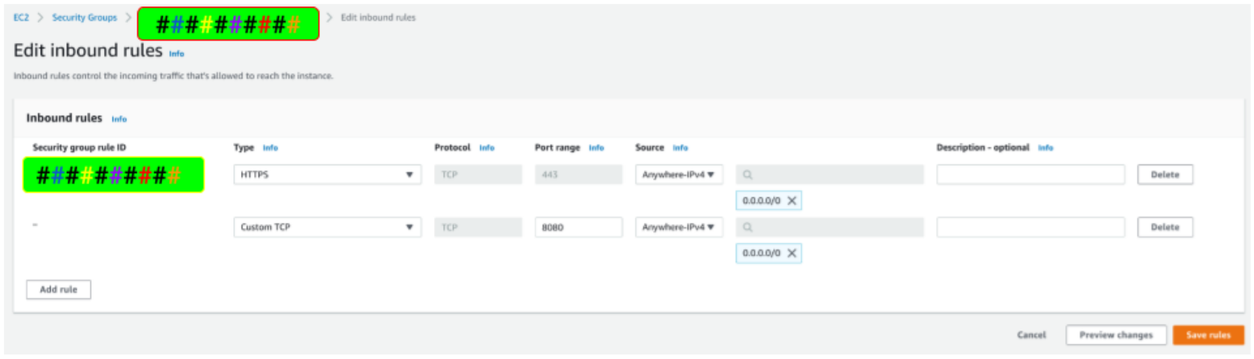
#### public_aws_ec2_setup
Production Server AWS EC2 Public Access Network Setup Community Plotly Dash

#### With the repos:
- Minimal https://github.com/lineality/mvp_production_wsgi_plotly_dash_ec2
- Geomap https://github.com/lineality/geomap_production_wsgi_plotly_dash_ec2/

# Publicly Accessible AWS EC2

The main goal of this article is creating and configuring an AWS EC2 instance (basically a cloud-computer, a web-server, that potentially is open to the public) (and configuring that) to be indeed open to the public AND using a production server (not a testing-only server). So the main goal here are the first steps of a public production server in EC2, but with an eye on the next step of what you are actually going to do with it. However, there is so much possible variation (depending on context) for that second step that only limited help for that will be possible here. The ec2 network configuration guide here will probably apply most places, but there will be only one example of a gunicorn wsgi production server for plotly dash. This is particularly hard to find instructions for online, but this guide will step you through. Other flask, fast-api projects have many online guides and better documentation.

#### The first milestone to focus on is getting to the 'inbound rules' for your ec2.
The "Edit inbound rules" screen in AWS is tragically buried and not easy to find even though it should be the obvious front-and-center part of EC2 configuration. But you can do it!


EC2 > Security Groups > ######### > Edit inbound rules

Edit inbound rules Info

Inbound rules control the incoming traffic that's allowed to reach the instance.

Inbound rules Info

Security group rule ID	Type	Protocol	Port range	Source		Description - optional
#########	HTTPS	TCP	443	Anywhere-IPv4	0.0.0.0/0	
-	Custom TCP	TCP	8080	Anywhere-IPv4	0.0.0.0/0	

But once you get to this holey-grail setup screen...what exactly should this be set to? You may need to try a few things and test what works. And please do test and experiment, do not just assume whomever will be able to connect or just rely on whatever documentation you might be trying to follow.

# Production Server:
There will be two production deploy examples here, a super-mvp production server for easy testing and extrapolation, and a more visually functional geo-map using open-street view.

For some reason the exact details are extremely important and there are many details that can go wrong. If what you are trying is not working, I recommend starting again completely from scratch. New instance, new settings, new everything.

It may seem non sequitur but I want to show you the code that works first. It is unclear to me why there is (as of 2022) no single place on the internet that shows a single working example or even a process or instructions for hosting plotly dash (probably the most universal dashboarding software) on EC2 (definitely the most common web services tool period).

The code/process is actually much simpler than other methods (which shall remain nameless). The source for this is technically plotly.com, but it is 10% from a larger heroku example they give (the ONLY example they give...why?) and then one critical last line buried in a help page (which also had to be modified).

The file structure can be 'flat' with just two files (app.py and requirements.txt (which is optional)) and the venv env directory (which is also optional).

This method probably with works with "app-factory" proliferation of folders and files, but it does not require that.

## Minimal MVP app.py code (works)

```
Less minimal app.py code here:
https://dash.plotly.com/deployment
+
correct invocation line here:
https://community.plotly.com/t/error-with-gunicorn-application-object-must-be-callable/31397

test invoke with: (but will end when your ec2 terminal session ends, resets, etc.)
   $ gunicorn app:server --bind=0.0.0.0:8050

For persistent production you will need: (end with kill process number)
   (ENV)$ nohup gunicorn app:server --bind=0.0.0.0:8050 &
   or
   (ENV)$ screen gunicorn app:server --bind=0.0.0.0:8050 &
```

```
# Import your Python Libraries (you may need boto3 for AWS)
from dash import Dash, dcc, html, Input, Output
import os

# Connects Dash -> Flask (for wsgi/gunicorn production server)
app = Dash(__name__)

# needed for gunicorn/wsgi to connect
server = app.server

# Your visualization code goes here etc.
app.layout = html.Div([
    html.H2('Hello World')
])

if __name__ == '__main__':
    app.run_server(host= '0.0.0.0', port=8050)

```

#### requirement.txt
```
dash
plotly
gunicorn
```
Note: here is an example of where the final list of packages will be longer when it includes all the dependencies, but all you need to install are these three.

## Test Test Test

As you modify the working code gradually to add in your own content, make sure to preserve the key parts of the code. Test as you go to make sure everything still works. These may include the following:

#### needed for gunicorn/wsgi to connect
```
server = app.server
```

#### Closing Line
```
if __name__ == '__main__':
    app.run_server(host= '0.0.0.0', port=8050)
```

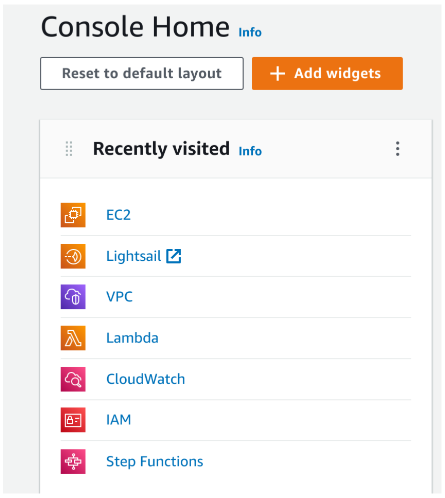
#### Invocation from ec2 terminal
```
$ screen gunicorn app:server --bind=0.0.0.0:8050 &
```

See: https://github.com/lineality/aws_rules_of_thumb_and_warnings

# Instructions to set up public AWS EC2
e.g. to host a flask server, dashboard, REST api endpoint, etc.
(See picture version as pdf in repo, pictures may help!)

### Getting Started:
- You will need a computer and an internet connection.
- Depending on the project, you may be able to use a table, or phone, but for some projects you will need a laptop or desktop with more complete resources. MacOS(bsd) or linux are recommended.
- You will need an AWS account (yours, employers, schools, etc.).

### Go to: AWS
Log in and go to the console.
https://us-east-1.console.aws.amazon.com

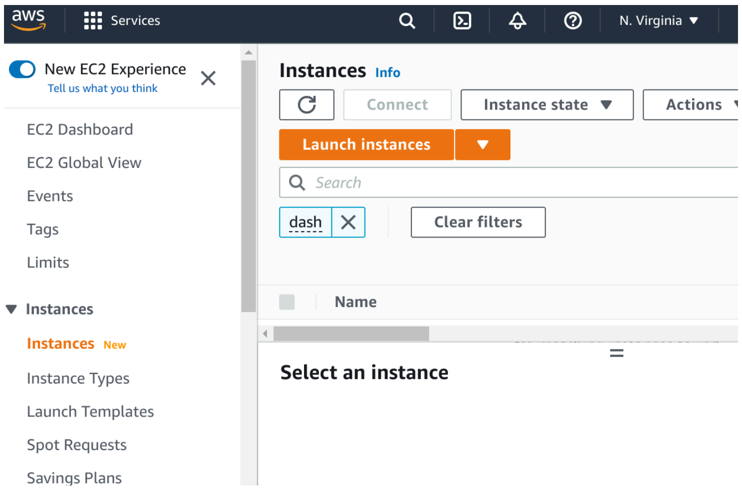


### Go to: EC2

https://us-east-1.console.aws.amazon.com/ec2/

### Go to: instances

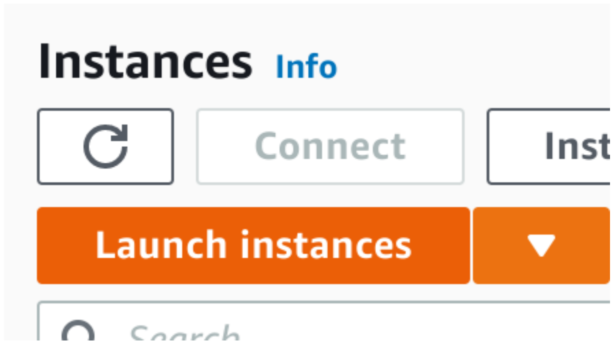
the instances tab



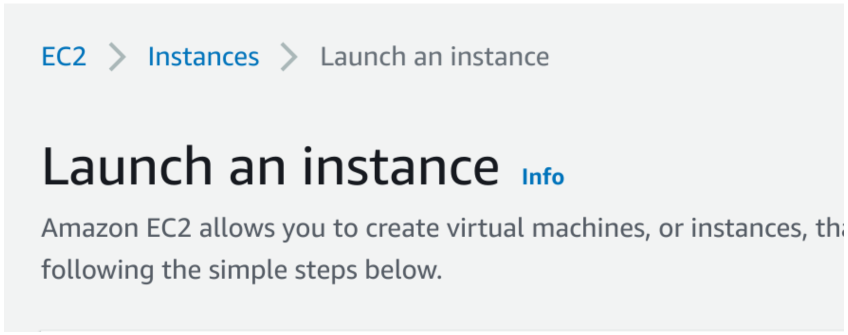
## Launch Instance:

Hit the big orange button that says "Launch instances"

(plural...for some reason...which of course takes to you "launch an instance" singular)



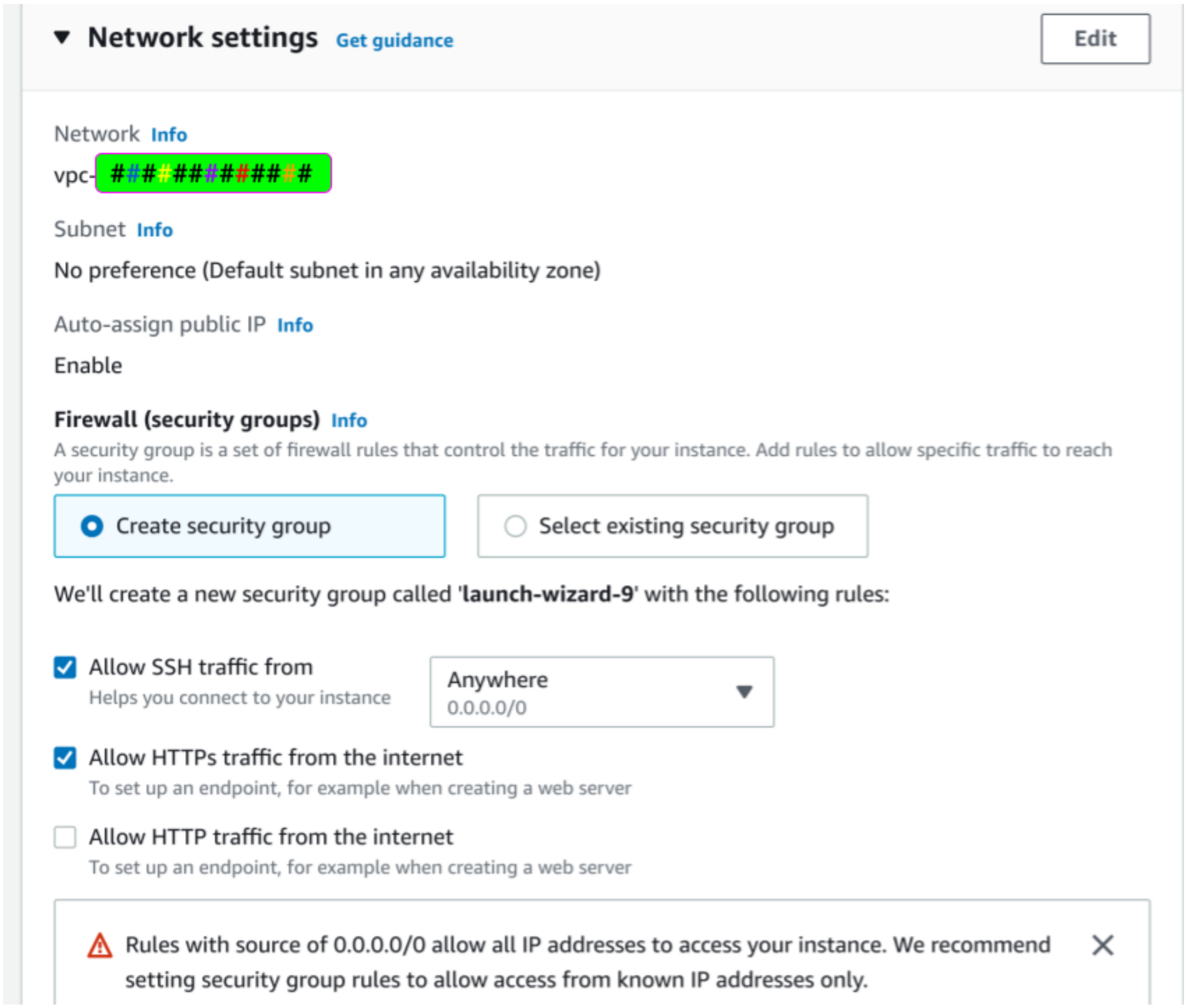
# *"launch an instance" singular*



Amazon EC2 allows you to create virtual machines, or instances, th
following the simple steps below.

## Configure:
1. Name and tags -> clear meaningful name, nothing is too obvious. recommended format: "ec2_purpose_yourname_datetime"
2. Application and OS Images (Amazon Machine Image) -> default amazon linux
3. Instance type -> nano (scroll down)
4. Key pair (login) -> select or make new pair
5. Network settings...(see below)

# Network settings:
1. firewall security group: create or select
2. "Allow SSH traffic from": must be on to use EC2 connect later (or SSH in yourself)
3. "Allow HTTPs traffic from the internet": If you want this to be public, allow.
4. http may be needed in the mess of aws connection issues, leave it on for now



Network Info
vpc-[redacted]

Subnet Info
No preference (Default subnet in any availability zone)

Auto-assign public IP Info
Enable

Firewall (security groups) Info
A security group is a set of firewall rules that control the traffic for your instance. Add rules to allow specific traffic to reach your instance.

● Create security group ○ Select existing security group

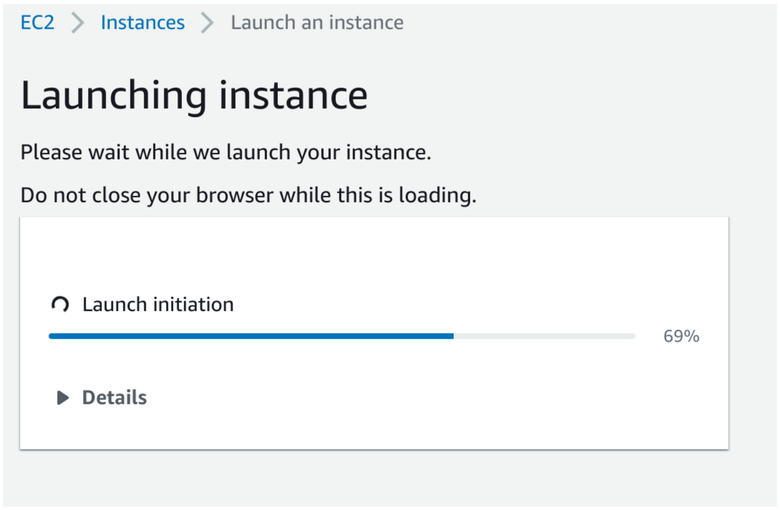
We'll create a new security group called '**launch-wizard-9**' with the following rules:

☑ Allow SSH traffic from — Anywhere 0.0.0.0/0
Helps you connect to your instance

☑ Allow HTTPs traffic from the internet
To set up an endpoint, for example when creating a web server

☐ Allow HTTP traffic from the internet
To set up an endpoint, for example when creating a web server

⚠ Rules with source of 0.0.0.0/0 allow all IP addresses to access your instance. We recommend setting security group rules to allow access from known IP addresses only.

### Network settings (continued...)
6. Storage (volumes -> use default
7. Advanced details Info -> ignore
8. Summary -> nothing to do or change here, examine if you want.
## Select: Launch Instance

EC2 > Instances > Launch an instance

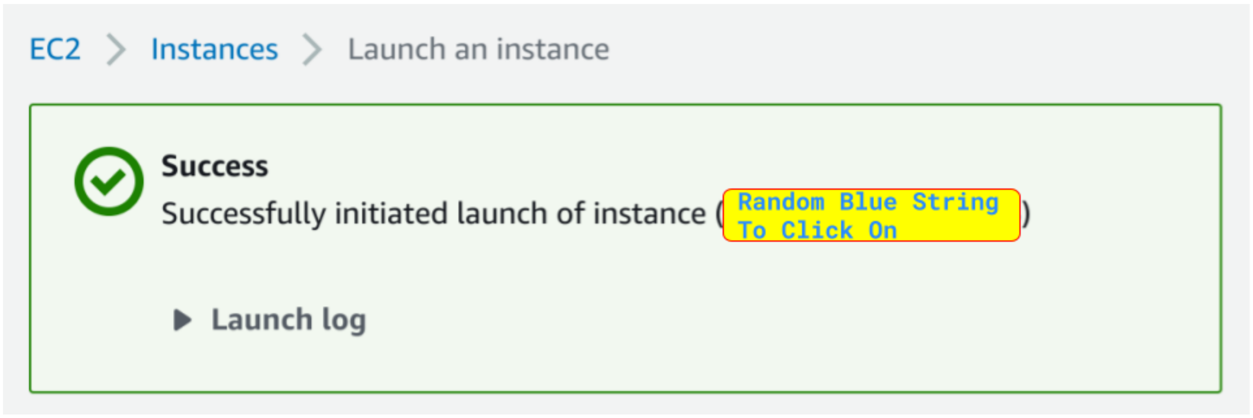
Launching instance

Please wait while we launch your instance.

Do not close your browser while this is loading.

Launch initiation 69%

Details

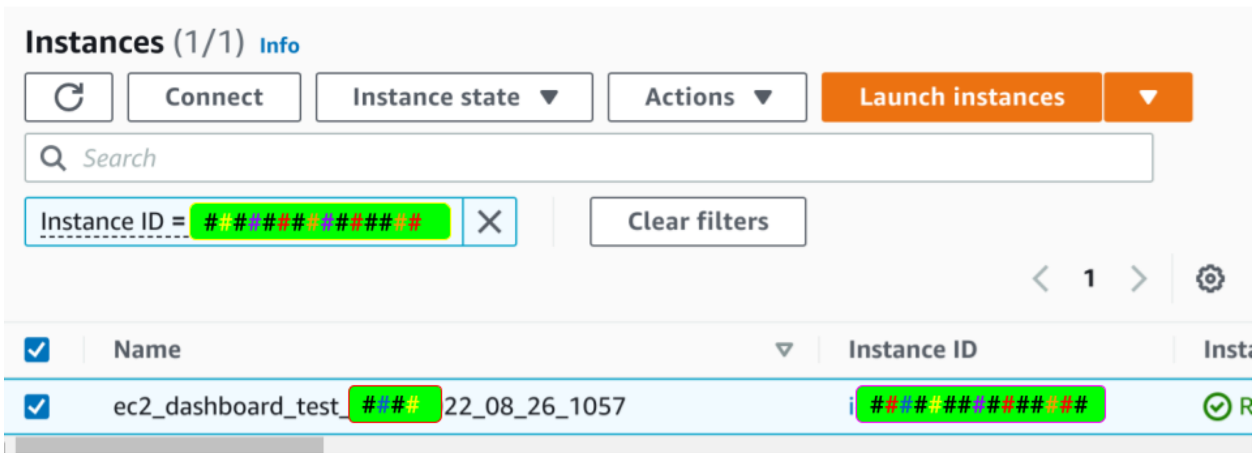
## Click on the blue random string .... (obviously...?)

EC2 > Instances > Launch an instance

Success

Successfully initiated launch of instance (Random Blue String To Click On)

Launch log

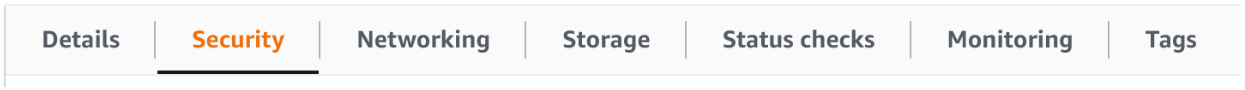
## Back at instances window:
your instance should now be highlighted: click on "connect" to connect via web
This is much easier that local-cli ssh (web connect is one of the few actually useful working advances AWS has made.



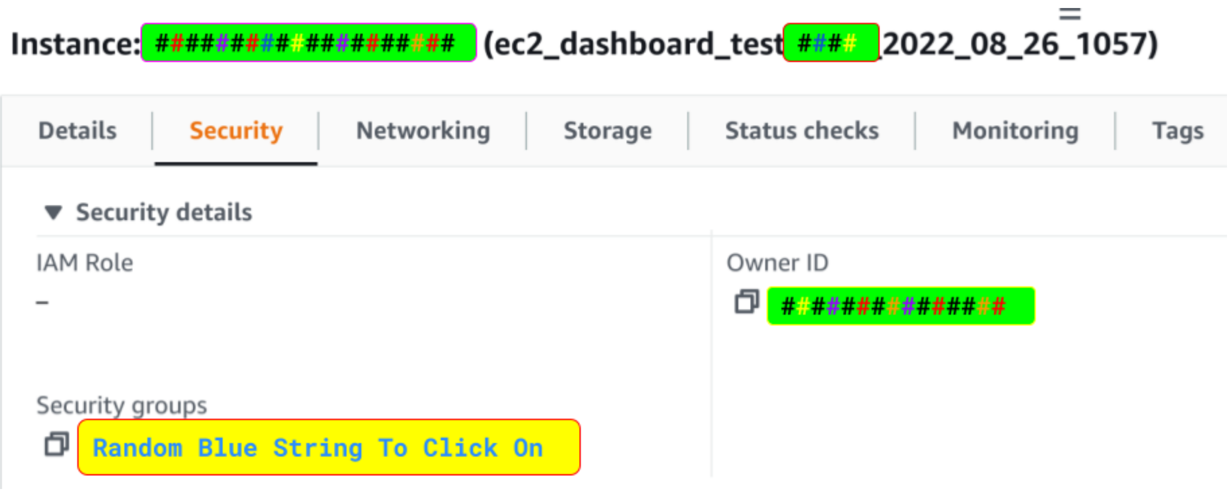
Instance ID = #############

Name	Instance ID
ec2_dashboard_test_####22_08_26_1057	i#############

# configure in "security"
(Obviously, since you want to do network configuration, and you have the choice of 'networking' you instead need to go to "security." So user friendly.)



## Another random blue-string-click
In the "Security" tab, under "security groups" (plural?) you see a random blue-string-link. click on that (to configure networking...obviously...)



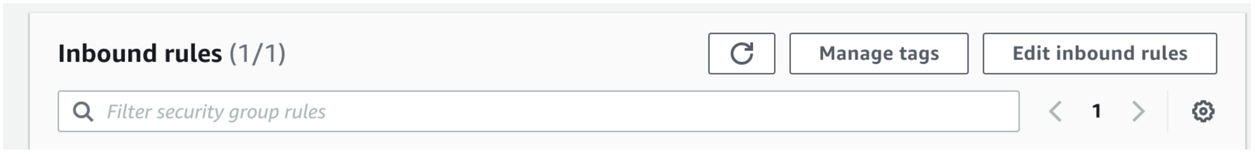
Instance:################## (ec2_dashboard_test####2022_08_26_1057)

IAM Role
–

Owner ID
############

Security groups
Random Blue String To Click On

## "Inbound rules" You are here!
Finally: This is the basic, rudimentary, necessary, "start here" configuration menu that all this has been leading up to (and should have started with), yet for some obscene reason AWS makes it impossible to even find.

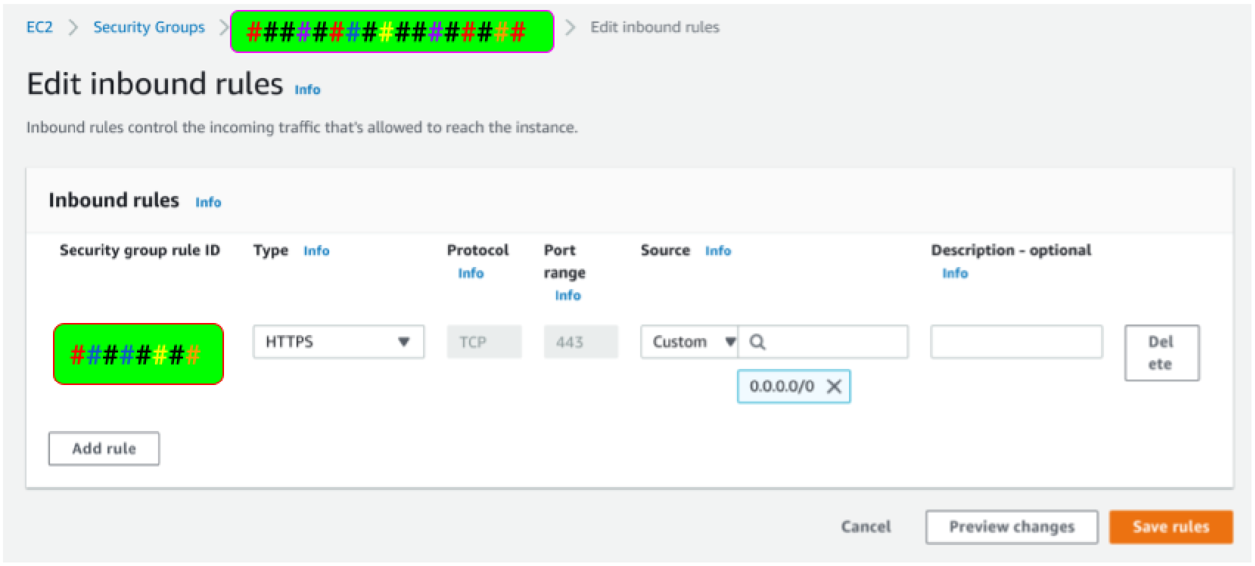


Inbound rules (1/1)

Filter security group rules

### Click "edit inbound rules"

### Make and save new rules.
Using the following tool (which you should see now),



EC2 > Security Groups > ################# > Edit inbound rules

Edit inbound rules Info

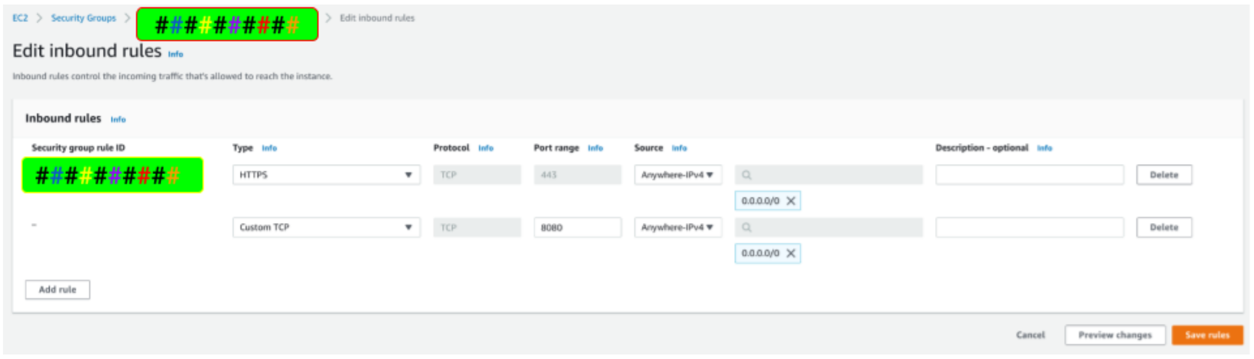
Inbound rules control the incoming traffic that's allowed to reach the instance.

Inbound rules Info

Security group rule ID	Type Info	Protocol Info	Port range Info	Source Info	Description - optional Info	
########	HTTPS	TCP	443	Custom 0.0.0.0/0		Delete

Create and save (using the big orange "Save rules" button) the rules in this table.
Existing rules may need to be modified or replaced (e.g. HTTPS may be set to custom, set it to Anywhere IPV4)

```
      Type         (Protocol)  Port Range  Source          (to)
1.    HTTPS TCP    TCP         80          Anywhere IPV4   0.0.0.0/0
2.    HTTPS TCP    TCP         443         Anywhere IPV6   ::/0
3.    Custom TCP   TCP         8050        Anywhere IPV4   0.0.0.0/0
4.    SSH          TCP         22          Custom          0.0.0.0/0
5.    HTTP  TCP    TCP         80          Anywhere IPV4   0.0.0.0/0
6.    HTTP  TCP    TCP         443         Anywhere IPV6   ::/0
```


EC2 > Security Groups > [redacted] > Edit inbound rules

Edit inbound rules Info

Inbound rules control the incoming traffic that's allowed to reach the instance.

Inbound rules Info

Security group rule ID	Type	Protocol	Port range	Source		Description - optional	
[redacted]	HTTPS	TCP	443	Anywhere-IPv4	0.0.0.0/0		Delete
–	Custom TCP	TCP	8080	Anywhere-IPv4	0.0.0.0/0		Delete

Another example rule set:

Set **Type** *HTTP*, **Protocol** *TCP*, **Port range** *80*, and **Source** to "*0.0.0.0/0*".

Set **Type** *HTTP*, **Protocol** *TCP*, **Port range** *80*, and **Source** to "*::/0*".

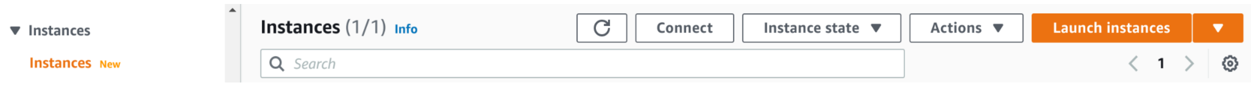
Set **Type** *Custom TCP*, **Protocol** *TCP*, **Port range** *8080*, and **Source** to "*0.0.0.0/0*".

Set **Type** *SSH*, **Protocol** *TCP*, **Port range** *22*, and **Source** to "*0.0.0.0/0*".

Set **Type** *HTTPS*, **Protocol** *TCP*, **Port range** *443*, and **Source** to "*0.0.0.0/0*".

Done.

### Go back to the instances tab



# Note!

The exact ports you need to select (e.g. 8080 vs. 8050) etc, may depend on what you are doing, and on how your project is configured (flask, dash, fast-api, etc.)

For plotly dash you may need to use 8050 and use this line in your app.run command:

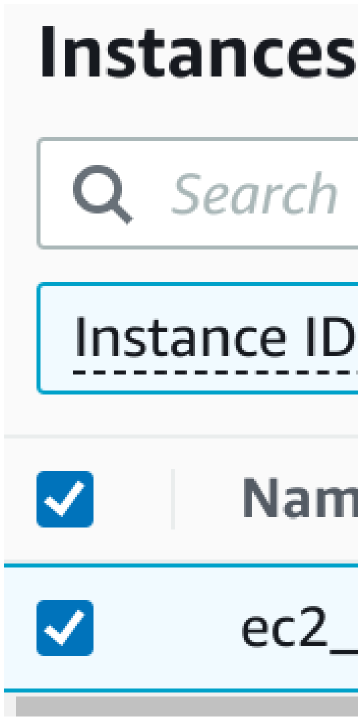
```
if __name__ == '__main__':
    app.run_server(host= '0.0.0.0',port=8050)
```

And you may need to add a port suffix after the ipv4URL you get from AWS.
#### In these working examples, plotly-dash's port 8050 was added to the end of the original url.
```
http://3.94.153.137:8050/
or
http://ec2-3-94-153-137.compute-1.amazonaws.com:8050/
```

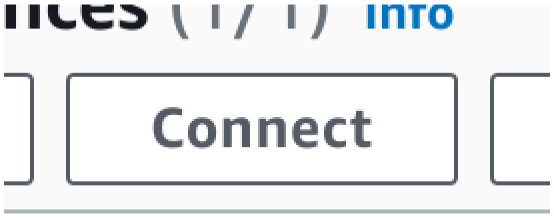
# Web Connect

Note: In order for the grey 'connect' button to be active (not ghost-grey), you will need to make sure you check the boxes and see a blue checkmark next to the EC2 you wish to connect to.

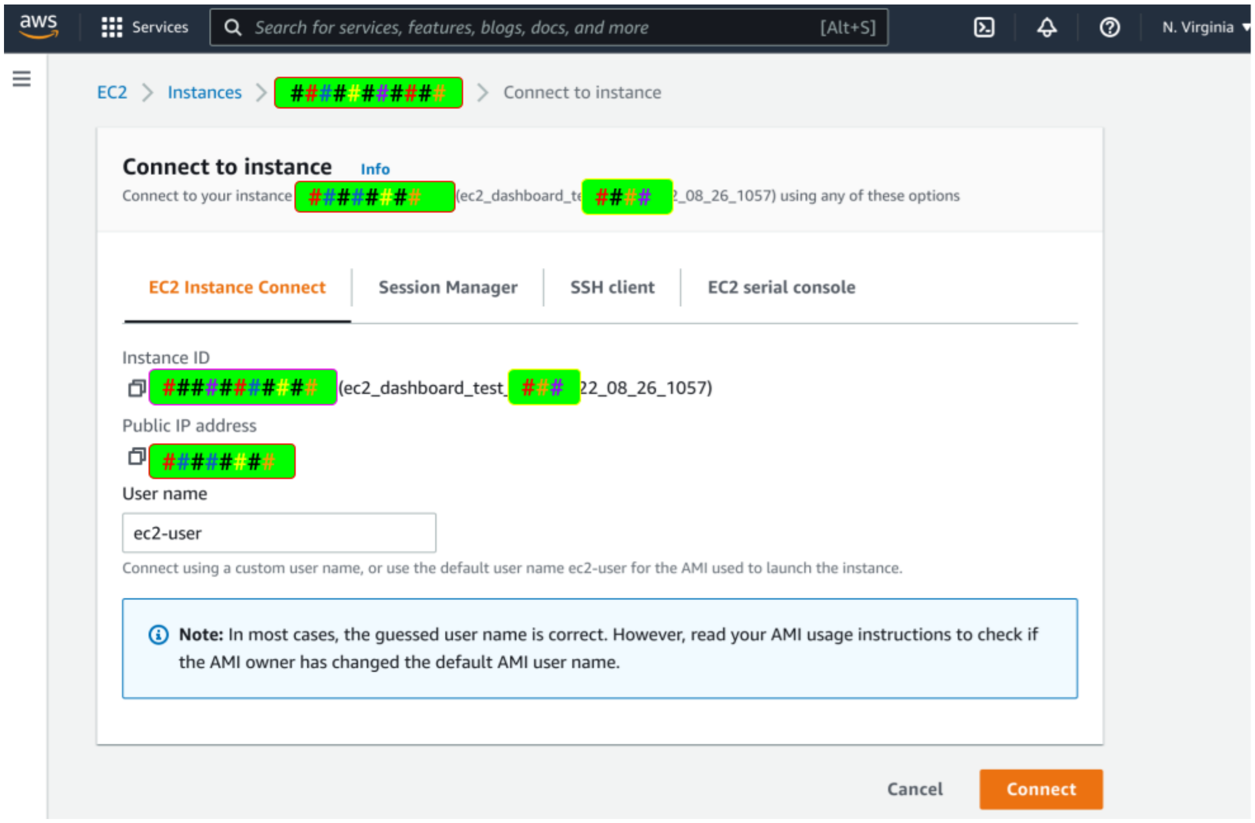
(blue check box pic)



## click "connect"
Click on the grey "Connect" button.



### In the 'connect to instance' window
in the "EC2 Instance Connect" tab...


aws
Services
Search for services, features, blogs, docs, and more [Alt+S]
N. Virginia

EC2 > Instances > [redacted] > Connect to instance

Connect to instance Info

Connect to your instance [redacted] (ec2_dashboard_te [redacted] 2_08_26_1057) using any of these options

EC2 Instance Connect | Session Manager | SSH client | EC2 serial console

Instance ID

[redacted] (ec2_dashboard_test_ [redacted] 22_08_26_1057)

Public IP address

[redacted]

User name

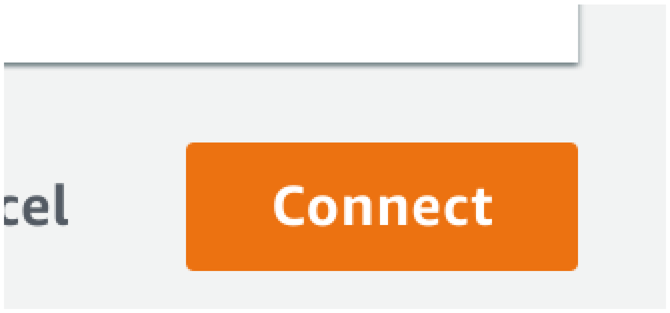
ec2-user

Connect using a custom user name, or use the default user name ec2-user for the AMI used to launch the instance.

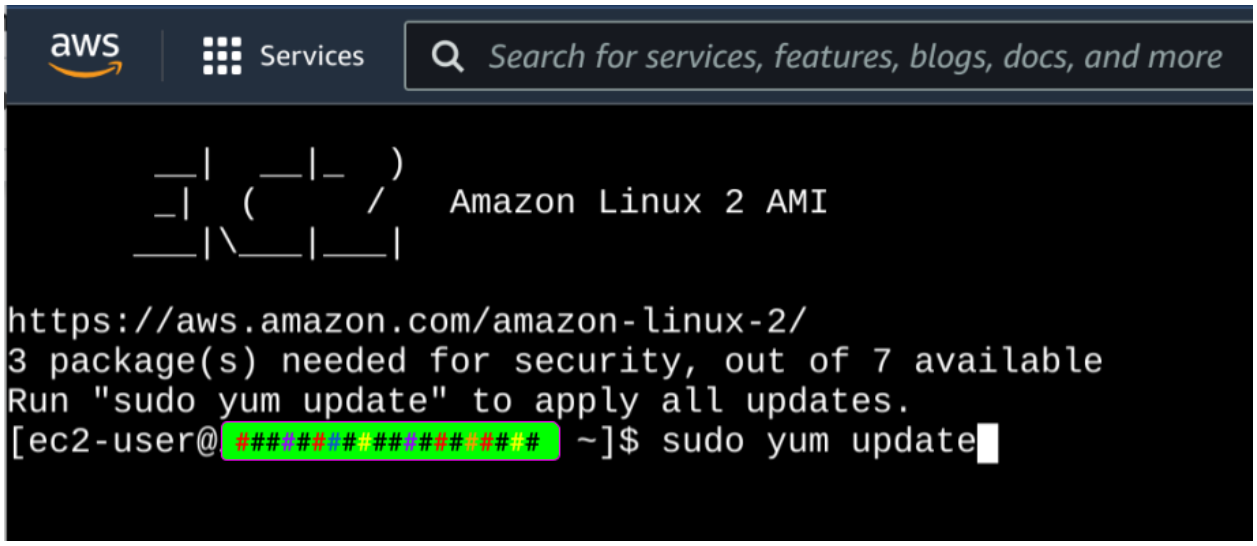
Note: In most cases, the guessed user name is correct. However, read your AMI usage instructions to check if the AMI owner has changed the default AMI user name.

Cancel | Connect

### Click on "Connect" (the big orange button)...(dejavu?)



### Like SSH but with no convoluted local aws-cli setup nightmare. (This is a good thing.)



```
       __|  __|_  )
       _|  (     /   Amazon Linux 2 AMI
      ___|\___|___|

https://aws.amazon.com/amazon-linux-2/
3 package(s) needed for security, out of 7 available
Run "sudo yum update" to apply all updates.
[ec2-user@[redacted] ~]$ sudo yum update
```

### Optional steps: for using github
If you are going to get files from github
```
$ sudo yum update -y
$ sudo yum install git -y
```

# Run server in EC2
The code you run in your EC2 to start the server will likely look something like this:
- update
- install git
- make your project folder
- project files
- project python env
- run !!

#### Steps:
```
$ sudo yum update -y

$ sudo yum install git -y

$ mkdir viz; cd viz
```

#### Add Project Code
Whether from github or just typed in on nano (fast enough), put your code in:
```

$ git clone https://github.com/lineality/mvp_production_wsgi_plotly_dash_ec2.git

$ cd mvp_production_wsgi_plotly_dash_ec2

or

$ git clone https://github.com/lineality/geomap_production_wsgi_plotly_dash_ec2.git

$ cd geomap_production_wsgi_plotly_dash_ec2

or

$ nano app.py

$ nano requirements.txt
```

### Once you have your files uploaded or written:
Using a venv env is recommended (if not technically required), e.g. linux base systems often have their own mix of python libraries pre-installed, and it's best to have a clean python system for your app where you know exactly what is installed.

```
$ python3 -m venv env; source env/bin/activate

(ENV)$ python3 -m pip install --upgrade pip

(ENV)$ pip install -r requirements.txt
```

## Why 'in background'? Persistence!

#### Test invoke with:
(but will end when your ec2 terminal session ends, resets, etc.)
```
    $ gunicorn app:server --bind=0.0.0.0:8050
```

BUT this will stop when you end your terminal session. As long as your terminal is open you are fine (e.g. for you testing at that moment), but if you want anyone online to be able to access that server any time, then the server must be running 'in the background' (or whatever equivalent) so that the server does not shut down for everyone else as soon as you close your terminal.

# Production

Test and run-with-persistance using these. (I found 'screen' worked better but do whatever works for you.)

#### For persistent production you will need: (end with kill process number)
```
   (ENV)$ nohup gunicorn app:server --bind=0.0.0.0:8050 &
   or
   (ENV)$ screen gunicorn app:server --bind=0.0.0.0:8050 &
```

The output may look like this (not the normal output saying what IP etc.)
```
(env) [ec2-user@XXX]$ nohup: ignoring input and appending output to ‘nohup.out’
```
You may want to wait until you have tested etc. before you run this.

Screen will look different. You can refresh the whole connection (and test to make sure the dashboard server is still going in the 'background').

## Check Connection:

#### Reminder:
You will probably need to add a port suffix after the ipv4URL you get from AWS.

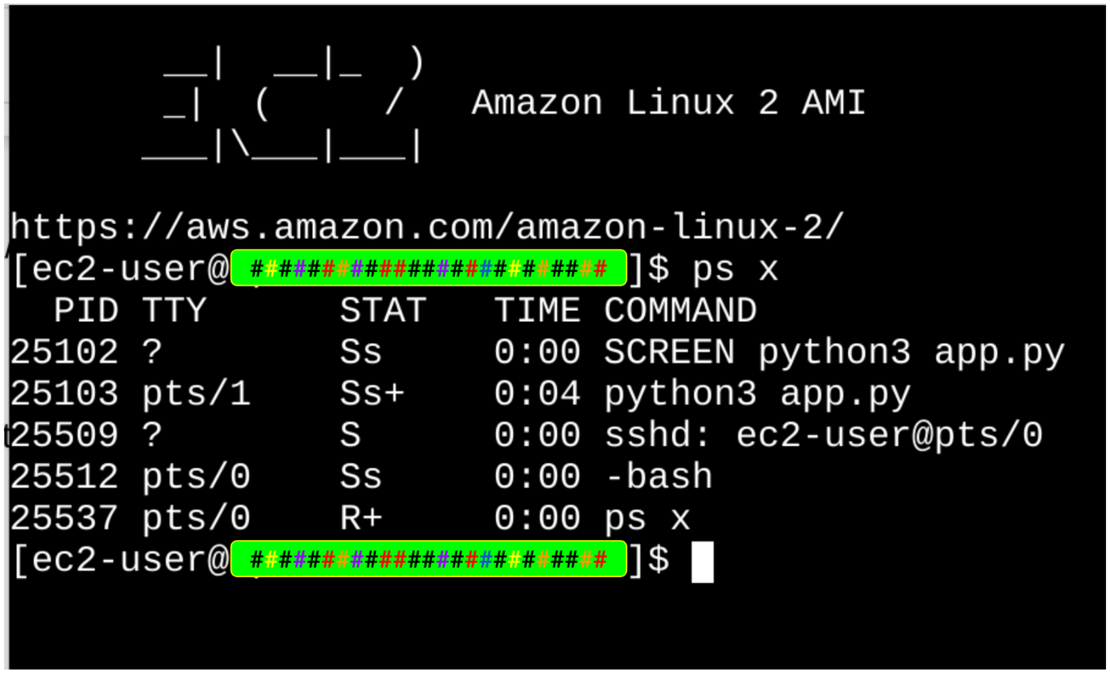
#### In these working examples, plotly-dash's port 8050 was added to the end of the original url.
```
(These are just examples that will not refer to your server; note the added port number at the end. "8050" This number will depend on what you set up and on what your software requires.)
http://3.94.153.137:8050/
or
http://ec2-3-94-153-137.compute-1.amazonaws.com:8050/
```

## See what processes are running:

```
$ ps -x

or

$ ps x
```
more suggestions here:
https://www.freecodecamp.org/news/linux-list-processes-how-to-check-running-processes/

Terms:
```
        PID = process ID (use this to 'kill' the process)
        TTY = name of controlling terminal for process

        STAT = process state code, such as Z (zombie), S (sleeping), and R (running),
                 note: 'sleeping' can me waiting for user-input,
```

Note: Your running-in-background process may appear as multiple processes, so do not kill one by mistake thinking it is a leftover from past testing or some other error.

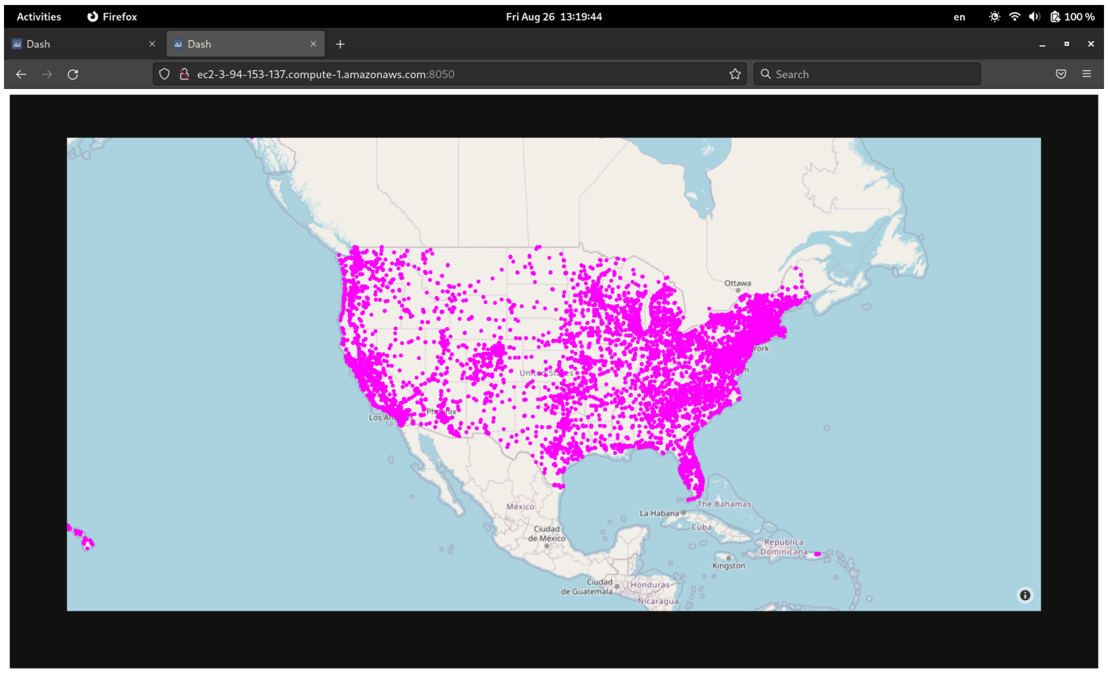


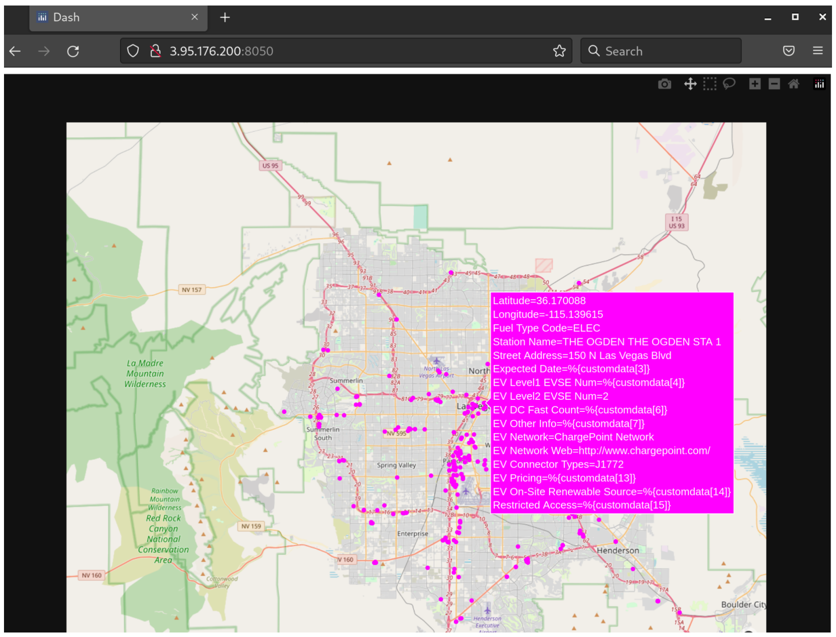
```
       __|  __|_  )
       _|  (     /   Amazon Linux 2 AMI
      ___|\___|___|

https://aws.amazon.com/amazon-linux-2/
[ec2-user@[redacted]]$ ps x
  PID TTY      STAT   TIME COMMAND
25102 ?        Ss     0:00 SCREEN python3 app.py
25103 pts/1    Ss+    0:04 python3 app.py
25509 ?        S      0:00 sshd: ec2-user@pts/0
25512 pts/0    Ss     0:00 -bash
25537 pts/0    R+     0:00 ps x
[ec2-user@[redacted]]$ 
```

## How to End a Process
Get the process id (PID) number by running ps -x (or something similar)
```
$ kill PID_NUMBER_HERE
```
More here https://linuxize.com/post/how-to-kill-a-process-in-linux/

## **Test Only Example (not Production)**
I recommend simply using a wsgi/gunicorn production method, but if for some reason you want to use the not-production server (complete with warnings to not use it), then experiment with this:
```
$ git clone https://github.com/lineality/plotly_dash_geomap_points_energy1.git
$ cd plotly_dash_geomap_points_energy1

(ENV)$ nohup python3 app.py &
or
(ENV)$ screen python3 app.py &
```

## Example:
EC2 deployed plotly dash app viewed in browser via public access setup:





```
Latitude=36.170088
Longitude=-115.139615
Fuel Type Code=ELEC
Station Name=THE OGDEN THE OGDEN STA 1
Street Address=150 N Las Vegas Blvd
Expected Date=%{customdata[3]}
EV Level1 EVSE Num=%{customdata[4]}
EV Level2 EVSE Num=2
EV DC Fast Count=%{customdata[6]}
EV Other Info=%{customdata[7]}
EV Network=ChargePoint Network
EV Network Web=http://www.chargepoint.com/
EV Connector Types=J1772
EV Pricing=%{customdata[13]}
EV On-Site Renewable Source=%{customdata[14]}
Restricted Access=%{customdata[15]}
```

# Resources:
- https://stackoverflow.com/questions/67166003/dash-app-not-working-when-deployed-on-amazon-ec2-instance

#### Gunicorn WSGI server / systemd / Nginx Webserver
1. https://medium.com/techfront/step-by-step-visual-guide-on-deploying-a-flask-application-on-aws-ec2-8e3e8b82c4f7

#### Direct
2. https://www.twilio.com/blog/deploy-flask-python-app-aws
You can view your live application by appending *8080* to your public IPv4Public IP address. In the example of this article, the URL would be "http://52.15.127.3:8080/".

#### apache webserver and mod_wsgi
3. https://medium.com/innovation-incubator/deploy-a-flask-app-on-aws-ec2-d1d774c275a2
4. **https://www.datasciencebytes.com/bytes/2015/02/24/running-a-flask-app-on-aws-ec2/**

#### Docker
5. https://github.com/TPhil10/Bourbonhuntr

## Some Main Factors in Context
Aside from security, one of the main factors determining how things can and should be done is what project this is for and what software this EC2 cloud-computer/server needs to run. Another

factor is whether that software is ~free to use, or whether you will be using a proprietary service. Deploying a rest-api-endpoint is different from deploying a graphical interactive dashboard; for me this project is focused on the graphical interactive dashboard deployment.

The set of choices and options become tangled rather quickly.

django
flask
fast-api

When you run flask or dash you get a warning saying that you should be using a production server, in particular a wsgi production server.

#### Choice: flask/dash vanilla vs. "production fancy" flask/dash

#### Production Choices
- flask -> apache2 web server (used to run most of internet)
- flask -> ngnix web server (increasingly runs most of internet)
- flask -> wsgi web server (obscure)
- flask -> gunicorn (obscure)

For example: There are many articles online which describe how to host a flask web server on EC2. Yet while flask is very easy to run locally, and while plotly dash uses flask, and while flask can be set up in various was on

There may be several ways that one can create a public facing server (an endpoint, a dashboard, etc.) using EC2 (essentially a cheap mini-web-server with which you can do many things...if not easily...).

### Factors:
1. Purpose & Tools
2. Budget

Setting up an EC2 server for flask, or for a plotly dash dashboard, or for fastapi, or for django, or for a rest-api-endpoint (or some combination of those), can be different case-by-case.

### List of AWS EC2 Methods
1. Simply hosting a micro-server directly (dash, flask, fast-api, etc)
2. Intermediation "production" server:
- WSGI Server
- Apache Server
- Nginx
3. Docker + AWS
- running docker in EC2

- uploading a pre-built docker image (maybe not EC2?)
- maybe some weird hybrid mix of things?

The approach taken in this guide will be the direct-microserver-method.

### Alternative AWS Services:
- Lambda function
- lambda function + api gateway
- api + gateway
- elastic beanstalk
- lightsail
- step function + anything above

### Alternative NOT-AWS Services:
- heroku
- google cloud services
- your own hardware server

If you have lots of money and or specific needs, you may be best served by AWS's pre-built services. If you don't have lots of money, often working directly with EC2 (or lambda, etc.) is most practical.

# Readings

1. Flask: Deploy to Production
https://flask.palletsprojects.com/en/2.2.x/tutorial/deploy/

2. https://kinsta.com/blog/nginx-vs-apac he/

3. "How to Serve a Flask App with Amazon Lightsail Containers"
https://aws.amazon.com/getting-started/hands-on/serve-a-flask-app/

4. Flask EC2 Ubuntu Apache
https://medium.com/innovation-incubator/deploy-a-flask-app-on-aws-ec2-d1d774c275a2

5. Gunicorn: Standalone WSGI Containers
(old) https://flask.palletsprojects.com/en/1.1.x/deploying/wsgi-standalone/
https://flask.palletsprojects.com/en/2.2.x/deploying/
https://flask.palletsprojects.com/en/2.1.x/quickstart/

6. make server independent of ssh session and run continuously
https://stackoverflow.com/questions/23029443/run-python-flask-on-ec2-in-the-background#23030970

7. examine running processes in EC2

https://www.freecodecamp.org/news/linux-list-processes-how-to-check-running-processes/

8. End a process
https://linuxize.com/post/how-to-kill-a-process-in-linux/

9. gUnicorn
https://dash.plotly.com/deployment
https://community.plotly.com/t/error-with-gunicorn-application-object-must-be-callable/31397