# Webhooks

Jannes Höke edited this page on 24 Aug · 15 revisions

## Introduction

Our examples usually start the bot using `Updater.start_polling` . This method uses the getUpdates API method to receive new updates for your bot. This is fine for smaller to medium-sized bots and for testing, but if your bot receives a lot of traffic, it might slow down the response times. There might be other reasons for you to switch to a webhook-based method for update retrieval.

**First things first:** You should have a good reason to switch from polling to a webhook. Don't do it simply because it sounds cool.

## Polling vs. Webhook

The general difference between polling and a webhook is:

- Polling (via `get_updates` ) periodically connects to Telegram's servers to check for new updates
- A Webhook is a URL you transmit to Telegram once. Whenever a new update for your bot arrives, Telegram sends that update to the specified URL.

## Requirements

There's a number of things you need to retrieve updates via a webhook.

### A public IP address or domain

Usually this means you have to run your bot on a server, either a dedicated server or a VPS. Read Where to host Telegram Bots to find a list of options.

Make sure you can connect to your server from the **public internet**, either by IP or domain name. If `ping` works, you're good to go.

### A SSL certificate

All communication with the Telegram servers must be encrypted with HTTPS using SSL. With polling, this is taken care of by the Telegram Servers, but if you want to receive updates via a Webhook, you have to take care of it. Telegram will not send you any updates if you don't.

There are two ways to do this:

1. A verified certificate issued by a trusted certification authority (CA)
2. A self-signed certificate

If you don't already have a verified certificate, use a self-signed one. It's easier and there is no disadvantage to it.

#### Creating a self-signed certificate using OpenSSL

To create a self-signed SSL certificate using `openssl` , run the following command:

```
openssl req -newkey rsa:2048 -sha256 -nodes -keyout private.key -x509 -days 3650
-out cert.pem
```

**Clone this wiki locally**

`https://github.com/pythor`

The `openssl` utility will ask you a few details. **Make sure you enter the correct FQDN!** If your server has a domain, enter the full domain name here (eg. `sub.example.com`). If your server only has an IP address, enter that instead. If you enter an invalid FQDN (Fully Qualified Domain Name), you won't receive any updates from Telegram but also won't see any errors!

## Choosing a server model

There actually is a third requirement: a HTTP server to listen for webhook connections. At this point, there are several things to consider, depending on your needs.

### The integrated webhook server

The `python-telegram-bot` library ships a custom HTTP server, based on the CPython `BaseHTTPServer.HTTPServer` implementation, that is tightly integrated in the `telegram.ext` module and can be started using `Updater.start_webhook`. This webserver also takes care of decrypting the HTTPS traffic. It is probably the easiest way to set up a webhook.

However, there is a limitation with this solution. Telegram currently only supports four ports for webhooks: *443, 80, 88* and *8443.* As a result, you can only run a **maximum of four bots** on one domain/IP address.

If that's not a problem for you (yet), you can use the code below (or similar) to start your bot with a webhook. The `listen` address should either be `'0.0.0.0'` or, if you don't have permission for that, the public IP address of your server. The port can be one of `443`, `80`, `88` or `8443`. For the `url_path`, it is recommended to use your Bot's token, so no one can send fake updates to your bot. `key` and `cert` should contain the path to the files you generated earlier. The `webhook_url` should be the actual URL of your webhook. Include the `https://` protocol in the beginning, use the domain or IP address you set as the FQDN of your certificate and the correct port and URL path.

```
updater.start_webhook(listen='0.0.0.0',
                      port=8443,
                      url_path='TOKEN',
                      key='private.key',
                      cert='cert.pem',
                      webhook_url='https://example.com:8443/TOKEN')
```

### Reverse proxy + integrated webhook server

To solve this problem, you can use a reverse proxy like *nginx* or *haproxy*.

In this model, a single server application listening on the public IP, the *reverse proxy*, accepts all webhook requests and forwards them to the correct instance of locally running *integrated webhook servers.* It also performs the *SSL termination*, meaning it decrypts the HTTPS connection, so the webhook servers receive the already decrypted traffic. These servers can run on *any* port, not just the four ports allowed by Telegram, because Telegram only connects to the reverse proxy directly.

**Note:** In this server model, you have to call `set_webhook` yourself.

Depending on the reverse proxy application you (or your hosting provider) is using, the implementation will look a bit different. In the following, there are a few possible setups listed.

#### Heroku

On Heroku using webhook can be beneficial on the free-plan because it will automatically manage the downtime required. The reverse proxy is set up for you and an environment is created. From this environment you will have to extract the port the bot is supposed to listen on. Heroku manages the SSL on the proxy side, so you don't have provide the certificate yourself.

```
import os

TOKEN = "TOKEN"
PORT = int(os.environ.get('PORT', '5000'))
updater = Updater(TOKEN)
# add handlers
```

```
updater.start_webhook(listen="0.0.0.0",
                      port=PORT,
                      url_path=TOKEN)
updater.bot.set_webhook("https://<appname>.herokuapp.com/" + TOKEN)
updater.idle()
```

### Using nginx with one domain/port for all bots

This is similar to the Heroku approach, just that you set up the reverse proxy yourself. All bots set their `webhook_url` to the same domain and port, but with a different `url_path`. The integrated server should usually be started on the `localhost` or `127.0.0.1` address, the port can be any port you choose.

**Note:** `example.com` could be replaced by an IP address, if you have no domain associated to your server.

Example code to start the bot:

```
updater.start_webhook(listen='127.0.0.1', port=5000, url_path='TOKEN1')
updater.bot.set_webhook(webhook_url='https://example.com/TOKEN1',
                        certificate=open('cert.pem', 'rb'))
```

Example configuration for `nginx` (reduced to important parts) with two bots configured:

```
server {
    listen              443 ssl;
    server_name         example.com;
    ssl_certificate     cert.pem;
    ssl_certificate_key private.key;

    location /TOKEN1 {
        proxy_pass http://127.0.0.1:5000;
    }

    location /TOKEN2 {
        proxy_pass http://127.0.0.1:5001;
    }
}
```

### Using haproxy with one subdomain per bot

In this approach, each bot is assigned their own *subdomain*. If your server has the domain *example.com*, you could have the subdomains *bot1.example.com*, *bot2.example.com* etc. You will need one certificate for each bot, with the FQDN set for their respective subdomain. The reverse proxy in this example is `haproxy`. The integrated server should usually be started on the `localhost` or `127.0.0.1` address, the port can be any port you choose.

**Note:** For this to work, you need a domain for your server.

Example code to start the bot:

```
updater.start_webhook(listen='127.0.0.1', port=5000, url_path='TOKEN')
updater.bot.set_webhook(webhook_url='https://bot1.example.com/TOKEN',
                        certificate=open('cert_bot1.pem', 'rb'))
```

Example configuration for `haproxy` (reduced to important parts) with two bots configured. Again: The FQDN of both certificates must match the value in `ssl_fc_sni`. Also, the `.pem` files are the `private.key` file and `cert.pem` files concatenated:

```
frontend  public-https
    bind        0.0.0.0:443 ssl crt cert_key_bot1.pem crt cert_key_bot2.pem
    option      httpclose

    use_backend bot1 if  { ssl_fc_sni bot1.example.com }
    use_backend bot2 if  { ssl_fc_sni bot2.example.com }

backend bot1
    mode            http
    option          redispatch
    server          bot1.example.com 127.0.0.1:5000 check inter 1000

backend bot2
    mode            http
    option          redispatch
    server          bot2.example.com 127.0.0.1:5001 check inter 1000
```

## Custom solution

You don't necessarily have to use the integrated webserver *at all*. If you choose to go this way, **you should not use the `Updater` class.** The `telegram.ext` module was designed with this option in mind, so you can still use the `Dispatcher` class to profit from the message filtering/sorting it provides. You will have to do some work by hand, though.

A general skeleton code can be found below.

**Setup part, called once:**

```
from queue import Queue  # in python 2 it should be "from Queue"
from threading import Thread

from telegram import Bot
from telegram.ext import Dispatcher

def setup(token):
    # Create bot, update queue and dispatcher instances
    bot = Bot(token)
    update_queue = Queue()

    dispatcher = Dispatcher(bot, update_queue)

    ##### Register handlers here #####


    # Start the thread
    thread = Thread(target=dispatcher.start, name='dispatcher')
    thread.start()

    return update_queue
    # you might want to return dispatcher as well,
    # to stop it at server shutdown, or to register more handlers:
    # return (update_queue, dispatcher)
```

**Called on webhook** with the decoded `Update` object (use `Update.de_json(json.loads(text), bot)` to decode the update):

```
def webhook(update):
    update_queue.put(update)
```

**Alternative (no threading)**

**Setup part, called once:**

```
from telegram import Bot
from telegram.ext import Dispatcher

def setup(token):
    # Create bot, update queue and dispatcher instances
    bot = Bot(token)

    dispatcher = Dispatcher(bot, None, workers=0)

    ##### Register handlers here #####

    return dispatcher
```

**Called on webhook** with the decoded `Update` object (use `Update.de_json(json.loads(text))` to decode the update):

```
def webhook(update):
    dispatcher.process_update(update)
```