basis for creating, enabling, and using all Cloud Platform…

cloud.google.com

Quickstart: Using the Console | Cloud Functions Documentation | Google Cloud…

This is a beta release of Google Cloud Functions. This API might be changed in backward-…

cloud.google.com

You can name your project and your function anything you want. Once you have created your new function, you will see it listed on your Cloud Functions overview page:

`function-1` is the function I just deployed from the quickstart ( `prod` is the function currently running Rotom Pokédex Bot). Take note of the endpoint when creating your function, or click on the name of your new function to view its details and navigate to the second tab "Trigger" to find your function's endpoint:

Visit your function in your browser, and you should see the text "No message defined!". Take a look at the quickstart function's source code:

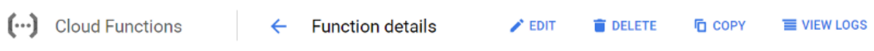
```
/**
 * Responds to any HTTP request that can provide a "message"
field in the body.
 *
 * @param {!Object} req Cloud Function request context.
 * @param {!Object} res Cloud Function response context.
 */
exports.helloWorld = function helloWorld(req, res) {
  // Example input: {"message": "Hello!"}
  if (req.body.message === undefined) {
    // This is an error case, as "message" is required.
    res.status(400).send('No message defined!');
  } else {
    // Everything is okay.
    console.log(req.body.message);
    res.status(200).send('Success: ' + req.body.message);
  }
};
```

When the `body.message` property of the incoming request is undefined, for example in a GET request from a browser, the function responds with "No message defined!". In the Test the function section of the quickstart, the function is called a JSON payload `{"message":"Hello World!"}` and responds with success instead.

## Modifying our function

Before we register our function's endpoint to receive updates from the Bot API, we should update our function's code to handle updates and reply to messages.

Click on the Edit button at the top of the your function's details to bring up the inline editor:



Inside the editor, replace the `exports.helloWorld` function with:

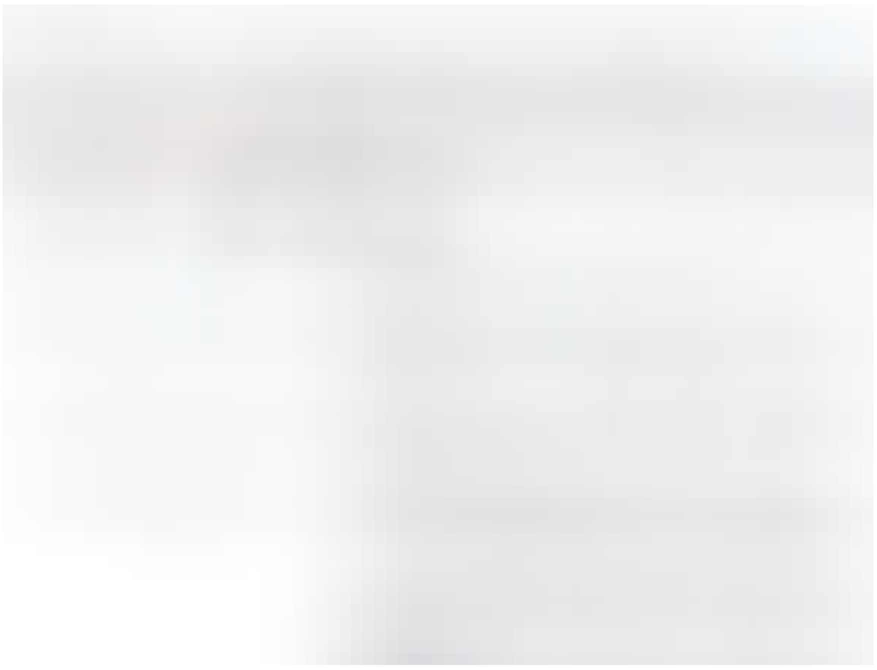
```
exports.haha = function (req, res) {
    const update = req.body;
    console.log(JSON.stringify(update));

    // update contains a message
    if (update.hasOwnProperty('message')) {
        // call the sendMessage method
        const reply = {
            method: 'sendMessage',
            chat_id: update.message.chat.id,
            text: 'haha',
        };

        return res.json(reply);
    }

    return res.sendStatus(200);
};
```

Also change the "Function to execute" to `haha` :

Scroll to the bottom and hit Save to deploy this new version of your function.

## Connecting our function to our bot

It's time to make our first actual Bot API request: we'll now call the `setWebhook` method to register our function's endpoint as our bot's webhook address to receive incoming updates. Look at the documentation for the `setWebhook` method (link):


Picture of text

To recap on calling Bot API methods from Part 1, the Bot API takes accepts both GET and POST requests and takes query string, JSON and form-encoded parameters. The endpoint for making your requests to takes the form `https://api.telegram.org/bot<token>/METHOD_NAME` and depends on your bot token and method you are calling. In this case, we are calling the `setWebhook` method so the URL is:

```
https://api.telegram.org/bot<token>/setWebhook
```

Replace `<token>` with the bot token you obtained a while ago. The word `bot` is part of the URL: for example, if your bot token is `123456:ABC-DEF1234ghIkl-zyx57W2v1u123ew11`, then the endpoint you should call is:

```
https://api.telegram.org/bot123456:ABC-DEF1234ghIkl-
zyx57W2v1u123ew11/setWebhook
```

From the documentation, the `setWebhook` method takes a few parameters; the only one which is required and which matters to us right now is the `url` parameter: this is the address we want the Bot API to send updates to. Since one parameter is pretty manageable, we can just append it to the endpoint as a query string:

```
https://api.telegram.org/bot123456:ABC-DEF1234ghIkl-
zyx57W2v1u123ew11/setWebhook?url=https://us-central1-your-
gcp-project.cloudfunctions.net/function-1
```

Note that you have to change the portion after `?url=` to your own Cloud Function's endpoint.

To actually make this request, you can use Postman (a useful GUI tool for making HTTP requests), a command-line tool such as cURL or PowerShell's `Invoke-WebRequest` , or even your browser since the Bot API accepts GET requests with query string parameters.

In this case, the simplest way would be to just paste it into your address bar and go.

For completeness' sake, I'll also provide a cURL example in Bash and an `Invoke-WebRequest` example in PowerShell:

Bash:

```
jiayu@jy-9360:~$ TOKEN=your-bot-token
jiayu@jy-9360:~$ URL=your-cloud-function-endpoint
jiayu@jy-9360:~$ curl
https://api.telegram.org/bot$TOKEN/setWebhook?url=$URL
```

PowerShell:

```
PS C:\Users\jiayu> $token = 'your-bot-token'
PS C:\Users\jiayu> $url = 'your-cloud-function-endpoint'
PS C:\Users\jiayu> Invoke-WebRequest
https://api.telegram.org/bot$token/setWebhook?url=$url
```

Whichever method you used, you should get a response like the following:

```
{"ok":true,"result":true,"description":"Webhook was set"}
```

At this point, you should be able to open Telegram, search for your bot by its username, and start talking to it. Congratulations! You've just made a (barely) working Telegram bot.

## A closer look at Haha Bot

Before we end off, let's take a closer look at Haha Bot's source code:

On line 1 is an assignment to the `exports` object in the global scope. This is a Node.js thing (Cloud Functions runs Node.js under the hood). The `exports` object contains a module's exported symbols. I've arbitrarily called decided to call the function `haha` since this is Haha Bot after all.

Also on line 1 is a function declaration containing two parameters, `req` and `res` . This is because Cloud Functions also uses Express as its web framework, and your exported function will be an Express route.

It's not necessary, although useful, to know Express to write cloud functions; just remember that the `req` object represents the incoming HTTP request, and the `res` object is used to respond to it. `req` and `res` are mnemonics for "request" and "response" after all.

On line 2, we access the `body` property on the `req` object. `req.body` represents the body of an incoming HTTP request. Since we know the incoming HTTP request is from the Bot API, the body of this request is the Update object (docs). We also log it for debugging just in case. Using `JSON.stringify()` ensures that we log the entire object and deeply-nested properties don't get replaced with `[object Object]` .

On line 5, we check if the update object has a `message` property. If you refer to the documentation for the Update object, it will always have one other field besides `update_id` representing the content of the incoming update. An update with a `message` property represents a new "incoming message of any kind—text, photo, sticker, etc." Other updates could have different fields representing different types of updates, for example edited messages ( `edited_message` ), inline queries ( `inline_query` ) or callback queries ( `callback_query` ) and more, but we're not interested these.

On line 8, we prepare a response to the incoming request we just received. This will be a JSON object with a few properties. As mentioned briefly in the Bot API documentation and Part 1, you can also make requests to the Bot API by responding to incoming webhooks with a `method` field containing the method you wish to call. In order to send a "haha" in response to the message we just received, we'll use the `sendMessage` method, hence `method: 'sendMessage'` . This is followed by the required `chat_id` and `text` parameters representing the chat we wish to send a message to, and the text to send.

Why `update.message.chat.id` for `chat_id` : We already know the incoming update has a `message` field containing a Message object (docs). The Message object has a field `chat` representing the conversation the message belongs to. (This is sometimes different from the `from` field, which represents the user who sent the message, for example in group chats.) Finally, the `chat` object contains the `id` of the originating chat.

On line 14, `res.json(body)` is part of the Express Response object (docs). This method converts its argument to a JSON string and sends it in the response body, as well as setting the appropriate `Content-Type` header. The function returns after this.

If the incoming update did not contain a `message` property, then execution jumps to line 17 instead. In this case, we just respond with a `200 OK` response, telling the Bot API that we've received and handled this update so that it doesn't try to resend it, and do nothing else.

## To be continued…

Now that we've got a rudimentary Telegram bot set up and working on Cloud Functions, we just need to update the function's source code to update the bot. In Part 3, we'll look for a source of Pokémon data to use and upgrade our bot to handle inline queries as well.

. . .

Thanks for reading! If you liked this post, you can also:

- Clap for this post and follow me on Medium for future updates
- Check out some of my previous posts about the Telegram API, using Cloud Functions to create dynamic README badges, or sentiment analysis of Facebook comments with the Google Cloud Natural Language API
- Try out Bus Eta Bot, my Telegram bot for checking bus etas in Singapore, or check out its source code on GitHub