Step-by-Step guide t
Chatbot with a simpl

Calixte Mayoraz · Follow
8 min read · Nov 4, 2022

Listen Share

Photo by Nicolas Picard on Unsplash

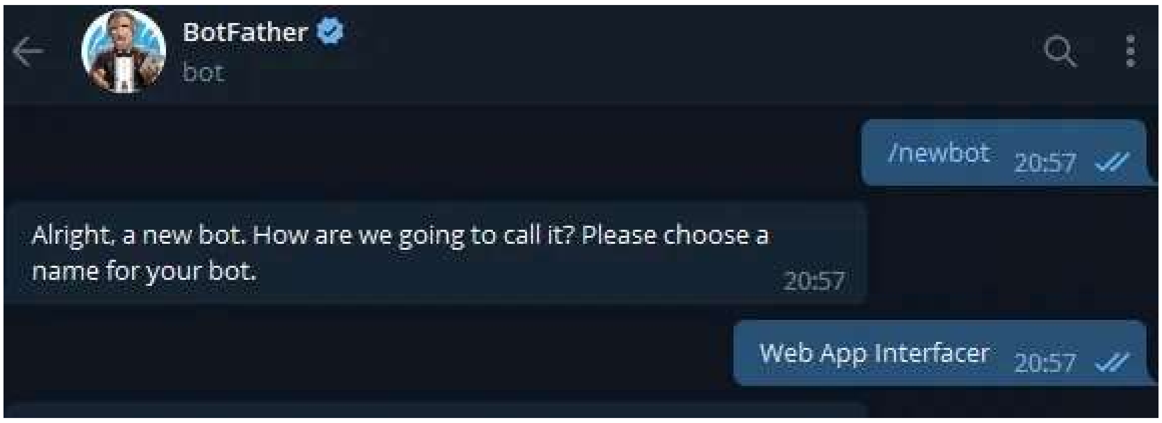
I started working on a project to help people discover craft beers using a Telegram Chatbot. To do this, the user needs to input their taste preferences along several types of tastes. I wanted to implement a UI with sliders to get this info from the user as intuitively as possible. Now, Telegram have done an incredible job at developing the chatbot backend to support Web-Apps, however the documentation (which is still under development) is at the time of writing this tutorial, *very* basic. So, I powered through it for a few days and thought I'd write a tutorial to help anyone in need for their future projects.

Let's make a simple chatbot that takes some input from a web-app, and stores it for later use.

> ***I'm assuming you have at least some basic knowledge of the following:***
>
> *Command-line stuff, Python, pip, Telegram, git, HTML/javascript*

Creating the bot

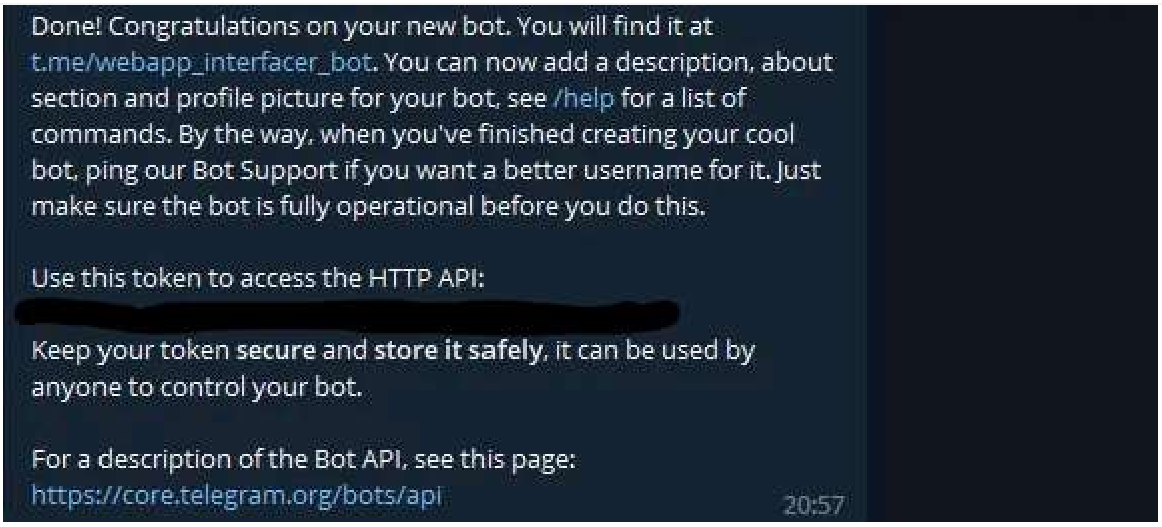
First things first, let's talk to botfather to create our bot.



/newbot 20:57

Alright, a new bot. How are we going to call it? Please choose a name for your bot. 20:57

Web App Interfacer 20:57


Done! Congratulations on your new bot. You will find it at t.me/webapp_interfacer_bot. You can now add a description, about section and profile picture for your bot, see /help for a list of commands. By the way, when you've finished creating your cool bot, ping our Bot Support if you want a better username for it. Just make sure the bot is fully operational before you do this.

Use this token to access the HTTP API:
[redacted]

Keep your token **secure** and **store it safely**, it can be used by anyone to control your bot.

For a description of the Bot API, see this page:
https://core.telegram.org/bots/api 20:57

Now we can start working on the bot. Let’s create a new folder for our code and install a few dependencies right off the bat.

```
# create our project
mkdir web_interfacer_bot
cd web_interfacer_bot

# Install some dependencies that we will need
pip install python_telegram_bot~=20.0a4
pip install python-dotenv~=0.21.0
```

And now we are ready to start coding. Let’s create an environment file to keep our sensitive information (like the bot token) in. Create a “.env” file at the root of your project and paste in your bot username (that you defined) and the bot token (given by botfather) inside like so:

```
BOT_USERNAME=webapp_interfacer_bot
BOT_TOKEN=1234567:ASDFASDFASDFsdjfhjskdfg9wexchgfsj45635
```

Now, we can create a python file that will load these environment variables from file (if it exists) or from the system environment variables (like when we are deploying). I’m calling it *credentials.py.*

```
"""credentials.py"""
import os
if os.path.exists(".env"):
    # if we see the .env file, load it
    from dotenv import load_dotenv
    load_dotenv()

# now we have them as a handy python strings!
BOT_TOKEN = os.getenv('BOT_TOKEN')
BOT_USERNAME = os.getenv('BOT_USERNAME')
```

Okay, now we can REALLY get down to making a bot. I’m using the boilerplate approach from python-telegram-bot, if you are unfamiliar with this framework, I **highly** suggest you take a read. Their tutorials are extremely well written and the rest of this tutorial will be easier to understand if this is your first bot.

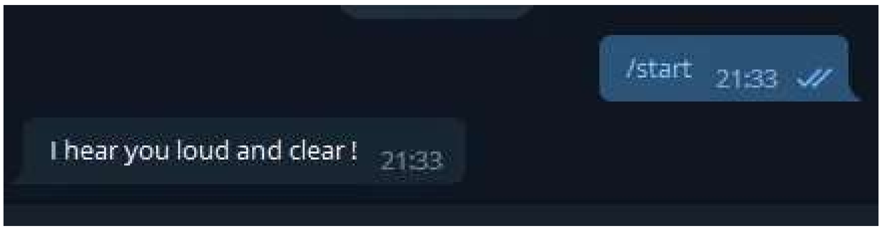
Create a file called "app.py" and copy/paste the following section inside:

```
"""app.py"""
from telegram import Update, KeyboardButton, ReplyKeyboardMarkup,
WebAppInfo
from telegram.ext import ApplicationBuilder, CallbackContext,
CommandHandler, MessageHandler, filters
from credentials import BOT_TOKEN, BOT_USERNAME
import json

async def launch_web_ui(update: Update, callback: CallbackContext):
    # For now, let's just acknowledge that we received the command
    await update.effective_chat.send_message("I hear you loud and
clear !")


if __name__ == '__main__':
    # when we run the script we want to first create the bot from
the token:
    application = ApplicationBuilder().token(BOT_TOKEN).build()

    # and let's set a command listener for /start to trigger our Web
UI
    application.add_handler(CommandHandler('start', launch_web_ui))

    # and send the bot on its way!
    print(f"Your bot is listening! Navigate to
http://t.me/{BOT_USERNAME} to interact with it!")
    application.run_polling()
```

And sure enough, if we run this python script and follow the link displayed on screen:

```
Your bot is listening! Navigate to http://t.me/webapp_interfacer_bot
to interact with it!
```

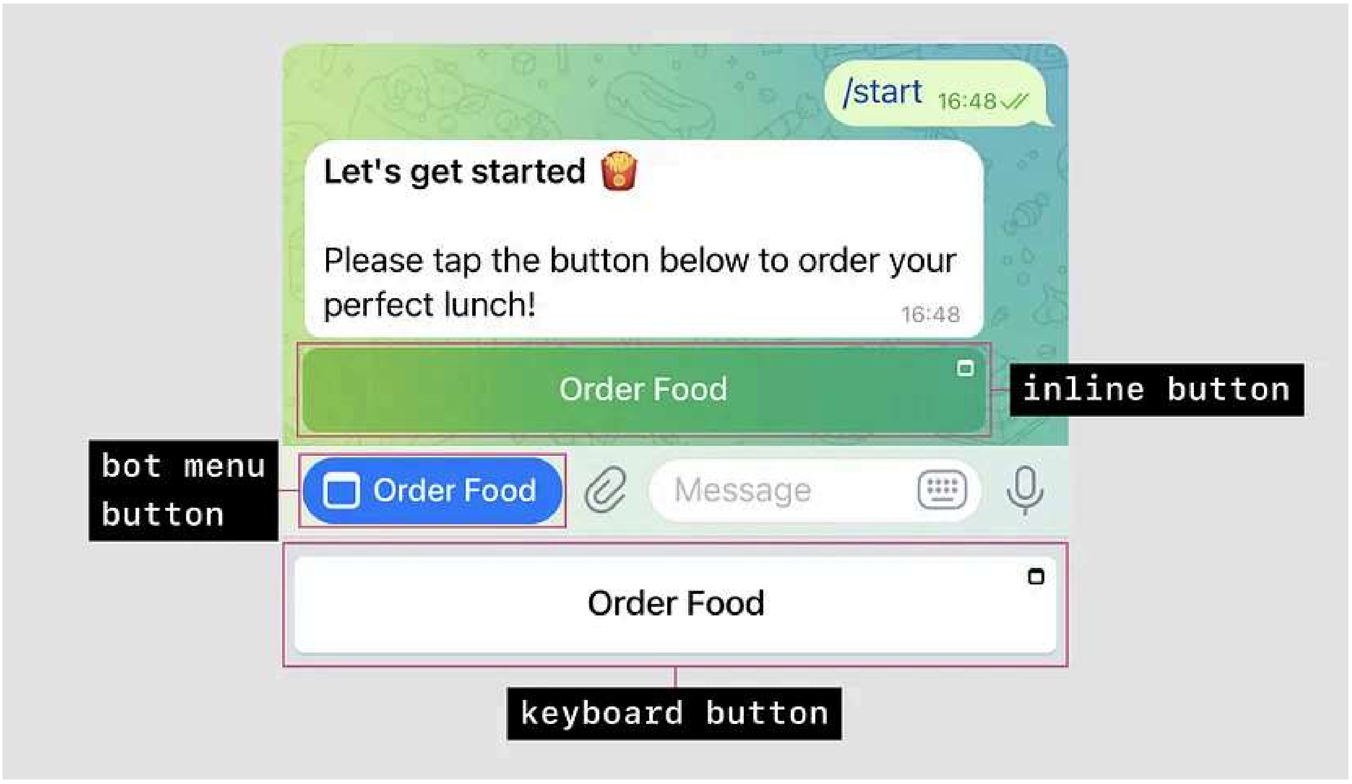
we can interact with our bot!


/start 21:33
I hear you loud and clear ! 21:33

Getting the bot to show a Web-App

I won't sugar-coat it, Web-Apps in Telegram aren't a walk in the park. For starters, there are three kinds of triggers to start web apps, and we will only be exploring one in this tutorial:

- **Inline or Menu button Web-Apps —** these are launched in Telegram, and then talk to an external server to handle anything the user does. Your bot must then handle any interaction with this third party to acknowledge to the user the results. (We won't be discussing them in this tutorial, they are more complex)
- **Keyboard button Web-Apps —** these are launched in Telegram and return data directly to the bot, without a need for a specialized server to handle the response from the user. **This is what we will be using.**

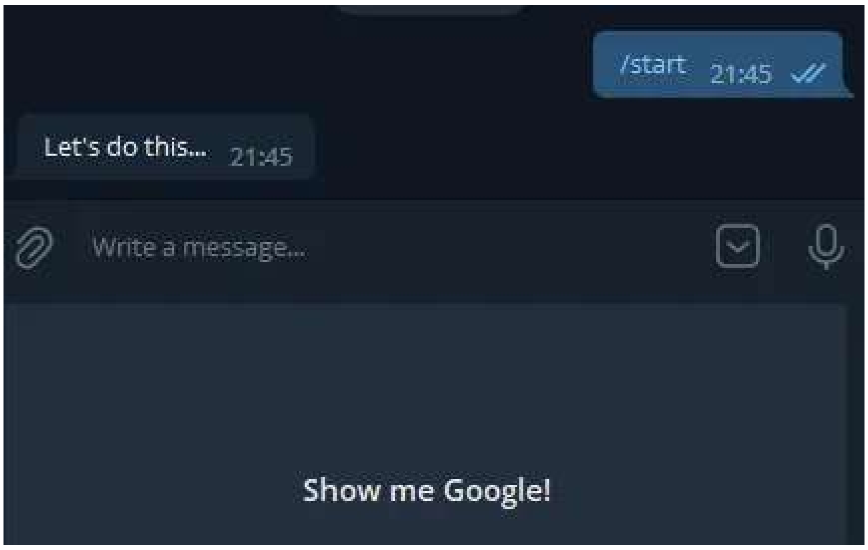


Three different types of buttons to launch Web-Apps (from https://core.telegram.org/bots/webapps)

First things first, let's just *proof-of-concept* this bad boy with a simple web-page. Update the *launch_web_ui* method in "app.py" to look like this:

```
async def launch_web_ui(update: Update, callback: CallbackContext):
    # For now, we just display google.com...
    kb = [
        [KeyboardButton("Show me Google!", 
web_app=WebAppInfo("https://google.com"))]
    ]
    await update.message.reply_text("Let's do this...", 
reply_markup=ReplyKeyboardMarkup(kb))
```

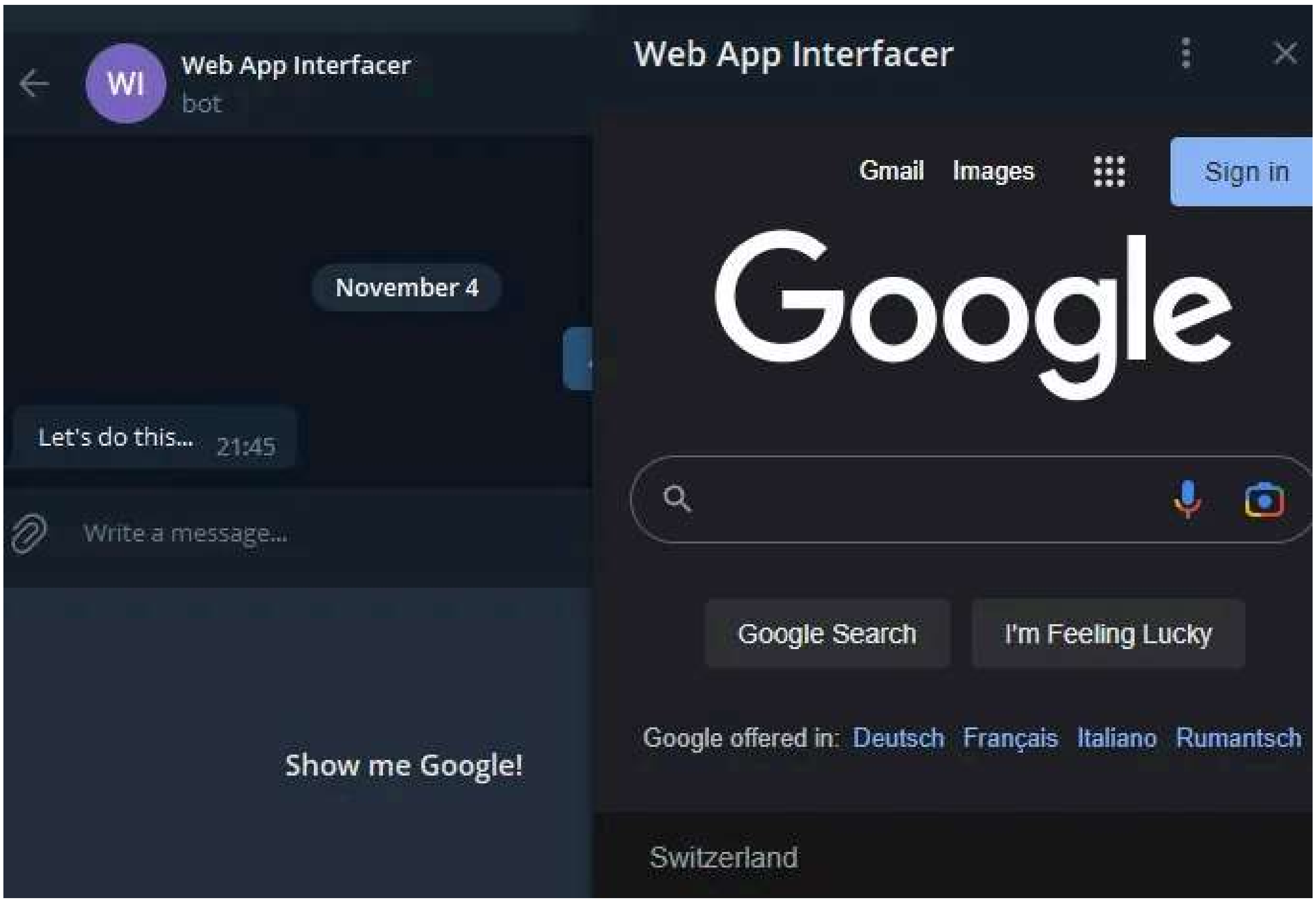
Let's talk to our bot again and see what happens:



/start 21:45

Let's do this... 21:45

And if we click the keyboard button…



Alright! This seems to be working nicely! Now, all we need to do is design our own web-app and display it instead of the google homepage!

Designing the Web-App

Let's do something quick and dirty using AngularJS.

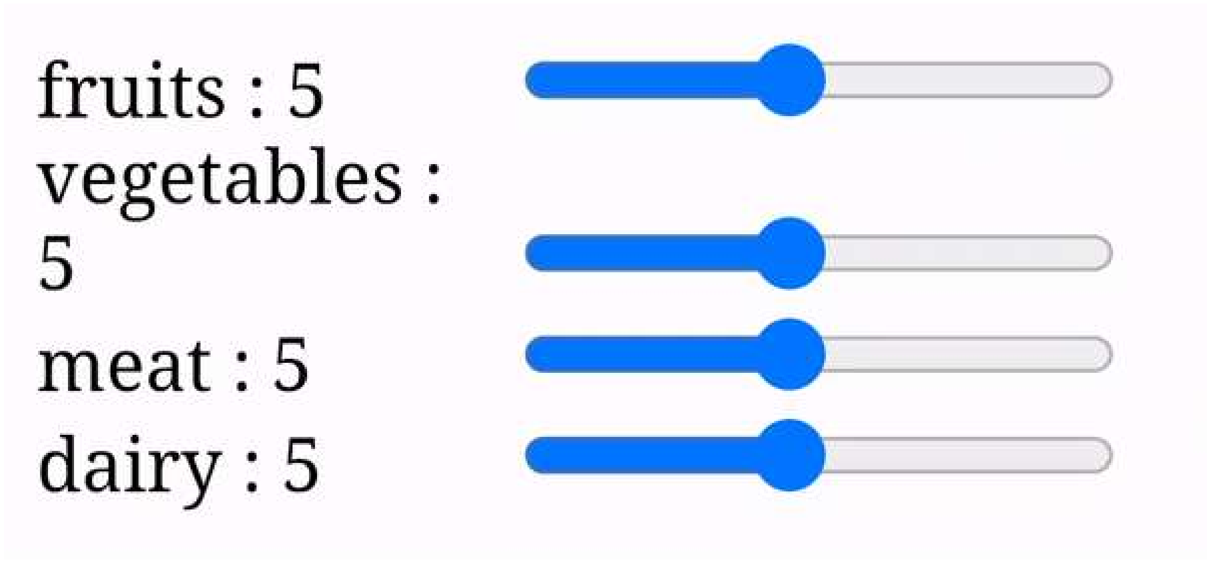
> ***Disclaimer** : Yes I know, AngularJS is outdated, but for tiny frontend projects like that, I actually really enjoy using it, kind of like some vamped up JQuery… For your project, the "how" doesn't really matter that much, all that matters is that you have a web-app that can send a "sendData" command with the inputted data. More below:*

Okay, I'll create a file called "index.html" and develop some simple interface to show you the basic idea:

```
<html ng-app="custom-webapp-ui" lang="en">
  <head>
    <!-- Load the Telegram Library -->
    <script src="https://telegram.org/js/telegram-web-app.js">
</script>
    <!--Load the AngularJS Library-->
    <script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.8.2/angular.m
in.js"></script>
    <script>
      //initialize the AngularJS stuff...
      angular.module("custom-webapp-ui",
[]).controller('CustomUIController', function
CustomUIController($scope) {
        //init our slider values that we will display
        $scope.foods = [
                { name: "fruits", value: 5 },
                { name: "vegetables", value: 5 },
                { name: "meat", value: 5 },
                { name: "dairy", value: 5 }
            ];
      });
    </script>
  </head>
  <body ng-controller="CustomUIController">
    <div ng-repeat="food in foods">
      <div style="width: 100px; display: inline-block">{{food.name}}
: {{food.value}}</div>
      <input style="display: inline-block" type="range" min="1"
max="10" ng-model="food.value" value="{{food.value}}">
```

```
      </div>
    </body>
  </html>
```

This gives us a very basic interface to work with :

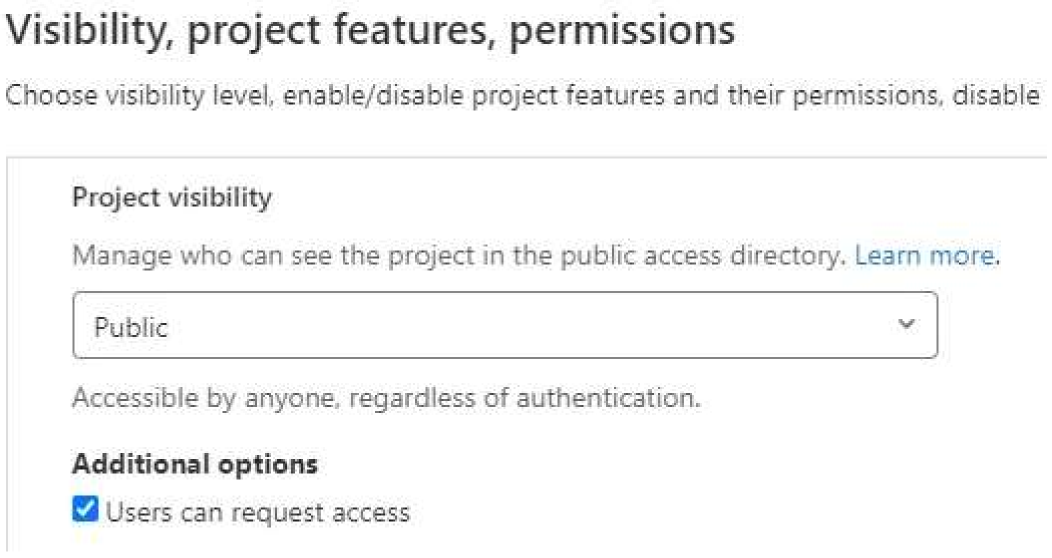


Now we bump into another issue, this needs to be hosted online somewhere…

Deploying the Web App online using GitLab Pages

> ***Note:*** *This approach is merely here to get results fast. Feel free to host your Web-App anywhere you like, and however you prefer! Just remember to use HTTPS, the chatbot will not work with HTTP pages.*

The easiest way to get this up and running ASAP is to use GitLab Pages. All you need is a (free) GitLab account, and to set up your repository there. Make sure the repository is public so that your web interface can be accessed by anyone (including your bot!)


Visibility, project features, permissions

Choose visibility level, enable/disable project features and their permissions, disable

Project visibility

Manage who can see the project in the public access directory. Learn more.

Public

Accessible by anyone, regardless of authentication.

Additional options

☑ Users can request access

You can check my repository for this example to compare notes if you need to.

> ***IMPORTANT:*** *do* ***NOT*** *add the .env file to the repository! If anyone but you gets a hold of the bot token, they can do whatever they want with it!*

Next, to ensure that our “index.html” is loaded into GitLab pages, we need to add a CI file. Create a “.gitlab-ci.yml” file in your project repository, and paste the following inside:

```
pages:
  stage: deploy
  script:
    - cp index.html public
  artifacts:
    paths:
      - public
  only:
    - master
```

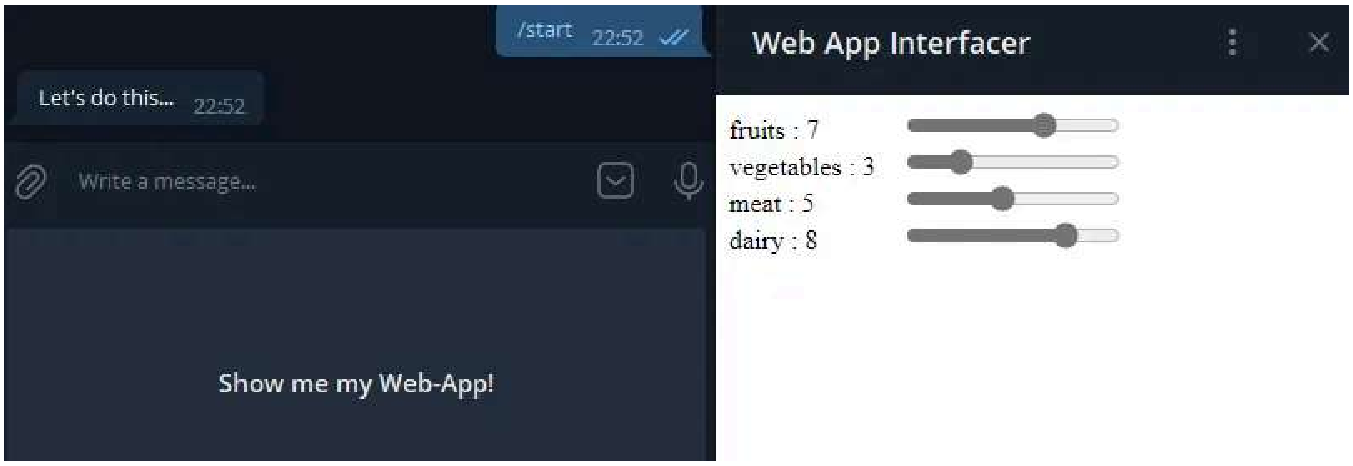
Now, when you push code to the master branch, it will make “index.html” available at https://<your-username>.gitlab.io/<project-slug>/ (which we can now access from the bot !)

Linking Everything Together

Alright this is getting interesting! If we get back to our python code, let’s just modify it a tad to show our Web-App instead of Google:

```
async def launch_web_ui(update: Update, callback: CallbackContext):
    # display our web-app!
    kb = [
        [KeyboardButton(
            "Show me my Web-App!",
           web_app=WebAppInfo("https://calixtemayoraz.gitlab.io/web-
interfacer-bot/") # obviously, set yours here.
        )]
    ]
    await update.message.reply_text("Let's do this...",
reply_markup=ReplyKeyboardMarkup(kb))
```

Let’s see the result:


/start 22:52

Let's do this... 22:52

Write a message...

Show me my Web-App!

Web App Interfacer

fruits : 7
vegetables : 3
meat : 5
dairy : 8

Awwwhhh yissss

Now, we are really **getting somewhere**!

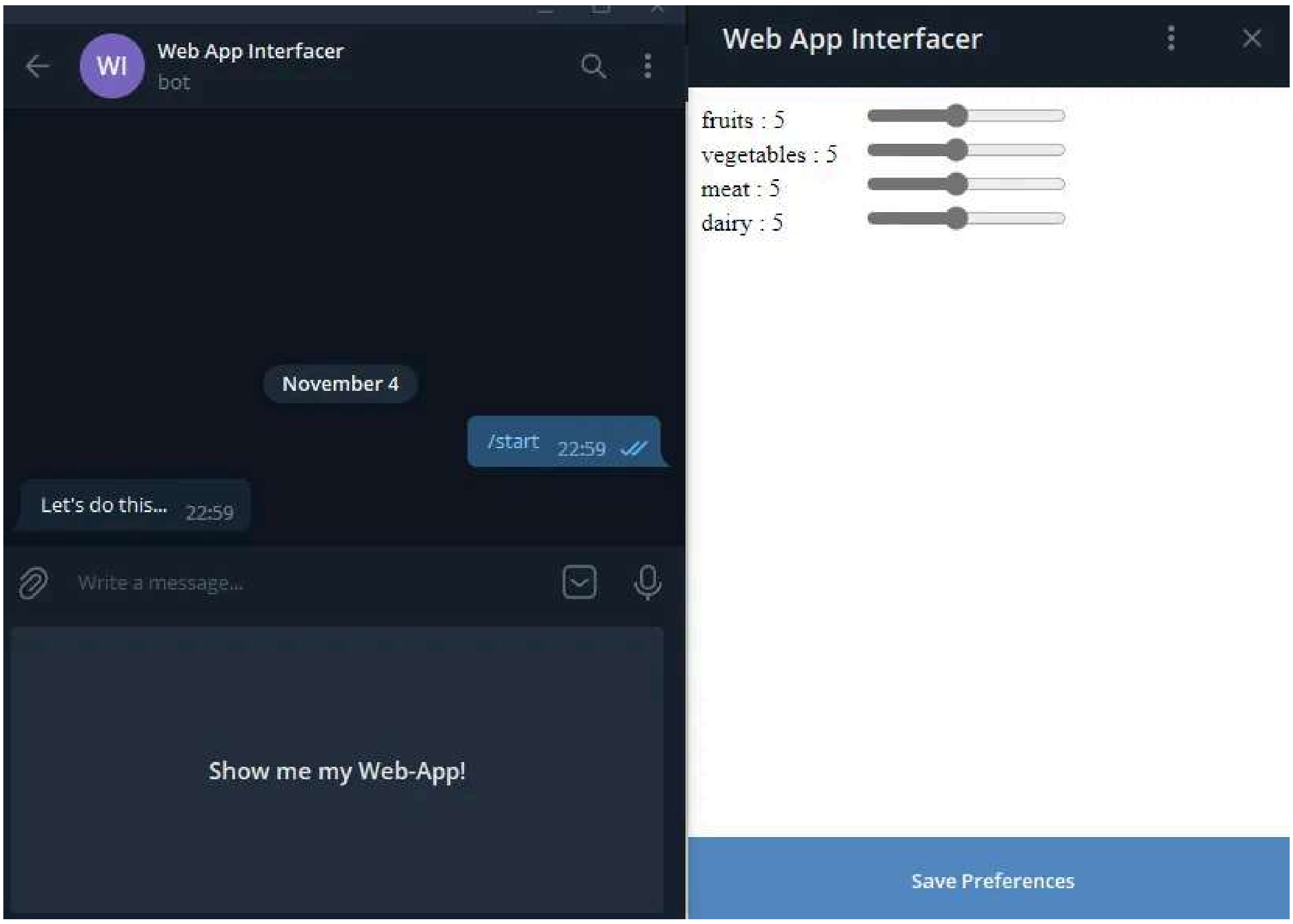
All we need now is to correctly send data back to the bot from the Web-App and to correctly parse it on the bot’s side.

Make the Web-App send data back to the bot

We just need to add a few things, namely the “submit” button (which telegram handily provides for us) and bind a function to send the data when it is pressed. If we modify the script in our “index.html” file:

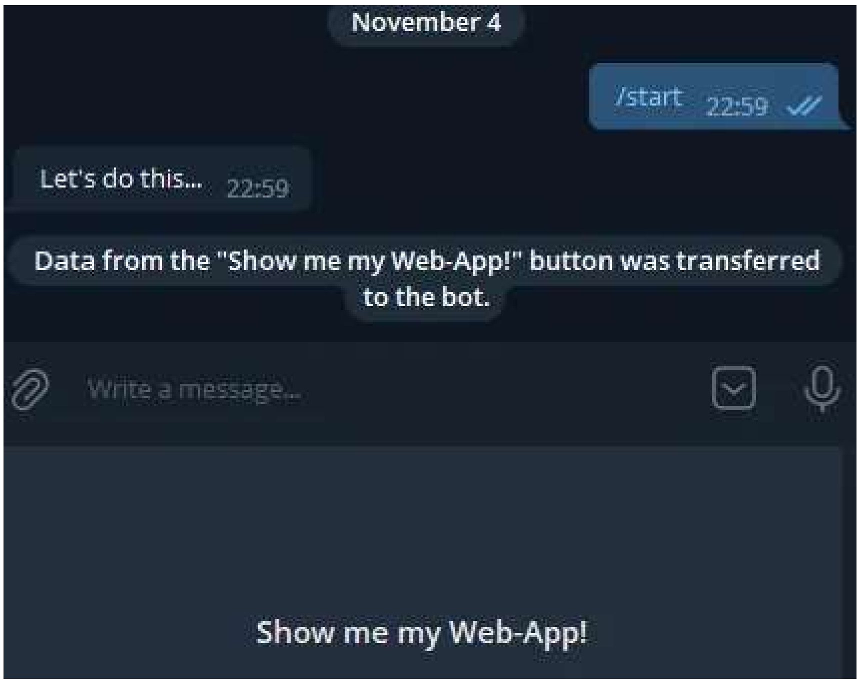
```
//initialize the AngularJS stuff...
angular.module("custom-webapp-ui",
[]).controller('CustomUIController', function
CustomUIController($scope) {
  //init our slider values that we will display
  $scope.foods = [
          { name: "fruits", value: 5 },
          { name: "vegetables", value: 5 },
            { name: "meat", value: 5 },
            { name: "dairy", value: 5 }
        ];

    //initialize the "save" button
    const mainButton = window.Telegram.WebApp.MainButton;
    mainButton.text = "Save Preferences";
    mainButton.enable();
    mainButton.show();

    // and make it send the "foods" object (as JSON string) back to
  the backend
    mainButton.onClick(function(){
      window.Telegram.WebApp.sendData(JSON.stringify($scope.foods));
    })
  });
```

And with these few lines of code, when we deploy our code and run the bot again…



We get a nice "save" button at the bottom!

Moreover, when we click the "save preferences" button:


November 4

/start 22:59

Let's do this... 22:59

Data from the "Show me my Web-App!" button was transferred to the bot.

Something got sent back! We just need to catch it.

I don’t know about you, but at this stage, I was getting tingles…

> ***It’s important to note** that any information you pass back from the UI to the bot needs to be serialized as a string (4096 bytes max) so any heavy stuff like images, etc should be handled differently.*

Handling the Response data from the bot’s side

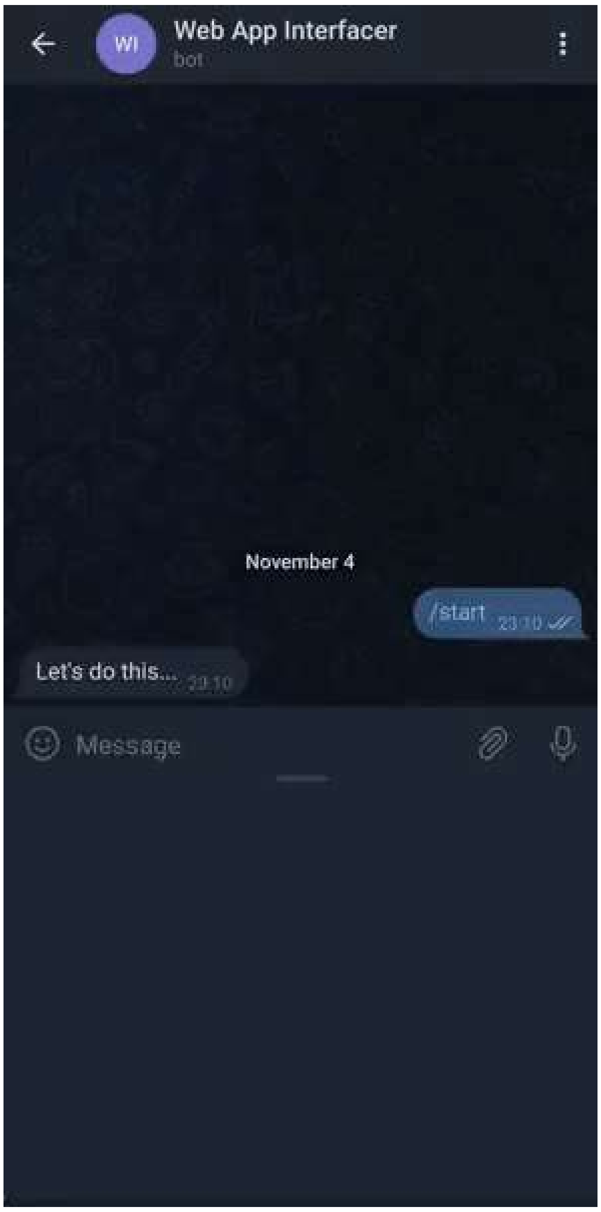
We are so close! All we need now is to handle a “service message” on the bot’s side to get the data sent back from the Web-App. Let’s add a simple method that just spits back the data sent from the user as a proof-of-concept:

```
async def web_app_data(update: Update, context: CallbackContext):
    data = json.loads(update.message.web_app_data.data)
    await update.message.reply_text("Your data was:")
    for result in data:
        await update.message.reply_text(f"{result['name']}:
{result['value']}")
```

and add a listener to trigger this method when we get a response from the web-app:

```
application.add_handler(MessageHandler(filters.StatusUpdate.WEB_APP_
DATA, web_app_data))
```

Finally:



November 4

/start 23:10

Let's do this... 23:10

There you go!

There you have it! End-to-end development of a very basic bot that opens a Web-App, allowing the user to enter info in a JavaScript environment and sending back the data to the bot. More specifically we saw:

- How to create a bot with botfather
- Make a simple reply to commands
- Display a web page from a KeyboardButton
- Design a basic UI for user input
- Send user inputted data back to the bot

- Parse that information in the bot to use afterwards

Hope this tutorial was helpful to you!

If you want, feel free to clone my repository for this project to get it running on your environment and pick it apart if you need.

Chatbot Development Python Web Development Web App Development Telegram Bot

Follow

Written by Calixte Mayoraz

20 Followers

Full Stack Developer, Data Scientist, Electronic Musician, Tinkerer. Endlessly curious about anything and everything.

More from Calixte Mayoraz

Calixte Mayoraz

Combining Deep Learning and Random Forests in Tensorflow

I've been working on a project for the last few months for anomaly detection, and have needed to classify Time-Sequences into various...

Feb 21, 2023

See all from Calixte Mayoraz

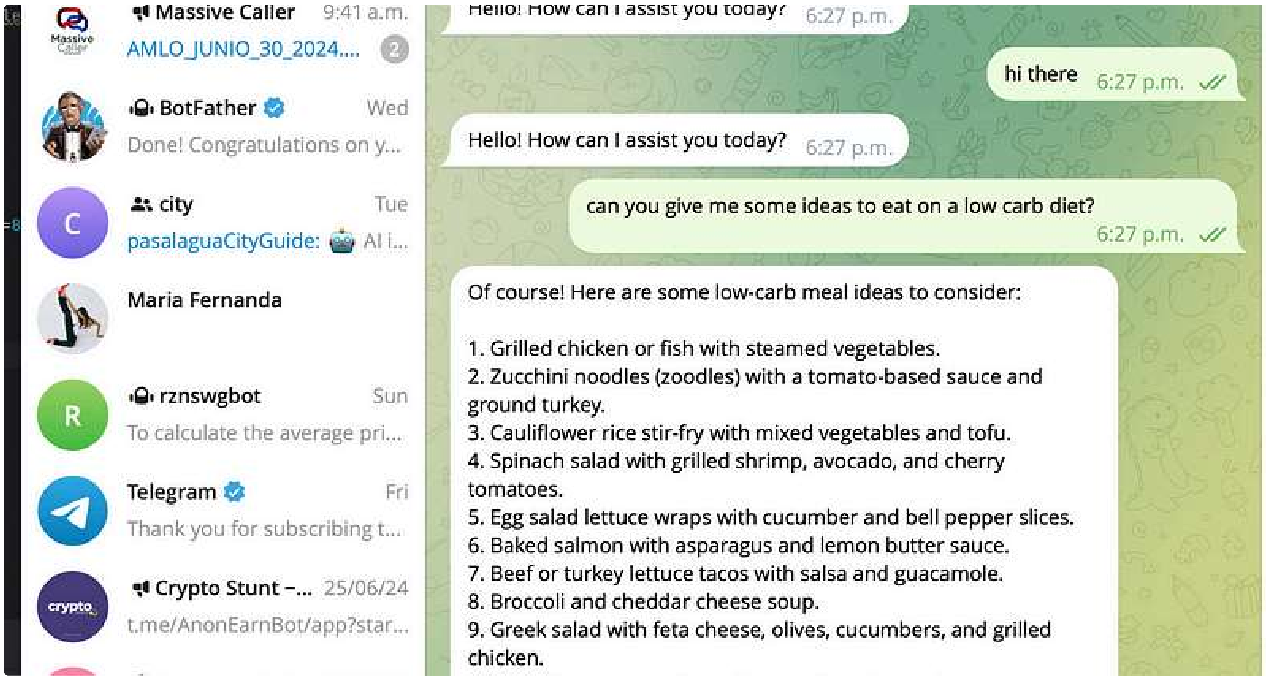
Recommended from Medium

Yusuf Fachroni

Building a Telegram Mini App with Flutter

What is Telegram Mini App?

Sep 25 63


Massive Caller 9:41 a.m.
AMLO_JUNIO_30_2024....
BotFather Wed
Done! Congratulations on y...
city Tue
pasalaguaCityGuide: AI i...
Maria Fernanda
rznswgbot Sun
To calculate the average pri...
Telegram Fri
Thank you for subscribing t...
Crypto Stunt -... 25/06/24
t.me/AnonEarnBot/app?star...

Hello! How can I assist you today? 6:27 p.m.

hi there 6:27 p.m.

Hello! How can I assist you today? 6:27 p.m.

can you give me some ideas to eat on a low carb diet? 6:27 p.m.

Of course! Here are some low-carb meal ideas to consider:

1. Grilled chicken or fish with steamed vegetables.
2. Zucchini noodles (zoodles) with a tomato-based sauce and ground turkey.
3. Cauliflower rice stir-fry with mixed vegetables and tofu.
4. Spinach salad with grilled shrimp, avocado, and cherry tomatoes.
5. Egg salad lettuce wraps with cucumber and bell pepper slices.
6. Baked salmon with asparagus and lemon butter sauce.
7. Beef or turkey lettuce tacos with salsa and guacamole.
8. Broccoli and cheddar cheese soup.
9. Greek salad with feta cheese, olives, cucumbers, and grilled chicken.

Juan Pasalagua

How to Create an Intelligent Telegram Bot with Python and ChatGPT

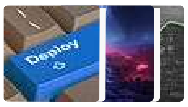
In this tutorial, I'll walk through the process of creating a Telegram bot that uses OpenAI's ChatGPT to respond to user messages. This bot...

Jul 4 75

Lists

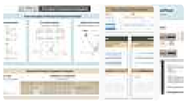
Coding & Development

11 stories · 886 saves



Predictive Modeling w/ Python

20 stories · 1634 saves

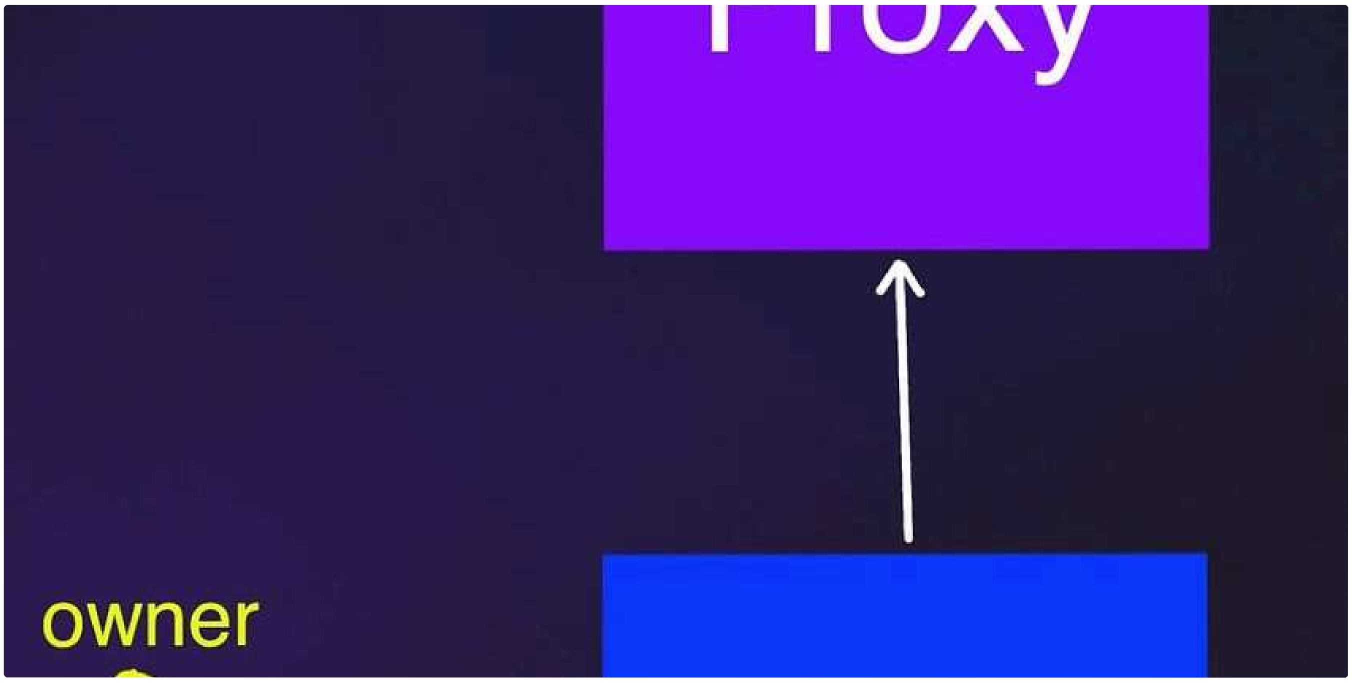


Practical Guides to Machine Learning

10 stories · 2003 saves

ChatGPT

21 stories · 860 saves



RareSkills

The Transparent Upgradeable Proxy Pattern Explained in Detail

The Transparent Upgradeable Proxy is a design pattern for upgrading a proxy while eliminating the possibility of a function selector clash.

Aug 22 22

luckyStars in CoinsBench

Chasing Alpha on the Solana Chain with Your Own Bot

Introduction

Jul 21 13

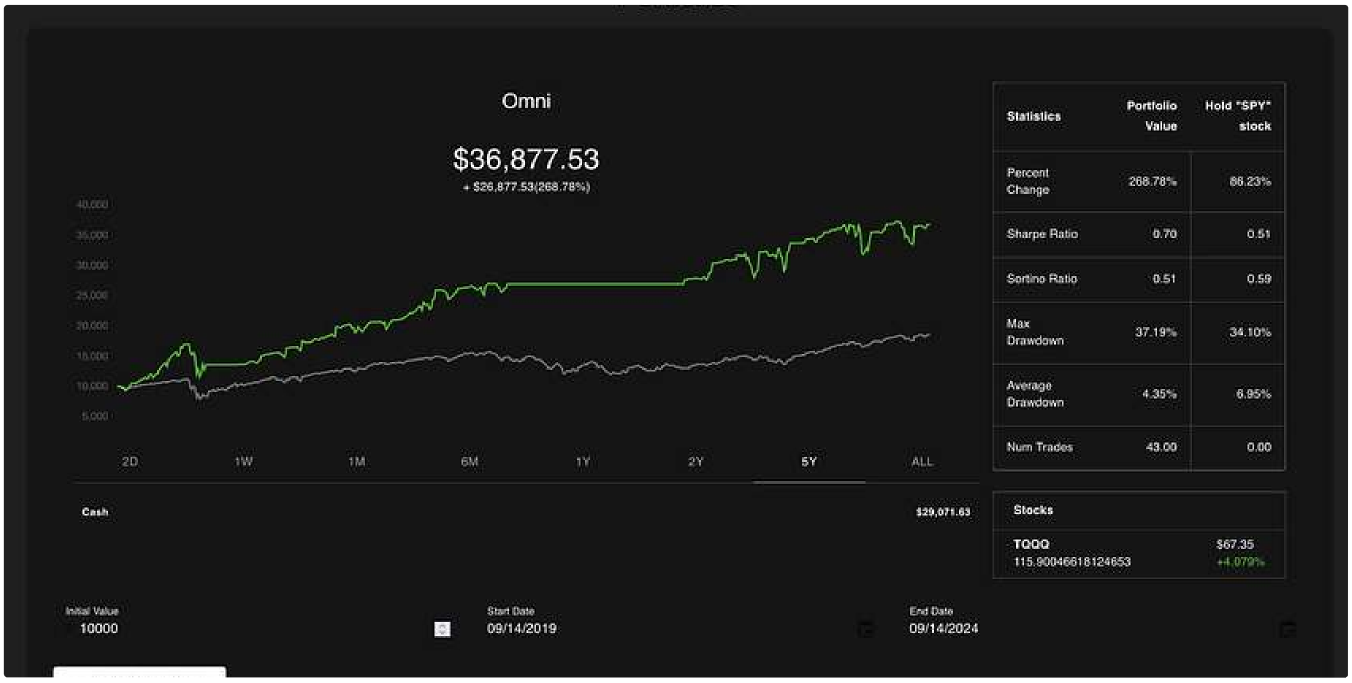
Always Free
24 GB RAM + 4 CPU + 200 GB
FREE
@harendraverma2 @harendra21 @harendra21

Harendra

How I Am Using a Lifetime 100% Free Server

Get a server with 24 GB RAM + 4 CPU + 200 GB Storage + Always Free

Oct 26 1.7K 23



Cash $29,071.63

Statistics	Portfolio Value	Hold "SPY" stock
Percent Change	268.78%	86.23%
Sharpe Ratio	0.70	0.51
Sortino Ratio	0.51	0.59
Max Drawdown	37.19%	34.10%
Average Drawdown	4.35%	6.95%
Num Trades	43.00	0.00

Stocks	
TQQQ 115.90046618124653	$67.35 +4.079%

Initial Value 10000

Start Date 09/14/2019

End Date 09/14/2024

Austin Starks in DataDrivenInvestor

I used OpenAI's o1 model to develop a trading strategy. It is DESTROYING the market

It literally took one try. I was shocked.

Sep 15 5.5K 135

See more recommendations