**Telegram Bot Assignment**

**Intro:**

In this tutorial we will create a Telegram bot.

This lesson will guide you step by step, in creating a server, making it public to the world, and interacting with the

Telegram API.

Get ready, the fun is about to begin.

**Setup:**

In order to complete this lesson you will need a few tools and libraries.

Let's make sure you get everything you need.

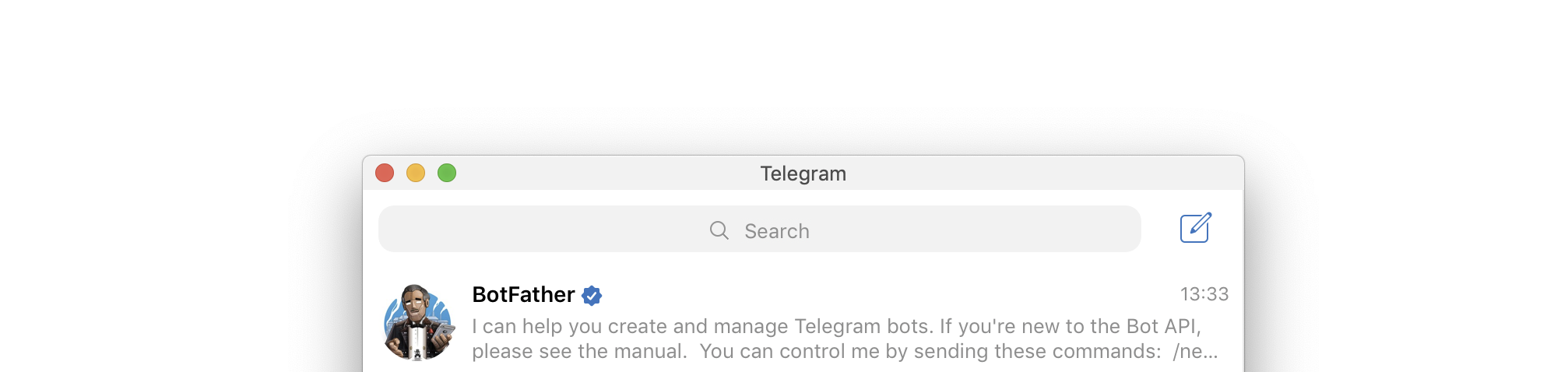
* Flask python package - pip install flask
* requests python package - pip install requests
* [Telegram](https://desktop.telegram.org/) app
* [ngrok](https://ngrok.com/download)

**Create a Telegram Bot**

We will start by creating a telegram bot. For that I assume you have a telegram application on your computer or mobile.

(If you don't install one and continue).

We create a telegram bot with the help of the BotFather, so search him in the telegram app:



Great! Now, after you found him type:

|  |
| --- |
| /newbot |

This will instruct the BotFather to create a new bot. After that you will need to give your bot a name (private in your user list) and a unique username.

Finally the BotFather will give you a token which you will need for later, so save it somewhere, we will get to it later.

Great! Let's move on!

**Server Setup:**

If you remember our architecture, we need a server to send and receive messages from the telegram server.

So we will start by creating a basic Flask server.

|  |
| --- |
| Import the flaks package  Create an app  Create a /sanity route that will return a simple "Server is running" message.  Run the server on an available port |

Now, make sure that everything works.

Your code should look something like that:

|  |
| --- |
| from flask import Flask  app = Flask(\_\_name\_\_)  @app.route('/sanity')  def sanity():return "Server is running"  if \_\_name\_\_ == '\_\_main\_\_':  app.run(port=5002) |

Before moving on, we want to make our server public. For that we will use [ngrock](https://ngrok.com/).

Ok now you know what to do - run ngrok by typing ngrok http <port number>

For example:

|  |
| --- |
| $ ngrok http 5002 |

where 5002 is the port you are running your server on,

and save it's external secured url:

Make sure you are taking the secured url, in this case: https://1562b746.ngrock.io

Check that your server is available through this address.

For example try your sanity route:

Great! we're all set.

Let's move on.

**Telegram API**

Ok, so we got our server public and running! amazing! Let's start using the Telegram API.

In order for Telegram to know our server we will start by sending an HTTP post request that will establish the connection.

Regular server usually just wait for requests to arrive, but in our case we need Telegram service to know us

and to be able to send us requests, and of course we will reply with the response that will eventually arrive to the

end user, the client that talks to our bot.

Here is how it looks like:

|  |
| --- |
| TOKEN = '1035593585:AAGndkikGde9JLIp10Ehb43S-YnGPwSMGbOg'  TELEGRAM\_INIT\_WEBHOOK\_URL = 'https://api.telegram.org/bot{}/setWebhook?url=https://1562b746.ngrok.io/message'.format(TOKEN)  requests.get(TELEGRAM\_INIT\_WEBHOOK\_URL) |

What is going on here?

Let's break it down.

The url structure is this:

<Telegram Base url>/bot<Bot Token>/<API Method>?<Url query param>

1. We start with Telegram base API url, which is: https://api.telegram.org/bot
2. Then we concat the bot token (remember? we got it from the BotFather)
3. The route we are using is: setWebhook
4. Last is the query string. Here we specify our server url, where Telegram will send the messages:
   1. The first part of the url, is the server domain, we got it from ngrok
   2. The second part is the path, the route which we will define in our server. In this case we chose message
5. Last is using the requests package (don't forget to import it) to send the post request.

Now Telegram can send us requests, but there is no one to receive them :(

You are right! We need to define a route:

|  |
| --- |
| @app.route('/message', methods=["POST"])  def handle\_message():  print("got message")  return Response("success") |

Now it's connected.

Check it out!

When you send a request to the bot, you will get a print on our server .

Let's wrap up by sending a message back to the end user.

Again we will need to compose the Telegram API route, this time for sending a message:

We will:

use the same base url: https://api.telegram.org/bot

concat the bot token: https://api.telegram.org/bot<bot token>

add the method name: https://api.telegram.org/bot<bot token>/sendMessage

add the query params: https://api.telegram.org/bot<bot token>/sendMessage?chat\_id=11&text=got it

All in all it will look like this:

|  |
| --- |
| @app.route('/message', methods=["POST"])  def handle\_message():  print("got message")  chat\_id = request.get\_json()['message']['chat']['id']  res = requests.get("https://api.telegram.org/bot{}/sendMessage?chat\_id={}&text={}'"  .format(TOKEN, chat\_id, "Got it"))  return Response("success") |

We got the chat\_id parameter from the request object.

Since there are many bot clients, we need to know the specific one to send the response to.

That's it! Ready for the magic?

Send a request to the bot and see the bot answer "got it".

Good Job!

But hey, this bot only replies got it for everything we ask it?

Well, now it is your part to add some cool features and make the bot your own.

**Exercises:**

Is Prime Bot

Write an is prime bot.

The bot should have a /prime command that will determine if the number that was passed is prime. If it is, return "prime", if not return "not prime". If it is even (and not 2), return to the user "Come on dude, you know even numbers are not prime!"

Example:

User sends: /prime 13

The bot responds: prime

Now add some more checks:

* /factorial - is a number a result of factorial operation
* /palindrome - is the number a palindrome
* /sqrt - does the number have an integer square root.

When everything is up and running take a look at your code.

* Think about your code structure.
* Are you using modules?
* Do you have encapsulation?
* Do all methods and modules have 1 single responsibility?
* Is your code generic? modular?

You will probably want to take some time to refactor your code. Go for it!

Note: think about your app design.

How should you structure your code?

**Is Prime Bot Extension**

Let's add another ability.

We want to keep track after the messages that we get and return the most popular number that got queried.

Let's add a /popular command.

This command will return the number that appeared the most in the bot history.

So what do we need in order to do that?

For that we will need to keep track of the bot messages. What a better way to do that than a DB.

Which one should we choose?

Good question.

Start by thinking about what you need to store, and what you need to query.

Then decide!

Choose the better fitting DB and plan its structure.

Good luck!