**Prerequisites**

Before we diving into the tutorial, there are a couple of things that need to be done first, which I won’t cover in much details here.

**Step 1: Setting up Docker**

Setting up Docker engine on your local machine for development is important for this tutorial.

**Step 2: Building the demo Shiny app and containerised ShinyProxy Docker images**

You can skip this step if you already have a function Docker image. You should have an image called shiny-euler-app (or a name of your choice).

Now, we need to build a Docker image that contains the ShinyProxy engine. Don’t worry about the settings in application.yml, as we will override them when we docker-compose it. Also, you don’t need to set up the container-network mentioned in that tutorial either.

Now, we will push the two images to DockerHub.

docker login

Type your username and password when prompted. You will see ‘Login Succeeded’ in the console. Now, tag and push the images to your DockerHub repo:

docker tag shinyproxy-example YOUR\_USERNAME/shinyproxy-example

docker tag shiny-euler-app YOUR\_USERNAME/shiny-euler-app

docker push YOUR\_USERNAME/shinyproxy-example

docker push YOUR\_USERNAME/shiny-euler-app

Now, you should able to pull this image from anywhere using the command below. Of course, you will need to log in first if you set the repo to private.

docker pull YOUR\_USERNAME/shiny-euler-app

docker pull YOUR\_USERNAME/shinyproxy-example

**Step 3: Launching AWS EC2 instances**

Following this framework, we will need two AWS EC2 instances, one for hosting ShinyProxy and one for logging and monitoring usage statistics. Below are the minimal specs:

*ShinyProxy Machine:*

* Ubuntu 16.04 LTS (although other versions may also work)
* Open port 22 (SSH), 80 (HTTP) and 443 (HTTPS)
* Minimal 1 GiB of Memory
* Minimal 8 GB of Storage
* Install Docker Engine and Docker Compose

*Usage Monitor Machine:*

* Ubuntu 16.04 LTS (although other versions may also work)
* Open port 22 (SSH), 3000 (Grafana) and 8086 (InfluxDB)
* Minimal 0.5 GiB of Memory
* Minimal 8 GB of Storage
* Install Docker Engine

Later in this tutorial, I will refer to the two AWS instances as ShinyProxy Instance and Usage Monitor Instance. Theoretically, we can use one instance to host both of the ShinyProxy and the usage monitor tools but we won’t get much flexibility if we decide to launch another ShinyProxy in future.

The installation of Docker Engine and Docker Compose on Ubuntu won’t be covered here. You can check out the official guides:  
[Install Docker Engine on Ubuntu](https://docs.docker.com/engine/install/ubuntu/) and  
[Install Docker Compose](https://docs.docker.com/compose/install/).

According to my experience, containerised ShinyProxy tend to be memory-hungry. It uses about 300 MiB even when idle. The total memory usage depends on your Shiny app but I think we should aim to get at least 1 GiB Memory for it to run smoothly. In production, making sure you have enough memory and adding sway memory are always good ideas. Fortunately, an AWS t2.micro instance is enough for the demo, and it is on the free tier!

**Step 4: Setting up a domain name (Optional)**

If you already own a domain name (e.g. [example.com](http://example.com)), it is easy to set up a subdomain (e.g. [app.example.com](http://app.example.com)) to host your Shiny app. You will need to set up an A record with your domain registrar and point it to the IP address of the ShinyProxy Instance. Using the  
[DNS Checker](https://dnschecker.org/) to check your newly set up domain name before continue to the next steps.

**Step 5: Setting up AWS Cognito (Optional)**

If you decide to use AWS Cognito for authentication (which I highly recommend), you will need to set up a Cognito User Pool in advance. Step 1 and 2 in the guide are essential for this tutorial. Once you have set the user pool, you will need to do a couple of things:

1. Take a note of your Pool Id, which follows the format Pool {region}\_{id} (e.g. us-east-1\_XXXXXXXXX). From here, you can also get the region of your user pool, which will be useful later.
2. Create a test user account for yourself under the Users and groups tab. Create a group called admins and assign the test account to that group.
3. Create a App client called ‘shiny-simulator-test’ (or a name of your choice). Note down the App client id and App client secret.
4. Under the App integration/App client settings, check Cognito User Pool as the Enabled Identity Providers. You can also use other Identity providers such as Facebook or Google but I won’t cover them here.
5. On the same page, set the Callback URL(s) to ‘https://{your\_domain}/login/oauth2/code/shinyproxy’ and the Sign out URL(s) to ‘https://{your\_domain}’. In the tutorial, I will use ‘[app.example.com](http://app.example.com)’ as a placeholder.
6. Again, on the same page, under OAuth 2.0, check Authorization code grant, Implicit grant for Allowed OAuth Flows and then check everything except for phone for Allowed OAuth Scopes.
7. On the App integration/Domain name, set the Domain prefix of your choice and note down the whole Amazon Cognito domain. This is the domain name of the sign-in page users will be redirected to when they visit your website. You have a choice of setting up your own domain for the sign-in page.

**Setting up HTTPS (SSL / TLS)**

**Step 1: Preparing configuration files**

For the core services (e.g. ShinyProxy, Nginx and Certbot), we will use Docker Compose to set them up in one go. I have tailored it to work with ShinyProxy but feel free to check the original post for technical details, which I won’t cover here.

Below is the structure of the directory shinyproxy-docker-compose-demo:

| docker-compose.yml

| init-letsencrypt.sh

|

+---application

| application.yml

|

\---data

\---nginx

app.conf

Now, we will need to update a couple of configuration files before docker-compose it. First, let’s take a look of the application.yml

application.yml

proxy:

title: My Awesome Shiny Portal

port: 8080 # use Port 8080 for ShinyProxy

container-wait-time: 30000 # how long should we wait for the container to spin up (30s as default as this is enough for our Shiny apps)

heartbeat-rate: 10000 # the user's browser will send a heartbeat call every heartbeat-rate milliseconds (10s as default)

heartbeat-timeout: 60000 # if the server does not receive a heartbeat for heartbeat-timeout milliseconds, the relevant proxy will be released (60s as default)

usage-stats-url: http://{your\_InfluxDB\_instance\_domain}:8086/write?db=shinyproxy\_usagestats # use InfluxDB to store usage statistics; can be in a different server

# usage-stats-username: xxxx # influxdb username if needed

# usage-stats-password: xxxx # influxdb password if needed

#### OpenID Connect Authentication ####

authentication: openid # use openid auth framework

openid:

roles-claim: cognito:groups # use the groups value passed by AWS cognito to identify user groups

auth-url: # https://{cognito\_domain\_prefix}.auth.{region}.[amazoncognito.com/oauth2/authorize](http://amazoncognito.com/oauth2/authorizetoken-url" \t "_blank)

[token-url](http://amazoncognito.com/oauth2/authorizetoken-url" \t "_blank): # https://{cognito\_domain\_prefix}.auth.{region}.[amazoncognito.com/oauth2/token](http://amazoncognito.com/oauth2/tokenjwks-url" \t "_blank)

[jwks-url](http://amazoncognito.com/oauth2/tokenjwks-url" \t "_blank): # <https://cognito-idp.>{region}.[amazonaws.com/{userPoolId}/.well-known/jwks.json](http://amazonaws.com/%7BuserPoolId%7D/.well-known/jwks.jsonlogout-url" \t "_blank)

[logout-url](http://amazonaws.com/%7BuserPoolId%7D/.well-known/jwks.jsonlogout-url" \t "_blank): # https://{cognito\_domain\_prefix}.auth.{region}.[amazoncognito.com/logout?client\_id={client\_id}&logout\_uri={your\_host\_url}](http://amazoncognito.com/logout?client_id=%7Bclient_id%7D&logout_uri=%7Byour_host_url%7Dclient-id" \t "_blank)

[client-id](http://amazoncognito.com/logout?client_id=%7Bclient_id%7D&logout_uri=%7Byour_host_url%7Dclient-id" \t "_blank): # get this from AWS Cognito user pool management page

client-secret: # get this from AWS Cognito user pool management page

#### Simple Authentication ####

# authentication: simple

# admin-groups: admins

# users:

# - name: admin

# password: admin

# groups: admins

docker:

internal-networking: true

# Below is a list of Shiny app and their config (including who can access to which)

specs:

- id: euler

display-name: Euler's number

container-cmd: ["R", "-e", "shiny::runApp('/root/euler')"]

container-image: shiny-euler-app

access-groups: admins

container-network: sp-net

server:

useForwardHeaders: true # this is very important to make the AWS Cognito auth working

logging:

file:

shinyproxy.log

However, I will point out a couple of important things:

1. The default port for ShinyProxy is 8080 and we use it. If this is changed, the docker-compose files below need to be updated accordingly.
2. usage-stats-url need to updated to point to the AWS EC2 instance hosting the usage statistics tools. The ‘:8086/write?db=shinyproxy\_usagestats’ bit is important as it specifies the port of the InfluxDB and the name of the database (which we will create later).
3. All the items under openid need to be updated if you have set up AWS Cognito User Pool (i.e. replace the content in {} with the information you noted down from the AWS Cognito User Pool setting page). If you are not using it, simply comment out this block and uncomment the ‘Simple Authentication’ block and set it up accordingly.
4. Under the specs is the place you specify what Shiny apps should be available in your ShinyProxy portal. The container-cmd would depend on the name of the directory that contains your R Shiny app (e.g. the server.R and ui.R). container-image needs to be consistent with the name of the image you pushed to DockerHub. Note that when you pull, it will add YOUR\_USERNAME/ to the image by default. Finally access-groups defines who can access to the app. Make sure you include the group that the test account is in (e.g. admins).
5. Under server, the useForwardHeaders need to be set to ‘true’ to make AWS Cognito works properly.

Now, let’s move to app.conf, which is the Nginx configuration file.

app.conf

server {

listen 80;

server\_name [app.example.com](http://app.example.com);

server\_tokens off;

location /.well-known/acme-challenge/ {

root /var/www/certbot;

}

location / {

return 301 https://$host$request\_uri;

}

}

server {

listen 443 ssl;

server\_name [app.example.com](http://app.example.com);

access\_log /var/log/nginx/shinyproxy.access.log;

error\_log /var/log/nginx/shinyproxy.error.log error;

server\_tokens off;

ssl\_protocols TLSv1 TLSv1.1 TLSv1.2;

ssl\_certificate /etc/letsencrypt/live/[app.example.com/fullchain.pem](http://app.example.com/fullchain.pem);

ssl\_certificate\_key /etc/letsencrypt/live/[app.example.com/privkey.pem](http://app.example.com/privkey.pem);

include /etc/letsencrypt/options-ssl-nginx.conf;

ssl\_dhparam /etc/letsencrypt/ssl-dhparams.pem;

location / {

proxy\_pass <http://shinyproxy:8080>;

proxy\_http\_version 1.1;

proxy\_set\_header Upgrade $http\_upgrade;

proxy\_set\_header Connection "upgrade";

proxy\_read\_timeout 600s;

proxy\_redirect off;

proxy\_set\_header Host $http\_host;

proxy\_set\_header X-Real-IP $remote\_addr;

proxy\_set\_header X-Forwarded-For $proxy\_add\_x\_forwarded\_for;

proxy\_set\_header X-Forwarded-Proto $scheme;

}

}

You will need to change all the instances of [app.example.com](http://app.example.com) (4 in total) to your own domain name and that’s it! This configuration file instructs Nginx to listen port 80 (HTTP) for challenges (an important step for setting up the SSL certificate) and redirect traffic to port 443 (HTTPS), which has the certificate. Then the traffic will be directed to port 8080 of the local shinyproxy service where the magic happens. The full Nginx set up is fairly complicated and you can check the [official guide](https://www.nginx.com/resources/wiki/start/topics/examples/full/) to tailor it for your need.

Next, we will set up docker-compose.yml, which link everything together.

docker-compose.yml

version: '3.5'

services:

nginx:

image: nginx:1.15-alpine

depends\_on:

- shinyproxy

- certbot

restart: unless-stopped

volumes:

- ./data/nginx:/etc/nginx/conf.d

- ./data/certbot/conf:/etc/letsencrypt

- ./data/certbot/www:/var/www/certbot

ports:

- "80:80"

- "443:443"

networks:

- sp-net

command: "/bin/sh -c 'while :; do sleep 6h & wait $${!}; nginx -s reload; done & nginx -g \"daemon off;\"'"

certbot:

image: certbot/certbot

restart: unless-stopped

volumes:

- ./data/certbot/conf:/etc/letsencrypt

- ./data/certbot/www:/var/www/certbot

networks:

- sp-net

entrypoint: "/bin/sh -c 'trap exit TERM; while :; do certbot renew; sleep 12h & wait $${!}; done;'"

shinyproxy:

image: YOUR\_USERNAME/shinyproxy-example

depends\_on:

- euler

container\_name: shinyproxy

expose:

- 8080

volumes:

- ./application/application.yml:/opt/shinyproxy/application.yml

- /var/run/docker.sock:/var/run/docker.sock

networks:

- sp-net

euler:

image: YOUR\_USERNAME/shiny-euler-app

networks:

sp-net:

name: sp-net

The details of this set up (especially for Nginx and Certbot services) key things to point out:

1. You will need to update the two images YOUR\_USERNAME/shiny-euler-app and YOUR\_USERNAME/shinyproxy-example.
2. We need to link the Certbot and Nginx together by using two shared volumes ./data/certbot/conf:/etc/letsencrypt and ./data/certbot/www:/var/www/certbot. In this way, when Certbot obtains the certificate, Nginx is able to access it.
3. The code after entrypoint under the Certbot service is to make sure Certbot check if your certificate is up for renewal every 12 hours as recommended by Let’s Encrypt.
4. The code after command under the Nginx service is to make that Nginx reloads the newly obtained certificates every 6 hours.
5. We need to instruct docker-compose to create a container network called sp-net that links all the services together.

Finally, We need Nginx to perform the Let’s Encrypt validation. However, Nginx won’t start if the certificates are missing.. That is, we first create a dummy certificate, start Nginx, delete the dummy and request the real certificates using Certbot. This progress is automated with the below shell script.

init-letsencrypt.sh

#!/bin/bash

if ! [ -x "$(command -v docker-compose)" ]; then

echo 'Error: docker-compose is not installed.' >&2

exit 1

fi

domains=([app.example.com](http://app.example.com))

rsa\_key\_size=4096

data\_path="./data/certbot"

email="YOUR\_EMAIL" # Adding a valid address is strongly recommended

staging=0 # Set to 1 if you're testing your setup to avoid hitting request limits

if [ -d "$data\_path" ]; then

read -p "Existing data found for $domains. Continue and replace existing certificate? (y/N) " decision

if [ "$decision" != "Y" ] && [ "$decision" != "y" ]; then

exit

fi

fi

if [ ! -e "$data\_path/conf/options-ssl-nginx.conf" ] || [ ! -e "$data\_path/conf/ssl-dhparams.pem" ]; then

echo "### Downloading recommended TLS parameters ..."

mkdir -p "$data\_path/conf"

curl -s <https://raw.githubusercontent.com/certbot/certbot/master/certbot-nginx/certbot_nginx/_internal/tls_configs/options-ssl-nginx.conf> > "$data\_path/conf/options-ssl-nginx.conf"

curl -s <https://raw.githubusercontent.com/certbot/certbot/master/certbot/certbot/ssl-dhparams.pem> > "$data\_path/conf/ssl-dhparams.pem"

echo

fi

echo "### Creating dummy certificate for $domains ..."

path="/etc/letsencrypt/live/$domains"

mkdir -p "$data\_path/conf/live/$domains"

docker-compose run --rm --entrypoint "\

openssl req -x509 -nodes -newkey rsa:1024 -days 1\

-keyout '$path/privkey.pem' \

-out '$path/fullchain.pem' \

-subj '/CN=localhost'" certbot

echo

echo "### Starting nginx ..."

docker-compose up --force-recreate -d nginx

echo

echo "### Deleting dummy certificate for $domains ..."

docker-compose run --rm --entrypoint "\

rm -Rf /etc/letsencrypt/live/$domains && \

rm -Rf /etc/letsencrypt/archive/$domains && \

rm -Rf /etc/letsencrypt/renewal/$domains.conf" certbot

echo

echo "### Requesting Let's Encrypt certificate for $domains ..."

#Join $domains to -d args

domain\_args=""

for domain in "${domains[@]}"; do

domain\_args="$domain\_args -d $domain"

done

# Select appropriate email arg

case "$email" in

"") email\_arg="--register-unsafely-without-email" ;;

\*) email\_arg="--email $email" ;;

esac

# Enable staging mode if needed

if [ $staging != "0" ]; then staging\_arg="--staging"; fi

docker-compose run --rm --entrypoint "\

certbot certonly --webroot -w /var/www/certbot \

$staging\_arg \

$email\_arg \

$domain\_args \

--rsa-key-size $rsa\_key\_size \

--agree-tos \

--force-renewal" certbot

echo

echo "### Reloading nginx ..."

docker-compose exec nginx nginx -s reload

In this shell script, you will need to change [app.example.com](http://app.example.com) with your own domain name and enter your email address, so Let’s Encrypt can contact you if your certificate is about to expire or has any issue.

Once you are happy with the configuration files, you may want to commit and push it to your own GitHub repo for easy access. Now we have everything we need. Let’s log in to our ShinyProxy Machine and try it out!

**Step 2: Setting up ShinyProxy Machine**

SSH to the ShinyProxy Machine and git pull your own configuration repo shinyproxy-docker-compose-demo (or some other name you specified). The structure of the files and directory should remain the same. You should already have the Docker Engine and Docker Compose installed. Start the Docker service:

sudo service docker start

And check the services are running:

sudo docker version

sudo docker-compose version

And pull the Docker images. Note that you may need to login here (if you images are in private repos).

sudo docker pull YOUR\_USERNAME/shiny-euler-app

sudo docker pull YOUR\_USERNAME/shinyproxy-example

If all is running fine, CD into shinyproxy-docker-compose-demo. First, we will run the init-letsencrypt.sh, but not before we make it executable first. Follow the command line prompt to complete the progress, during which all the services will be up using docker-compose and if this is the first time you are running the script, it will take some time to build the images for Certbot and Nginx.

chmod +x init-letsencrypt.sh

Then run it with

sudo ./init-letsencrypt.sh

You will see something like below if the certificate is successfully issued.

Congratulations! You have successfully enabled <https://example.com> and <https://www.example.com>

-------------------------------------------------------------------------------------

IMPORTANT NOTES:

Congratulations! Your certificate and chain have been saved at:

/etc/letsencrypt/live/[example.com/fullchain.pem](http://example.com/fullchain.pem)

Your key file has been saved at:

/etc/letsencrypt/live/[example.com//privkey.pem](http://example.com/privkey.pem)

Your cert will expire on 2017-12-12.

Then, if you check the Docker containers using the command below, you will find shinyproxy, nginx and certbot running.

sudo docker stats

If not, simply docker-compose it up.

sudo docker-compose up

Give it a few seconds and visit [app.example.com](http://app.example.com). You should either see the AWS Cognito login page or the ShinyProxy login page depending on your setup.

If you decide to take the site down, just use the following command and everything will be nice and clean.

sudo docker-compose down

**Setting up InfluxDB, Telegraf and Grafana for usage statistics logging**

Setting up the usage statistics monitoring tools is relatively straightforward and can be done without scheduling downtime of your ShinyProxy Machine.

**Step 1: Setting up Telegraf on ShinyProxy Machine**

ShinyProxy has a built-in function that sends the basic usage information (login, sign-out, start/stop apps etc.) to InfluxDB. However, we also need to collect metrics (such as memory and CPU usage) from the host machine and Docker Engine. To achieve this, we will need to use Telegraf.

Again, we need to SSH to ShinyProxy Machine and create a new folder called telegraf in etc directory.

sudo mkdir -p /etc/telegraf

Now we will need to make sure the Docker service is running:

sudo service docker start

Similar to other services, Telegraf needs a configuration file to work properly and it can be quite complicated. Fortunately, we can create a template configuration file by running the config command from the Telegraf’s Docker container. The following command will download the telegraf image and run it to generate the default configuration file.

sudo docker run --rm telegraf telegraf config | sudo tee /etc/telegraf/telegraf.conf < /dev/null

Now, time to test your vim skill. We need to open the telegraf.conf and do two things. First, locate the [[outputs.influxdb]] section and update the urls to the ip/domain address of the Usage Monitor Machine. Don’t forget to specify the port 8086.

urls = ["http://{domain\_name}:8086"]

Then locate the HTTP Basic Auth section and modify the credentials accordingly if you do plan to set up credentials for your InfluxDB

username = ""

password = ""

Finally, locate the [[inputs.docker]] section, uncomment the block and set it up with the following settings:

# Read metrics about docker containers

[[inputs.docker]]

## Docker Endpoint

## To use TCP, set endpoint = "tcp://[ip]:[port]"

## To use environment variables (ie, docker-machine), set endpoint = "ENV"

endpoint = "unix:///var/run/docker.sock"

## Set to true to collect Swarm metrics(desired\_replicas, running\_replicas)

## Note: configure this in one of the manager nodes in a Swarm cluster.

## configuring in multiple Swarm managers results in duplication of metrics.

gather\_services = false

## Only collect metrics for these containers. Values will be appended to

## container\_name\_include.

## Deprecated (1.4.0), use container\_name\_include

container\_names = []

## Set the source tag for the metrics to the container ID hostname, eg first 12 chars

source\_tag = false

## Containers to include and exclude. Collect all if empty. Globs accepted.

container\_name\_include = []

container\_name\_exclude = []

## Container states to include and exclude. Globs accepted.

## When empty only containers in the "running" state will be captured.

## example: container\_state\_include = ["created", "restarting", "running", "removing", "paused", "exited", "dead"]

## example: container\_state\_exclude = ["created", "restarting", "running", "removing", "paused", "exited", "dead"]

## container\_state\_include = []

## container\_state\_exclude = []

## Timeout for docker list, info, and stats commands

timeout = "5s"

## Whether to report for each container per-device blkio (8:0, 8:1...) and

## network (eth0, eth1, ...) stats or not

perdevice = true

## Whether to report for each container total blkio and network stats or not

total = false

## docker labels to include and exclude as tags. Globs accepted.

## Note that an empty array for both will include all labels as tags

docker\_label\_include = []

docker\_label\_exclude = []

## Which environment variables should we use as a tag

tag\_env = ["JAVA\_HOME", "HEAP\_SIZE"]

## Optional TLS Config

## tls\_ca = "/etc/telegraf/ca.pem"

## tls\_cert = "/etc/telegraf/cert.pem"

## tls\_key = "/etc/telegraf/key.pem"

## Use TLS but skip chain & host verification

## insecure\_skip\_verify = false

Then we run the following command. Since telegraf is run within a container, the unix socket will need to be exposed within the telegraf container. This can be done in the docker CLI by add the option -v /var/run/docker.sock:/var/run/docker.sock.

sudo docker run -d -p 998:998 --name=telegraf \

--net=sp-net \

-e HOST\_PROC=/host/proc \

-v /var/run/docker.sock:/var/run/docker.sock \

-v /proc:/host/proc:ro \

-v /etc/telegraf/telegraf.conf:/etc/telegraf/telegraf.conf:ro \

telegraf

Run the command below to check if telegraf container is running.

sudo docker container logs -f --since 10m telegraf

**Step 2: Setting up InfluxDB on Usage Monitor Machine**

Now, SSH to the Usage Monitor Machine, which should already have Docker Engine running. InfluxDB installation should be fairly easy. If you launched your AWS EC2 instance using an Ubuntu 16.04 LTS system, InfluxDB can be downloaded and installed with the following commands:

wget <https://dl.influxdata.com/influxdb/releases/influxdb_1.8.0_amd64.deb>

sudo dpkg -i influxdb\_1.8.0\_amd64.deb

You can also use Docker image to build InfluxDB or even Docker Compose to set up, but I found the standalone version simple enough to set up and maintain. Once the installation is competed, start the service by:

sudo service influxdb start

And check if the service is running properly.

sudo service influxdb status

Now, let’s create two databases, one for ShinyProxy and one for Telegraf. Note that the name of the databases needs to be consistent with the settings in ShinyProxy application.yml and telegraf.conf. Open the InfluxDB CLI with:

influx

And then

CREATE DATABASE shinyproxy\_usagestats

CREATE DATABASE telegraf

If everything is working fine and your ShinyProxy Machine is running. You will be able to see data appears in the two databases shortly. Note that to see the data in shinyproxy\_usagestats you may need to log in to your app once. The query language InfluxDB uses is not so different from SQL but the data structure is different since it is a time-series database. I have also listed some below:

USE telegraf

SHOW FIELD KEYS

SHOW SERIES

USE shinyproxy\_usagestats

SHOW FIELD KEYS

SHOW SERIES

SELECT \* FROM event

quit

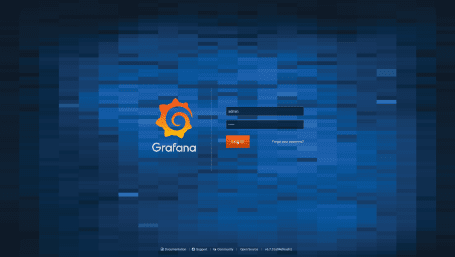
Of course, we are not going to monitor our apps using the InfluxDB CLI client, so let’s move on to the next part to set up Grafana!

**Step 3: Setting up Grafana on Usage Monitor Machine**

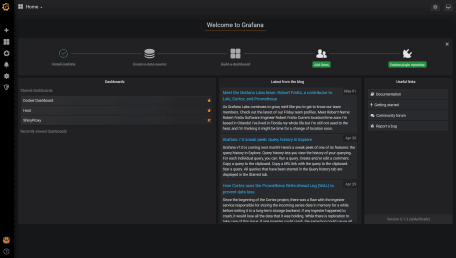
In the Usage Monitor Machine, let’s run Grafana using Docker.

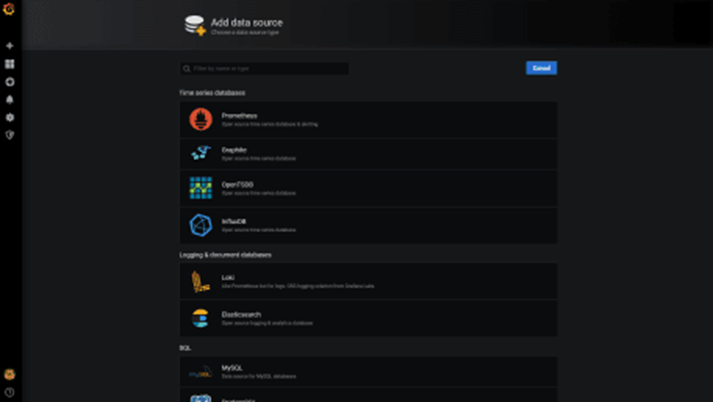
sudo docker run -d --name=grafana -p 3000:3000 grafana/grafana

Now, open your browser and head over to your Grafana homepage. If you are using AWS EC2 instance, it should be something like this: ‘[http://ec2-XX-XX-XX-XX.compute-1.amazonaws.com](http://ec2-xx-xx-xx-xx.compute-1.amazonaws.com/):3000’. Don’t forget the specify the port 3000 as that is the one listened by Grafana. You will then see the following login UI. The default username and password are both ‘admin’. When you log in for the first time, it will prompt you to change your password.

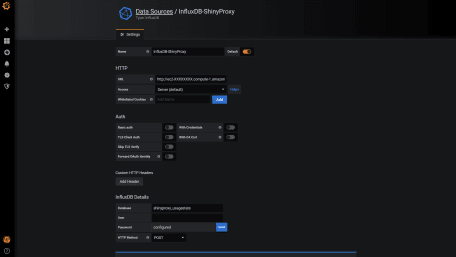


Once you have signed in, you will see the home page below. Now, click the Add data source button and choose InfluxDB when prompted.

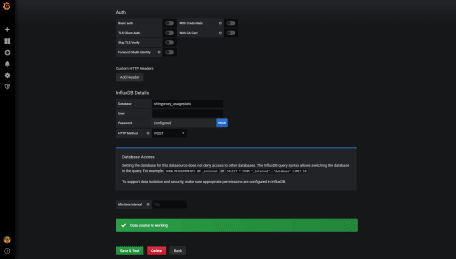




Below is the data source settings. The bare minimal settings we need to specify is the Name, URL and Database. URL should point to the address of your Usage Monitor Machine, followed by the 8086 port. ‘[http://ec2-XX-XX-XX-XX.compute-1.amazonaws.com](http://ec2-xx-xx-xx-xx.compute-1.amazonaws.com/):8086’. Let’s set the ShinyProxy first. So we will need to put shinyproxy\_usagestats for Database.

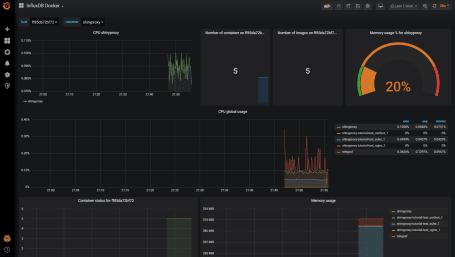


Once it is done, click the Save & Test button at the end of the page and you will see something like below that says your database is connected. Now, repeat the same process to connect to the Telegraf database (which is called telegraf if you have followed the tutorial), which stores the host system metrics and Docker stats.

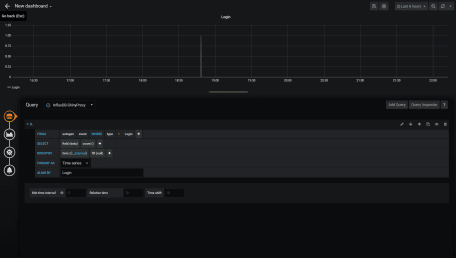


The next step is to build the dashboards to visualise the data. There are two ways we can do this. We can either build them from scratch or use the official & community built dashboards. For Docker and host metrics, there are well-established dashboards. To add them, you need to hover on the ‘Plus’ sign on the left and then ‘Import’. Then enter the dashboard ID, which you can find [here](https://grafana.com/grafana/dashboards?orderBy=name&direction=asc). Below are the two dashboards I like (1443 for host and 1150 for Docker). Once you have picked the dashboard you like, you need to specify the InfluxDB data source, which we have just set up.





At the time of the writing, there is no available Grafana dashboard for ShinyProxy. So we need to build one from scratch. To do so, hover on the ‘Plus’ sign again and then click ‘Dashboard’. A dashboard contains one or more panels and each panel consist of the query part and the visualisation part. We will start by clicking ‘Add Query’. Below is the panel editor. It contains a query builder that helps you to get started. Below is a simple query to monitor the login. You can also switch to the query text editing mode, which is more flexible in my opinion. There isn’t much information recorded by ShinyProxy but you have the basic such as login, sign-out, start and stop apps. More complicated queries can be built to show the number of users at the moment, how long each of them uses the apps and etc. I will look into this in the future. Meanwhile, feel free to explore yourself and contribute to the Grafana community by sharing your ShinyProxy dashboards!



That concludes this tutorial. We now have a functional Shiny app served by ShinyProxy with a secured host and monitoring capability. One thing we can further improve is the scalability of the app. Currently, it is served by a single AWS EC2 instance. Depending on the size of memory and number of vCPUs, it can serve one to a few dozen users simultaneously. However, we usually don’t expect users 24/7. As such, building an elastic stack using Docker Swarm or Kubernetes would be much more cost-efficient.