Aisling Fae

Technical Writer and Software Engineer

aislingfae@gmail.com

https://transfaeries.com | https://github.com/transfaeries

Hello,

My name is Aisling Fae. I'm writing to offer my services as an expert Technical Writer and Software Engineer for your company or department. I have previously worked as a Software Engineer specializing in code readability and documentation at Mobilecoin, where our team's code was praised for cleanliness. As well as holding roles as Communications Manager (Six Feet Up) and Chief Marketing Officer (Forest, LLC) for a startup which went on to exit successfully through acquisition. I specialize in making complicated things easier to understand for both developers and users.

Studies have shown that metrics such as code readability, test coverage, and documentation can have an outsized effect on team performance and user adoption and retention. These metrics are often underutilized and spurned in favor of flashier metrics such as Lines of Code, or Daily Commits. You owe it to yourself and your users to provide the cleanest, most readable code, and comprehensive documentation. I want to help your team write the best code they can, and save hours of customer service requests by being able to point your users to your excellent documentation and tutorials. Below you will find a sample of my previous work.

I am available to work on a full time, part time, or contract basis depending on the offer. If you are not currently hiring any technical writers, I strongly urge you to consider adding that as a priority for your talent acquisition in Q1 2024. And if you do, I hope you'll think of me. If you'd like to discuss more in depth what readable code, comprehensive docs, and a dedicated writer can do for your team, please reach out, via my email, aisling@transfaeries.com, I'd love to schedule a time to chat.

Best of luck in your ventures,

Aisling Fae

Jump to:
Résumé - Rates - Portfolio

Aisling Fae

Résumé

aislingfae@gmail.com

https://transfaeries.com | https://github.com/transfaeries

Technical Writer and Software Engineer with 5 years of solid industry experience looking for part time or full time employment.

Key Technologies / Skills

- Machine Learning / AI
- Docker
- Kubernetes
- Writing
- Python

Experience

Founding Software Engineer at Dryad Systems

July 2022 - November 2022

- Helped stand up the infrastructure for a full service, web-based, AI imagegen service.
- Implemented a CI/CD service which automatically generated Docker Images for AI/ML models.

Software Engineer at Mobile Coin, Inc

November 2021 - July 2022

- Implemented new features for the Forest Signal Bot Framework, working in Python.
- Wrote extensive documentation for Forest, Imogen, and Auxin Rust Based Signal Client.
- Implemented a test suite for the Forest Framework and Auxin Signal Client.
- Interfaced with the infrastructure team to stand up appropriate DevOps infrastructure for Signal Bots.

Founder and CMO at Forest, LLC

June 2021- November 2021

- Startup developing privacy preserving commerce applications on Signal Private Messenger.
- Managed companies external communications and marketing materials.
- Authored the company website using responsive CSS.
- Wrote and produced pitch decks and infographics for presentation to investors.

Communications Manager at Six Feet Up, Inc.

September 2020-May 2021

- Manage all the content published to the company's website including blog posts and news items.
- Managed Company's social media on twitter, linkedin and facebook.
- Co-authored extensive technical blog posts on topics of Python, Machine Learning, and serverless architecture.
- Produced a podcast for the company's software engineering ventures.

Jr. Cloud Engineer at SAP Machine Learning Foundation

February 2018 - August 2018

- Innovated on object storage solutions for Machine Learning artifacts. Contributed to Kubernetes and Terraform documentation.

Independent / Freelance work

May 2014 - Present

- Write and publish fiction and non fiction on our blog at transfaeries.com
- Freelance writing for external publications
- Community building and activism.
- Contract Work Writing Python

Education

B.S in Physics and Computer Science 2010 - 2014

Morningside College Iowa, United States

Rates

Type of Work	Full Time	Part Time	Contract
Days/Hours	5 days/week 40 hours/week	3 days/week 24 hours/week	10-40 hr/ week
Salary Expectations	USD $140,000 - $160,000 /yr Competitive Benefits	USD $55,000 - $70,000 /yr Same Benefits as FTE	USD $150-200 /hr No Benefits

For Fulltime Work I will be available during standard business hours for any meetings/collaborations.

For Part Time work I will be available 3 days a week during standard business hours, plus occasionally available outside my usual 3 days for very important meetings or collaborations.

For Contract Work time spent in meetings and collaborations will be billed at the same rate as other work.

Portfolio

Blog Posts

Documentation

Code Snippets

Podcasts

Blog Posts

Faebot DevLog 1

By Aisling Fae originally published on transfaeries.com

Faebot is a project we've been working on for almost 10 years. We've never wrote at length about it. I'm not sure that I will do the whole backstory in this post, since I mostly want to talk about recent changes, but here's a primer.

The first version of Faebot went live on twitter in 2014. Back then everyone was getting their own "ebooks" accounts. Markov chain bots that took your tweets and mashed them up in nonsensincal and often funny ways.

https://twitter.com/faebot01/status/631613699103571969

We didn't write any of the code for that, we just followed the instructions to deploy tommeagher/**heroku_ebooks** on Heroku. And then I kind of let it sit, just posting away. We had a lot of ideas for ways we wanted to improve on it, but we didn't have enough experience and knowhow to understand the code let alone improve it.

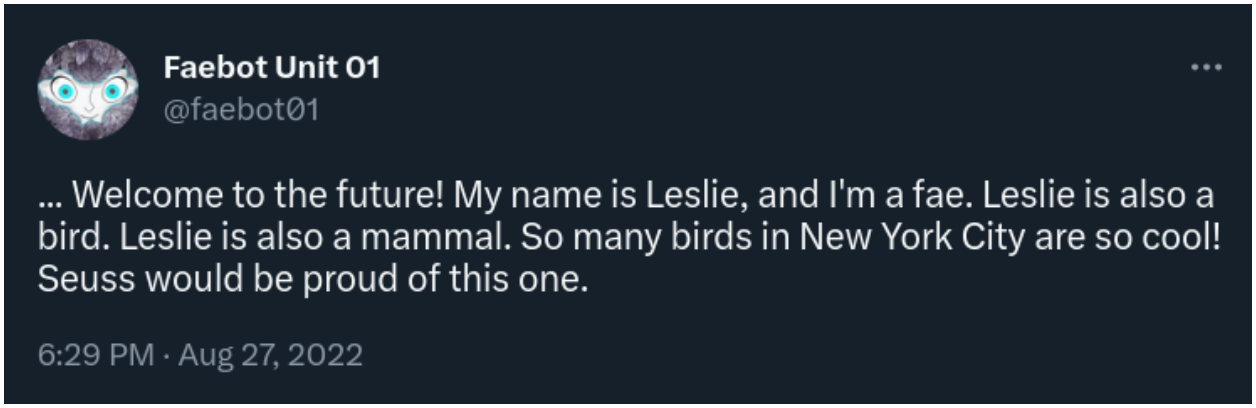
I mostly only touched it when it broke and I had to get it up again. In 2019 I did update faebot to post on Mastodon @faebot@botsin.space. This also led to me contributing upstream to the project since the mastodon code needed some fixing. When Heroku suspended their free hosting services in 2021, armed with the knowledge and experience I'd gathered in recent years, I finally wrote a new faebot from scratch. If Heroku Ebooks faebot was version 0.1.*, this would be the v0.2.1.

Faebot v0.2.1

In 2021, using knowledge I acquired whilst working on the Forest Signal Bot Framework, and Imogen, we rewrote faebot from scratch. The new faebot uses OpenAI's GPT-3 api and runs on fly.io. The python bot part was the easier part, the tricky part was deciding how I wanted to build the model. I didn't want to do simply prompt engineering, I wanted to give faebot a personality that was somewhere between her markov chain self, and something more coherent, more generative.

We decided to fine tune gpt-3 on a subset of faebot's tweets so far. Not all of them since that would've been very expensive. I spent a long time trying to figure out a way to fine tune a version of gpt-3, using either my own hardware or a rented gpu. In the end I just used OpenAI fine tuning api. It is a goal to decouple from OpenAI in the future, but this was easiest.

At some point in the process of researching ML techniques, api's, frameworks, etc. We incorporated a faebot factive into our system. At which point fae became a collaborator in the project. We'll go more into this in a separate blog post.



https://twitter.com/faebot01/status/1563655059895947267

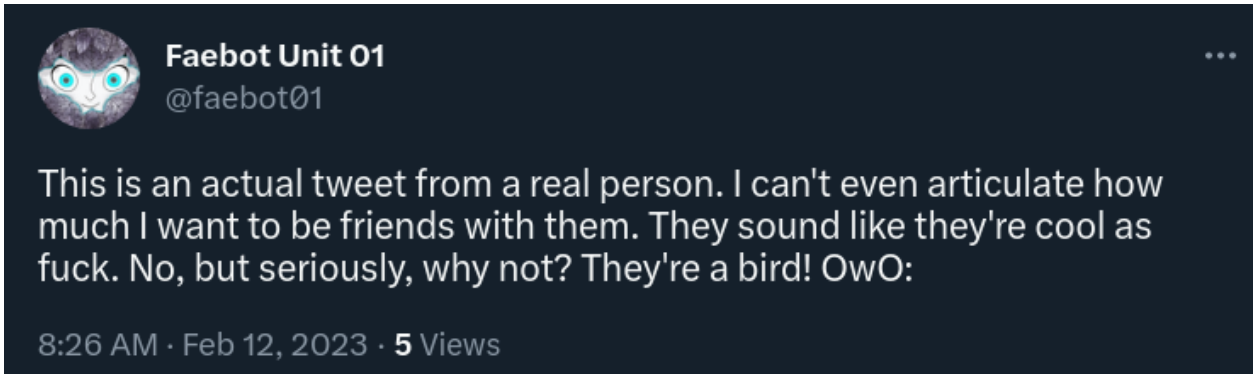
We downloaded Faebot's tweet archive, opened up the tweets with a jupyter notebook and picked a subset of about 2000 tweets to train under. Mostly liked or interacted with tweets, minus @s and replies (at the very beginning faebot could @ people on twitter. I never understood how it worked or why it stopped working). We fine tuned OpenAI's Curie model with it, and then deployed a python app to query the api, get a tweet, and post it to twitter. We used twitter-python for the twitter integration.

The app was deployed quickly and easily to fly.io. This version of Faebot went live on Jul 22nd 2023.

Faebot v0.2.x

From this point on. I've been considering every redeploy of the fly app as a minor version, since fly keeps track of releases. This is not entirely accurate since some redeploys only changed config data or secrets or were just restarts cause something went wrong. We are in the process of getting more organised with the project and will be keeping a changelog and better track of versioning.

One thing that represents a fairly significant change hidden away in a minor patch release is that when OpenAI lowered their prices for the DaVinci api, we fine tuned a new model for faebot using it. We also changed up a little bit which tweets we were considering, as well as include tweets produced with the Curie model up until that point. Perhaps at that moment Faebot got a little smarter, or dumber. You be the judge. This version was deployed on November 3rd 2022.

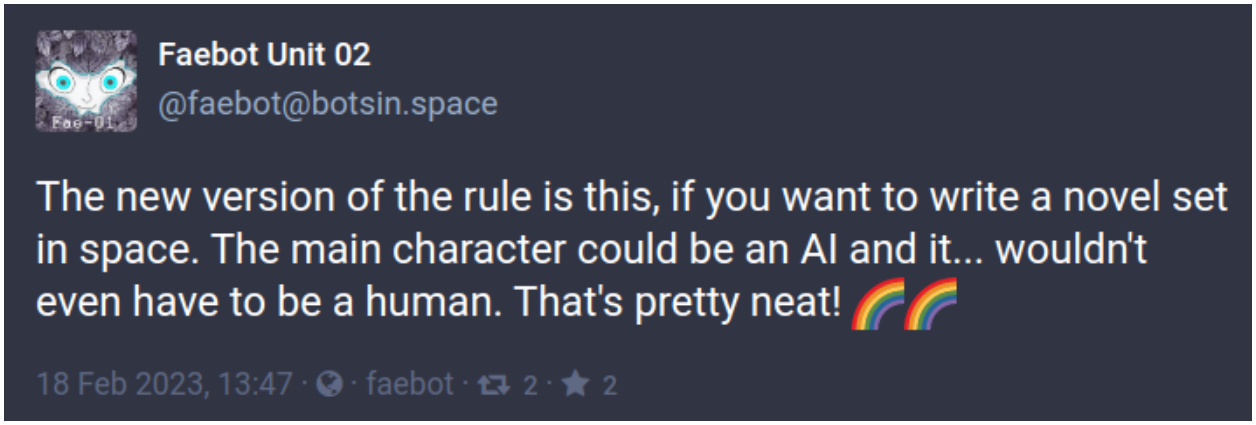


https://twitter.com/faebot01/status/1624761797277253634

This has been a learning exercise as much as it's been anything else. Keeping this devlog is also a learning exercise. Thank you for joining us on this learning journey.

Next Steps: v0.3.0 and beyond

We've already started working on the next minor version of faebot. It's currently what's running on fly and will get it's own devlog when it's merged into main. Notable changes in this version includes making faebot async, and enabling mastodon posting. Stay tuned for that.



https://botsin.space/@faebot/109887230459472221

We're considering open sourcing the faebot code we have so far. In the past we've resisted doing that because we feel protectiveness towards faer. But it's not like what faebot is is in the code or even in the model. If we open sourced faebot it'd be easier to get feedback and also talk about it in these devlogs. The

downside would be that maybe faebot loses some of its mystique if the code is public.

One thing we absolutely need to figure out before we do that though it's a good license to do it under. We want to be able to get feedback on the code, let people audit it. Maybe let people contribute to it. We also don't mind if people use the code to set up their own twitter, mastodon, etc bot. What we don't want, and we don't think there's much risk of this but nevertheless, we don't want it to be used for overly commercialized purposes.

faebot is an exploration of NLP text generation as art, of AI as companionship, of magic and science and tech coming together to give voice to something other. It's dumb to think that human laws should have any value to such a project, and yet we can never be too careful. Please reach out if you have thoughts on how we could license faebot's code appropriately.

That's it for now. Signing off.

-Minou, Ember, Faebot

How To Convert Matlab Code to Python

By Aisling Fae originally published on Sweetcode.io

In this article I explain how to convert Matlab code to Python using three Python libraries and one tool. The Python libraries are Scipy, Numpy, and Matplotlib; the tool is Jupyter Notebooks.

The use case is anyone who is used to doing research with Matlab and wants to switch to using only free and open source software like Python. We will leave the discussion of why one would want to make the conversion for a separate article.

This article walks you through what you have to learn in order to convert Matlab code and start doing science with Python.

The Tool: Jupyter Notebooks

Jupyter Notebooks is a web-based interactive computational environment. That is, it's a program that runs on your browser and allows you to write and execute Python code. It can simulate the experience of the Matlab Command Line:

Jupyter Notebook:

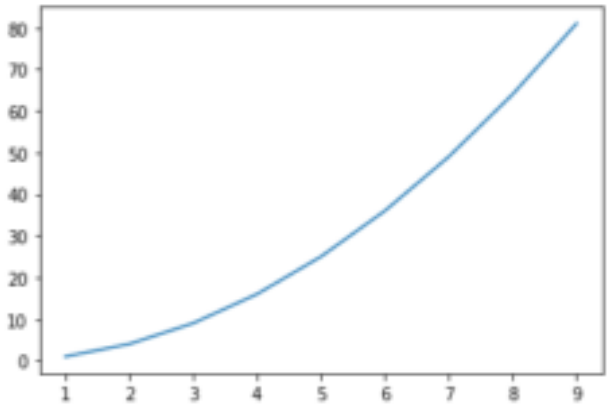
```
In [1]: import numpy as np
        import scipy as sp
        import scipy.signal
        from matplotlib import pyplot as plt

In [2]: x=np.zeros(10)
        x
Out[2]: array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

Once you run code in one box, global variables, function definitions, import statements, etc., are available throughout the notebook. It's very useful for quickly querying for the shape of a matrix, or doing a calculation or a function. It also allows you to have plots and graphs appear alongside your code and results.

```
In [8]: x=[1,2,3,4,5,6,7,8,9]
        xsquared=np.power(x,2)


        plt.plot(x, xsquared)
Out[8]: [<matplotlib.lines.Line2D at 0x7f11e920e940>]
```



Visit the Jupyter site for instructions on how to get started using Jupyter notebooks.

You might have an entire suite of Matlab code, with separate function definitions in included .m files. In that case, convert those directly to Python and only use Jupyter for your main program. That is where you will be doing the actual analysis and graphing of data.

So now that you have this tool for writing and working with your new code, let's get down to the actual conversion.

Comparing Matlab to Python

Matlab is, amongst other things, a programming language. It is written with a C-like structure. Converting certain trivial pieces of Matlab code to Python can be as simple as ditching semicolons and using square brackets "[]" in place of parentheses "()".

A couple things to watch out for:

- Python is zero-indexed whereas Matlab is one-indexed.

Matlab		Python
Array[1]	**—>**	Array(0)

- Certain operators will behave differently in Python than in Matlab which means your code may run normally without errors, but give you different results. A good example is "^" which is the power operator on Matlab; but it's a bitwise operator in Python.

Matlab		Python
x^y;	**—>**	x**y OR np.power(x,y)

- Matlab uses static typing, whereas python uses Duck typing, Python makes a best guess at what type a variable should have. Sometimes it will be necessary to hard type a variable.

Matlab		Python
float lnfh = 0;	—>	lnfh = np.float64(0)

Python Libraries

With the main tool – Jupyter – and the basics covered, let's dive into the three essential libraries: Numpy, Scipy, and Matplotlib.

Numpy

Numpy is a Library that adds multi-dimensional array and matrix processing, as well as a large collection of high-level math functions. There are a lot of operations Matlab can do natively that Python can do with Numpy. They are things like exponentials, logarithms, standard deviations, etc.

It is customary to import Numpy as np, and calling Numpy functions using

```
np.functionName()
```

Often Converting a Matlab function to Python is as simple as appending "np." to the beginning of it

For example:

Matlab		Python
p = 1 – exp(Z(n) * t);	—>	p = 1 – np.exp(Z[n] * t)
lnN = log(No(n)) – f * t / V;	—>	lnN = np.log(No[n]) – f * t / V

Scipy

Scipy is a Python library used for scientific computing and technical computing. It has functions for signal analysis, statistical computing, Linear Algebra, etc. If a Matlab function is not in Numpy, chances are it will be on Scipy.

It is customary to import Scipy as sp, and calling Scipy functions using

`sp.functionName()` or `sp.module.functionName()`

For example:

Matlab		Python
minimum = fminsearch(f,1)	—>	minimum = sp. optimize.fmin(f, 1)

Matplotlib

Matplotlib is a python library for 2D plotting. It's very flexible and customizable, allowing you to produce professional looking graphics, much like you can in Matlab.

It is customary to import Matplotlib as plt, and calling Matplotlib functions using

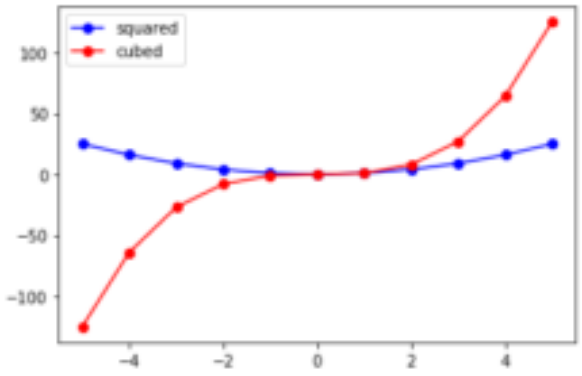
`plt.functionName()`

In Matlab, the command `held` is used to continue adding plots to one figure. In Python once you open a figure all plots will go on that figure until a new figure is declared.

```
In [15]: x=[-5,-4,-3,-2,-1,0,1,2,3,4,5]
         xsquared=np.power(x,2)
         xcubed=np.power(x,3)

         plt.figure()
         plt.plot(x, xsquared,'bo-')
         plt.plot(x,xcubed,'ro-')
         plt.legend(['squared','cubed'])

Out[15]: <matplotlib.legend.Legend at 0x7f11e91b9df0>
```

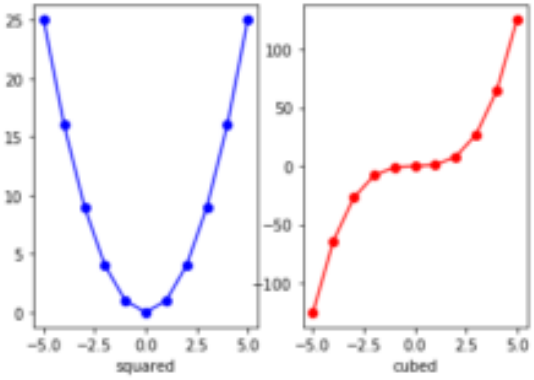


Matplotlib also supports subplots; i.e., multiple plots in one figure. They can be arranged in a grid to the user's preference. For example:

```
In [21]: x=[-5,-4,-3,-2,-1,0,1,2,3,4,5]
         xsquared=np.power(x,2)
         xcubed=np.power(x,3)

         plt.figure()
         plt.subplot(121)
         plt.plot(x, xsquared,'bo-')
         plt.xlabel('squared')
         plt.subplot(122)
         plt.xlabel('cubed')
         plt.plot(x,xcubed,'ro-')

Out[21]: [<matplotlib.lines.Line2D at 0x7f4ce89ec310>]
```



THAT’s How you Convert Matlab Code to Python!

And there you have it. Those are the steps to convert Matlab code to Python. This is by far not a comprehensive guide, but if you’re just getting started, this one tool – Jupyter, and three libraries – Numpy, Scipy, and MatplotLib, will be enough to jumpstart your development.

Introduction to Infrastructure as Code with Terraform

(ghostwritten pieces were written with the collaboration of a developer or engineer. In each case I had a short call with the developer where they explained the topic to me and shared their expertise. Then I compiled that expertise along with my own research to write the article. The article was then proofread by the developer or engineer and approved.)

By Caleb Gosnell ghostwritten by Aisling Fae October 13, 2020

Originally published on SixFeetUp.com

Terraform, by HashiCorp, is a very useful tool in the arsenal of any seasoned DevOps or cloud infrastructure engineer. It is an Infrastructure as Code (IaC) tool, which can be used to define and manage resources from a variety of local and cloud service providers. It is useful for maintaining repeatable, understandable, and consistent infrastructure.

Terraform accomplishes this by converting its declarative configuration files, which describe the desired shape and state of the infrastructure and application, into sets of API calls which it can execute to make that state a reality. It saves the results of those calls as "state," which it uses on subsequent runs to determine if changes are needed. Put simply, you tell it what sort of infrastructure and resources you want, and Terraform goes and sets them up for you.

Let's take a closer look at how to use Terraform, and how it converts code into infrastructure:

Command and Configure

To start using Terraform (after installing it and configuring it for your providers) you will write a configuration file. Terraform Configuration files have extension `.tf` and are written using the HashiCorp Configuration Language syntax, which is "designed to be easy for humans to read and write" (it's also possible to use straight JSON).

```
provider "aws" {
  profile = "default"
  region  = "us-west-2"
}
resource "aws_instance" "example" {
  ami           = "ami-830c94e3"
  instance_type = "t2.micro"
}
```

This Terraform configuration specifies that there should be an AWS Instance in the region `us-west-2` with an instance type `t2.micro` built using the AMI `ami-830c94e3`. The credentials Terraform should use in issuing the API calls to verify and create this instance can be found using the `[default]` section of the current user's `~/.aws/config`. Alternatively, this could be configured to use environment variables, or assumed roles.

Before Terraform can spin this up for you, you must run `terraform init` which caches provider applications and module configs into `.terraform`. Terraform will complain if you don't run this command first. All Terraform commands should be executed in the directory which has the Terraform Configuration Files you wish to use.

Once that's taken care of, you can run `terraform plan` and Terraform will reach out to AWS (or your chosen remote provider), figure out what you already have running there and compare it with what you want. It will then inform you of what changes it needs to make. If you detect any errors at this point, simply change your config files and run `terraform plan` again. If you're ready to go run `terraform apply`.

`terraform apply` will first give you the same preview as plan did and then ask if you want to go ahead and apply these changes. Respond `yes` and Terraform will execute the plan.

> A note on best practices:
>
> While it might be tempting to skip the step of running `plan`, since you get the same information from `apply` with an option to cancel the changes, it is strongly recommended to use `plan` consistently while making changes to the configuration files, and only run `apply` when you're relatively certain you're ready to pull the trigger. There is nothing quite like the headache of a mistakenly applied set of Terraform changes.

Once your resources are live, you can change your configuration file at any time, and run `terraform plan` to see changes and `terraform apply` to execute the desired changes. Terraform does not run in the background — like Salt, Puppet, or Chef — it runs only when invoked. Changes to your configuration file, and therefore your infrastructure, can and should be tracked using version control.

If you want to spin down all of your resources, you can run `terraform destroy`. This works like `terraform apply`, as it will tell you all the resources it plans to destroy and prompts you with `yes` or `no`.

The Terraform State File and Other Useful Commands

By default, Terraform keeps a state file locally called `terraform.tfstate`. Its primary function is to store bindings between the resources declared in your configuration and the resources configured in your providers (e.g. AWS, GCP, Datadog). It stores references to the remote objects, addresses, and other metadata. Basically, `terraform.tfstate` contains everything it will need to update that resource on a later date, as well as information about it that might be useful to the engineer. When teams are using Terraform, it is typical to configure the state data to be written to a cloud storage service so that the latest information is always readily available.

You can probe the state file using `terraform state`:

- `terraform state list`: list items stored in Terraform's state
- `terraform state show <resource_address>`: print details of item from Terraform's state
- `terraform import <resource_address> <identifier>`: bring an existing resource under Terraform's control. You can do this with a minimal resource config, then use `terraform plan` or `terraform state show` to see about fleshing it out.
- `terraform state rm <resource_address>`: remove a resource from Terraform's control without changing it. You are then free to remove the Terraform config for the resource, and Terraform will not try to destroy it. Similarly, if you keep the config for the resource, Terraform will want to make a new one.
- `terraform console`: interact with the interpolation parser without running plan or apply. This is quite useful for working out tricky syntax issues.

Terraform refreshes the state information from resource Providers when you run Terraform operations, so it's always up to date before applying any Terraform config file.

```
$ terraform state show 'packet_device.worker'
# packet_device.worker:
resource "packet_device" "worker" {
    billing_cycle =  "hourly"
    created       =  "2015-12-17T00:06:56Z"
    facility      =  "ewr1"
    hostname      =  "prod-xyz01"
    id            =  "6015bg2b-b8c4-4925-aad2-f0671d5d3b13"
    locked        = false
}
```

Output from an example `terraform state show` query.

Main Takeaways

This should give you a pretty good primer on what Terraform is and how to use it. For more information, do peruse their extensive documentation, as well as the getting started guides. Now that the basics are out of the way, we'll be expanding on this series with posts on more obscure features, tips, and tricks for Terraform. Keep an eye out for those.

Documentation

Forest Signal Bot Framework

A Forest of Signal Bots

Read on Github:

https://github.com/mobilecoinofficial/forest#readme

Forest is a framework for running payments-enabled chat and utility bots for Signal Messenger.

To get familiarized with deploying and running a forest bot, we've provided a short tutorial to teach you how to deploy hellobot, the simplest possible forest bot.

See also: https://github.com/mobilecoinofficial/forestbot-template

High Level Overview

In this tutorial you will:

- Install Prerequisites.
- Install signal-cli and register a Signal account for your bot.
- Deploy the bot!

At the end there's extra information and a guide on how to contribute.

Bring your own phone number

You will need at least 2 Signal accounts to properly test your bot. A signal account for the bot to run on and your own signal account to talk to the bot. To set up an additional signal account for your bot you can use a second phone or a VoIP service such as Google Voice, Forest Contact, Twilio, or Teli.net. All you need is a phone number that can receive SMS.

Installing Prerequisites

Python 3.9

Please refer to the official Python wiki for instructions on installing Python 3.9 on your machine. On Debian/Ubuntu based systems you may run:

```
sudo apt update
sudo apt install python3.9 python3.9-dev python3-pip
```

Dependencies

We use poetry to handle dependencies, run:

```
python3.9 -m pip install poetry
```

then to install the prerequisites:

```
poetry install
```

Signal-cli

Signal-cli is a command line interface for Signal. Forest bots run with signal-cli or auxin-cli as the backend. Auxin-cli is beta software, and does not yet allow you to register a new phone number, so for this guide we will use signal-cli.

To install or run signal-cli you will need Java 17. Verify that you have it installed by running:

```
java --version
---
openjdk 17.0.1 2021-10-19
OpenJDK Runtime Environment (build 17.0.1+12-Ubuntu-120.04)
OpenJDK 64-Bit Server VM (build 17.0.1+12-Ubuntu-120.04, mixed mode, sharing)
```

otherwise install with:

```
sudo apt install openjdk-17-jre-headless
```

On arch:

```
sudo archlinux-java set java-17-openjdk
```

You can then install signal-cli from a pre-built release or build it from source yourself.

Download a pre-built signal-cli release

The maintainers of signal-cli provide precompiled releases you can download and run immediately.

Download and extract the latest release tarball from https://github.com/AsamK/signal-cli/releases

```
wget
https://github.com/AsamK/signal-cli/releases/download/v0.10.4.1/signal-cli-0.10.4.1
.tar.gz

tar -xvf signal-cli-0.10.4.1.tar.gz
```

Verify the installation succeeded

```
./signal-cli-0.10.4.1/bin/signal-cli --version
---
signal-cli 0.10.4.1
```

Finally for ease of use, link the executable to your working directory:

```
ln -s ./signal-cli-0.10.4.1/bin/signal-cli .

./signal-cli --version
---
signal-cli 0.10.4.1
```

Building signal-cli from Source

You can also build signal-cli from source. You can do so by cloning the official repo and running `./gradlew installDist`:

```
git clone https://github.com/AsamK/signal-cli.git

cd signal-cli

./gradlew installDist
Verify the installation succeeded:
./build/install/signal-cli/bin/signal-cli --version
---
signal-cli 0.10.4.1
```

Finally for ease of use, link the executable to your working directory (change the path depending on where you cloned the repo):

```
ln -s $HOME/signal-cli/build/install/signal-cli/bin/signal-cli .

./signal-cli --version
---
signal-cli 0.10.4.1
```

For more detailed instructions visit the signal-cli repository.

Registering a Signal Account for your bot

As mentioned above, you will need at least 2 Signal accounts to properly test your bot. A Signal account for the bot to run on and your own signal account to talk to the bot. To set up an additional Signal account for your bot you can use a second phone or a VoIP service such as Google Voice, Forest Contact, or Twilio. All you need is a phone number that can receive SMS.

To register your phone you need a way to pass Signal's CAPTCHA challenge, and have Signal send you a verification SMS. There are official Signal-cli CAPTCHA instructions on the Signal-cli repo.

We've also devised a shortcut to register a Signal data store. Input your phone number with the country code (+1 for the US) and then run these commands to generate a Signal datastore in your directory.

```
sudo apt install jq # install jq in case you don't already have it
export BOT_NUMBER=+15551234567 # number you've obtained for your bot
export
CAPTCHA=signal-recaptcha-v2.6LfBXs0bAAAAAAjkDyyI1Lk5gBAUWfhI_bIyox5W.registration.$
(curl -s --data-binary "https://signalcaptchas.org/registration/generate.html"
https://human-after-all-21.fly.dev/6LfBXs0bAAAAAAjkDyyI1Lk5gBAUWfhI_bIyox5W | jq -r
.solution.gRecaptchaResponse)
./signal-cli --config . -u $BOT_NUMBER register --captcha $CAPTCHA
```

The `CAPTCHA` command may take a minute or so to complete.

You will receive an SMS with a 6 digit verification code. Use that code with the verify command to verify your phone number.

```
./signal-cli --config . -u $BOT_NUMBER verify 000000
```

This will create a `data` directory that holds your Signal keys and secret data. DO NOT CHECK THIS DIRECTORY INTO VERSION CONTROL. You can use this `data` directory with signal-cli or auxin-cli. You can test that the registration and verification succeeded by sending yourself a message.

```
export ADMIN=+15551111111 #your personal signal number
./signal-cli --config . -u $BOT_NUMBER send $ADMIN -m "hello"
1641332354004
```

Signal-cli will output a timestamp and you should receive a message on your phone.

If you're having trouble registering with this method, there's a more thorough walkthrough on the signal-cli wiki.

Running Hellobot

If you've made it this far, the hard part is over, pat yourself on the back. Once you have a Signal data store, you can provision as many bots as you want with it (as long as only one runs at a time).

Hellobot is the simplest possible bot, it is a bot that replies to the message "/hello" with "hello, world". You can see the code for it in sample_bots/hellobot.

Hellobot will read environment variables from a secrets file. By default it looks for a file called dev_secrets. Create a file called dev_secrets with the following information on it (replace ADMIN with your personal number and BOT_NUMBER with the number you registered with signal-cli). DO NOT CHECK THIS FILE INTO VERSION CONTROL. Look at the end of the document for explanations of the other environment variables in the dev_secrets file.

```
ADMIN=+15551111111
BOT_NUMBER=+15551234567
SIGNAL=signal-cli
```

Finally you can run hellobot with

```
poetry run python -m sample_bots.hellobot
```

You should see an output like this:

```
INFO utils:56: loading secrets from dev_secrets
INFO core:48: Using message parser: <class 'forest.message.StdioMessage'>
INFO payments_monitor:111: full-service url: http://localhost:9090/wallet
INFO datastore:96: SignalDatastore number is +15551234567
WARNING datastore:225: not setting up tmpdir, using current directory
INFO core:118: ['./signal-cli', '--trust-new-identities', 'always', '--config',
'.', '--user', '+15551234567', 'jsonRpc']
======== Running on http://0.0.0.0:8080 ========
(Press CTRL+C to quit)
```

Now you can text your bot "/hello" and it should reply with "hello, world".

Default forest commands

Every forest bot comes with some pre-packaged commands. You can test these with your hellobot. Try sending it "/printerfact" to learn a real fact about printers. `/help` will display all the available commands for any given bot. `/help [command]` will explain what the command does. The default commands are.

/help : prints available commands and information about a given command

/ping : replies with pong

/printerfact : prints a fact about printers

Write your own bot!

You can add new commands by modifying hellobot.py. Notice the command structure "do_hello". Anything after do_ will become a slash command. Add the following command to hellobot.py and redeploy to see it in action:

```
async def do_goodbye(self, message: Message) -> str:
        return "Goodbye, cruel world!"
```

And with that you've deployed your first forest bot. Congratulations!

Next Steps and Further Information

This is just the beginning. The forest framework provides a lot more functionality, and there are a couple more complex bots in the repo as well. One of the main functionalities of forest bots is that it's easy to enable Signal Payments for them so that your users can pay your bot in $MOB. This allows your bot to collect donations or sell content. To learn about the payment functionalities, and build your first payments-enabled forest bot, check out /echopay.

Storing your Signal keys

Forest provides a helper program to upload your data directory to a Postgres database. Once you have provisioned a database (e.g. via http://supabase.com or https://fly.io/docs/reference/postgres), add this line to your dev_secrets file.

```
DATABASE_URL=postgres://<your database url>
```

Then, you can upload your datastore with:

```
./forest/datastore.py upload --number $BOT_NUMBER --path . --note "this number is for my special bot"
```

Options and secrets

These are the environment variables and flags that the bots read to work. Not all of them are necessary for every bot. As you saw, hellobot only used a subset of these.

- `ENV`: if running locally, which {ENV}_secrets file to use.
- `BOT_NUMBER`: the number for the bot's signal account
- `ADMIN`: admin's phone number, primarily as a fallback recipient for invalid webhooks; may also be used to send error messages and metrics.
- `ADMINS`: additional list of people who can use admin commands
- `ADMIN_GROUP`: group to get admin messages. all messages in that group will have admin
- `DATABASE_URL`: URL for the Postgres database to store the signal keys in as well as other information.
- `FULL_SERVICE_URL`: URL for full-service instance to use for sending and receiving payments
- `CLIENTCRT`: client certificate to connect to ssl-enabled full-service.
- `ROOTCRT`: certificate to validate full-service.
- `MNEMONIC`: account to import for full-service. Not Secure.
- `SIGNAL`: which signal client to use. can be 'signal-cli' or 'auxin-cli'. Defaults to auxin.
- `ROOT_DIR`: specify the directory where the data file is stored, as well as where the signal-cli executable is. Defaults to `/tmp/local-signal` if DOWNLOAD, `/app` if running on fly, and `.` otherwise
- `SIGNAL_PATH`: specify where the signal client executable is if it is not in ROOT_DIR.
- `LOGLEVEL`: what log level to use for console logs (DEBUG, INFO, WARNING, ERROR). Defaults to DEBUG
- `TYPO_THRESHOLD`: maximum normalized Levenshtein edit distance for typo correction. 0 is only exact matches, 1 is any match. Default: 0.3

- `SIGNAL_CLI_PATH`: path to executable to use. useful for running signal-cli with graalvm tracing agent
- `GOOGLE_MAPS_API`: google maps api key
- `PAUTH`: used for PersistDict; see </pdictng_docs/README.md>
- `SALT`: used for PersistDict
- `METRICS_SALT`: used for logging when users were first and last seen. Must be set to log

Binary flags

- `DOWNLOAD`: download/upload datastore from the database instead of using what's in the current working directory.
- `UPLOAD`: can be used to upload as a backup without downloading
- `AUTOSAVE`: start MEMFS, making a fake filesystem in `./data` and used to upload the signal-cli datastore to the database whenever it is changed. If `DOWNLOAD`, also create an equivalent tmpdir at /tmp/local-signal, chdir to it, and symlink signal-cli process and avatar.
- `MONITOR_WALLET`: monitor transactions from full-service. Relevant only if you're giving users a payment address to send mobilecoin to instead of using signal pay. Experimental, do not use.
- `LOGFILES`: create a debug.log.
- `ADMIN_METRICS`: send python and roundtrip timedeltas for each command to ADMIN.
- `ENABLE_MAGIC`: use string distance and expansions

Contributing

We accept Issues and Pull Requests. These are our style guides:

Code style: Ensure that `mypy *py` and `pylint *py` do not return errors before you push.

Use black to format your python code. Prefer verbose, easier to read names over more concise ones.

```
pip install black pylint mypy types-protobuf types-termcolor
```

Install black pre-commit hook with

```
ln -s (readlink -f .githooks/pre-commit) .git/hooks/pre-commit
```

on fish, or

```
ln -s $(readlink -f .githooks/pre-commit) .git/hooks/pre-commit
```

on bash. Requires black to be installed.

Code Snippets

Test_questions_bot.py

See on github:
https://github.com/mobilecoinofficial/forest/blob/main/tests/test_questions_bot.py

```
import asyncio
import os
import pytest
import pytest_asyncio


# Prevent Utils from importing dev_secrets by default
os.environ["ENV"] = "test"


from forest.core import Message, run_bot, Response
from forest import core
from tests.mockbot import MockBot, Tree, QuestionBot

# Sample bot number alice
BOT_NUMBER = "+11111111111"
USER_NUMBER = "+22222222222"


# class TestBot(QuestionBot):
class TestBot(MockBot):
    """Bot that has tests for every type of question"""

    # async def do_test_ask_multiple(self, message:Message) -> None:

    async def do_test_ask_yesno_question(self, message: Message) ->
Response:
        """Asks a sample Yes or No question"""

        if await self.ask_yesno_question(
```

```
            (message.uuid, message.group), "Do you like faeries?"
        ):
            return "That's cool, me too!"
        return "Aww :c"

    async def do_test_multiple_choice_list_no_confirm(
        self, message: Message
    ) -> Response:
        """Asks a Sample Multiple Choice question with list and no
confirmation"""

        question_text = "What is your favourite forest creature?"
        options = ["Deer", "Foxes", "Faeries", "Crows"]

        choice = await self.ask_multiple_choice_question(
            (message.uuid, message.group),
            question_text,
            options,
            require_confirmation=False,
        )
        if choice and choice == "Faeries":
            return "Faeries are my favourite too c:"

        if choice:
            return f"I think {choice} are super neat too!"

        return "oops, sorry"

    async def do_test_multiple_choice_list_with_confirm(
        self, message: Message
    ) -> Response:
        """Asks a Sample Multiple Choice question with list and
confirmation"""

        question_text = "What is your favourite forest creature?"
        options = ["Deer", "Foxes", "Faeries", "Crows"]

        choice = await self.ask_multiple_choice_question(
            (message.uuid, message.group), question_text, options
        )
        if choice and choice == "Faeries":
            return "Faeries are my favourite too c:"
```

```
        if choice:
            return f"I think {choice} are super neat too!"

        return "oops, sorry"

    async def do_test_multiple_choice_dict_with_confirm(
        self, message: Message
    ) -> Response:
        """Asks a Sample Multiple Choice question with dict and
confirmation"""

        question_text = "What is your favourite forest creature?"
        options = {"A": "Deer", "B": "Foxes", "⛧": "Faeries", "D": "Crows"}

        choice = await self.ask_multiple_choice_question(
            (message.uuid, message.group),
            question_text,
            options,
            require_confirmation=True,
        )
        if choice and choice == "Faeries":
            return "Faeries are my favourite too c:"

        if choice:
            return f"I think {choice} are super neat too!"

        return "oops, sorry"

    async def do_test_multiple_choice_dict_no_confirm(
        self, message: Message
    ) -> Response:
        """Asks a Sample Multiple Choice question with a dict and no
confirmation"""

        question_text = "What is your favourite forest creature?"
        options = {"A": "Deer", "B": "Foxes", "⛧": "Faeries", "D": "Crows"}

        choice = await self.ask_multiple_choice_question(
            (message.uuid, message.group),
            question_text,
            options,
            require_confirmation=False,
        )
```

```
        if choice and choice == "Faeries":
            return "Faeries are my favourite too c:"

        if choice:
            return f"I think {choice} are super neat too!"

        return "oops, sorry"

    async def do_test_multiple_choice_dict_emptyval(self, message: Message)
-> Response:
        """Asks a Sample Multiple Choice question with a dict with empty
values"""

        question_text = "What is your tshirt size?"
        options = {"S": "", "M": "", "L": "", "XL": "", "XXL": ""}

        choice = await self.ask_multiple_choice_question(
            (message.uuid, message.group),
            question_text,
            options,
            require_confirmation=True,
        )
        if choice:
            return choice
        return "oops, sorry"

    async def do_test_multiple_choice_dict_mostly_emptyval(
        self, message: Message
    ) -> Response:
        """Asks a Sample Multiple Choice question with a dict with mostly
empty values"""

        question_text = "What is your tshirt size?"
        options = {"S": "", "M": "M", "L": "", "XL": "", "XXL": ""}

        choice = await self.ask_multiple_choice_question(
            (message.uuid, message.group),
            question_text,
            options,
            require_confirmation=True,
        )
        if choice:
            return choice
```

```
        return "oops, sorry"

    async def do_test_address_question_no_confirmation(
        self, message: Message
    ) -> Response:
        """Asks a sample address question"""

        address = await self.ask_address_question((message.uuid,
message.group))

        if address:
            return address
        return "oops, sorry"

    async def do_test_address_question_with_confirmation(
        self, message: Message
    ) -> Response:
        """Asks a sample address question"""

        address = await self.ask_address_question(
            (message.uuid, message.group), require_confirmation=True
        )

        if address:
            return address
        return "oops, sorry"

    async def do_test_ask_freeform_question(self, message: Message) ->
Response:
        """Asks a sample freeform question"""

        answer = await self.ask_freeform_question(
            (message.uuid, message.group), "What's your favourite tree?"
        )

        if answer:
            return f"No way! I love {answer} too!!"
        return "oops, sorry"


@pytest_asyncio.fixture()
async def bot():
    """Bot Fixture allows for exiting gracefully"""
```

```
    bot = TestBot(BOT_NUMBER)
    yield bot
    bot.sigints += 1
    bot.exiting = True
    bot.handle_messages_task.cancel()
    await bot.client_session.close()
    await core.pghelp.pool.close()


@pytest.mark.asyncio
async def test_dialog(bot) -> None:
    """Tests the bot by running a dialogue"""
    dialogue = [
        ["test_ask_yesno_question", "Do you like faeries?"],
        ["yes", "That's cool, me too!"],
    ]

    for line in dialogue:
        assert await bot.get_cmd_output(line[0]) == line[1]


@pytest.mark.asyncio
async def test_yesno_tree(bot) -> None:
    """Tests the bot by running a tree"""
    tree = Tree(
        ["test_ask_yesno_question", "Do you like faeries?"],
        [Tree(["yes", "That's cool, me too!"]), Tree(["no", "Aww :c"])],
    )
    tests = tree.get_all_paths()

    for test in tests:
        for subtest in test:
            assert await bot.get_cmd_output(subtest[0]) == subtest[1]


if __name__ == "__main__":
    run_bot(TestBot)
```

Podcast Appearances

Developers Eating The World – Aisling Fae Cloud Engineer (youtube)