In case you are not familiar with AWS, it is a cloud services provider that among many other things, it provides access to virtual servers hosted on the cloud, so you don’t have to take care of a physical infrastructure yourself, also, to follow along with this article, there are some AWS specific terms that we need to define in simple terms:

**EC2 Instance**: It refers to an individual instance of a virtual server.

**Amazon Machine Images (AMI)**: It provides the information required to launch an instance, like the Operating System, pre-installed libraries and configurations required for a server to work. Note that you can make your own AMIs.

**Regions**: Amazon EC2 is hosted in multiple locations world-wide. These locations are composed of Regions, and each Region is a separate geographic area.

**Security Group**: A security group acts as a virtual firewall for your instance to control inbound and outbound traffic.

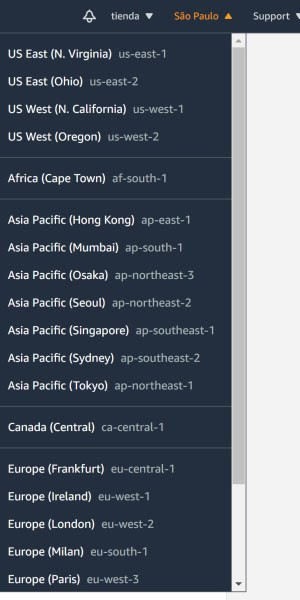
# Create an AWS Account and Security Credentials

In order to use AWS services you need to have an account, if you already have one, you can skip this part, if not, to create one go to https://aws.amazon.com/ and click on “Create an AWS Account”, fill in the required information and keep pressing “Continue”, you are going to be asked to provide your credit card information, but you are not going to be charged for using the products comprised on the “Free Tier” at least for a year.

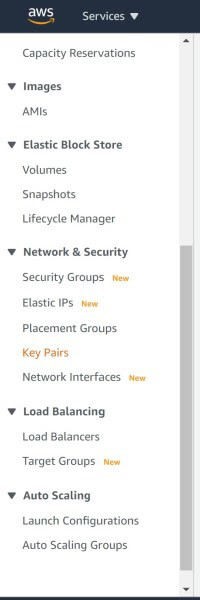
Once your account has been created, go to the “AWS Management Console” and log in, once there, click on the search button, type “EC2” and select “EC2 Virtual Servers in the Cloud”.



On the upper right corner choose a suitable region, the geographically closer to you the better. Remember your choice since you have to set this exact same location in the playbooks later.



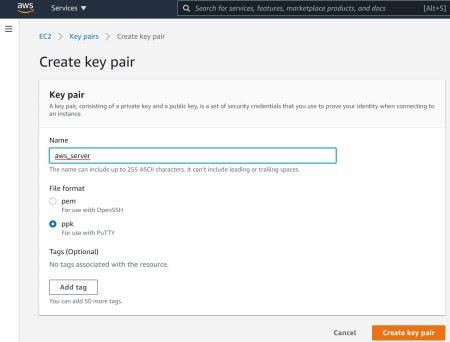
Then in the left menu, look for “Network & Security -> Key Pairs” and click on it.



On the right upper corner look for “Create Key Pair” and click on it.



You are going to see this dialog box, fill it this way, click on “Create Key Pair” and save the resulting file on this path, ~/.ssh/aws\_server.pem, in your machine.



Set propper permissions for your key file, the .ssh directory permissions should be 700 (rwx ) and the private key (aws\_server.pem) should be 600 (rw ).

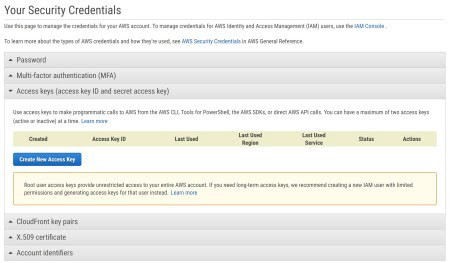
sudo chmod 700 ~/.ssh/

sudo chmod 600 ~/.ssh/aws\_server.pem

Next, you need to create an access key, so you can have access to your account programmatically, go to your account menu and select “My Security Credentials”.



Then select “Create New Access Key” and save the file with your credentials



 Keep the downloaded file safe and at hand, you are going to need it later. Be

careful not to share this file since it could enable other people to generate charges to your credit card.

# Install Ansible

It is possible to install Ansible locally on the EC2 instance itself and run playbooks directly on it with connection: local but it wouldn’t be practical, because among other things it would require us to manually launch the EC2 instance, it is better to run the playbooks from a system other than your EC2 instance.

For installing Ansible, if you have a Unix based system (i.e. Linux, macOS) at your disposal, you are golden, installing the latest Ansible version is very simple.

# Install python3

sudo apt install python3 # On macOS use `brew install python3` # Install pip3

sudo apt install python3-pip

# Install the latest Ansible version with pip3 sudo pip3 install ansible

If you are on Windows, sadly, there is no way to run Ansible natively, your best bet is to enable WSL (Windows Subsystem for Linux), install a Linux distribution from the Microsoft Store (I recommend Ubuntu), and execute the previous steps in your Linux VM.

Also, since we are going to be testing this with a low power EC2 instance (i.e. T2.Micro), some of these tasks are going to take a while and the SSH connection might get automatically closed due to inactivity, to prevent this situation, activate the sending of “keep-alive” packets to the server by editing the ssh\_config file with sudo nano /etc/ssh/ssh\_config and adding this two lines under Host \*:

ServerAliveInterval 300

ServerAliveCountMax 2

 Is very important not to skip the previous step, otherwise, the ssh connection is going to silently fail while running long tasks in Ansible and you are going to be waiting pointlessly with your EC2 instance doing nothing in reality.

# Download and Configure the Playbooks

The playbooks are on a public repository on GitHub, you can clone the repository with these commands:

# Install git if you don't have it already

sudo apt install git # On macOS use `brew install git`

To start configuring the playbooks, you usually first need to define an “inventory” (a list of servers to connect to), but in this case, the provision\_ec2\_instance.yml playbook is going to create one automatically for you when you run it and the other playbooks are going to get an updated inventory from AWS automatically using the aws\_ec2 plugging when needed,

this is one of the advantages of working with well supported cloud computing services like AWS, but in order for the aws\_ec2 plugging to do its magic, you need to provide it with your AWS credentials and define your AWS EC2 region, so edit the aws\_r\_server/inventories/aws\_ec2.yml file, and fill your credentials and region on the respective variables. Remember that you already created these credentials on a previous step and you saved a .csv file containing them, also the AWS Region you set here has to be the same one you selected when you started creating your credentials.

---

plugin: aws\_ec2

aws\_access\_key: 'your\_key\_goes\_here' aws\_secret\_key: 'your\_secret\_goes\_here'

regions:

* 'closest\_aws\_region' # e.g. sa-east-1

hostnames:

* 'ip-address' # Do not change this

keyed\_groups:

* key: tags.inventory\_group

Now, if you want to change the default installation settings, you can do it by editing the variables on the aws\_r\_server/vars/config\_vars.yml file, although, the default options are fine for most use case scenarios, including reasonable security settings to use on real-world applications. The only things you certainly need to change here are the AWS credentials and region, the email address for security notifications and the password for the PostgreSQL main user.

 This post has been written as an example of applying this approach with cloud computing services so for simplicity’s sake, I have made it work only with the Ubuntu Server 20.04 LTS AMI, so to be clear, it is not OS-agnostic. If you want to use it with an RHEL based AMI you would need to modify the playbooks considerably.

---

# AWS Config Vars ############################## ################################

# AWS credentials

aws\_access\_key: 'your\_key\_goes\_here' aws\_secret\_key: 'your\_secret\_goes\_here'

# General AWS configurations

aws\_region: 'closest\_aws\_region' # e.g. sa-east-1 aws\_ec2\_ami: ami-0c3c87b7d583d618f # Ubuntu 20.04 LTS

# List of instances instances:

* name: rstudio group: aws

security\_group: ["default", "public\_server"]

ssh\_key: aws\_server

# AWS firewall settings security\_groups:

* name: public\_server rules:
  + proto: tcp # http from\_port: 80

to\_port: 80 cidr\_ip: 0.0.0.0/0

* + proto: tcp # https from\_port: 443

to\_port: 443 cidr\_ip: 0.0.0.0/0

* + proto: tcp # SSH TCP from\_port: 22

to\_port: 22 cidr\_ip: 0.0.0.0/0

* + proto: udp # SSH UDP from\_port: 22

to\_port: 22 cidr\_ip: 0.0.0.0/0

* + proto: tcp # PostgreSQL from\_port: 5432

to\_port: 5432 cidr\_ip: 0.0.0.0/0

rules\_egress: []

# System Configurations ############################## ##########################

# Personal user

personal\_user: 'your\_linux\_user\_name' personal\_user\_password: 'very\_secure\_password'

# Swap parameters swap\_file\_path: /var/swap

# Use any of the following suffixes # c=1

# w=2

# b=512

# kB=1000 # K=1024

# MB=1000\*1000 # M=1024\*1024 # xM=M

# GB=1000\*1000\*1000 # G=1024\*1024\*1024

swap\_file\_size: 3GB swappiness: '10'

|  |  |  |
| --- | --- | --- |
| # Language and locale  language\_pack: [] | # | e.g. language-pack-es-base |
| default\_locale: 'en\_US.UTF-8' | # | e.g. es\_PE.UTF-8 |

# Security Configurations ############################## ########################

# Notification email for fail2ban send\_email: true

fail2ban\_email: 'your\_email@something.com'

# Postgresql Password

postgres\_password: 'very\_secure\_password'

# Access rules for Postgresql postgresql\_rules:

* { contype: local, users: all, address: samehost, method: trust }
* { contype: local, users: postgres, address: samehost, method: trust

}

* { contype: host, users: all, address: 0.0.0.0/0, method: password }

# Main Software Versions ############################## #########################

# Shiny-server version to install shiny\_server\_version: '1.5.16.958'

# RStudio version to install rstudio\_version: '1.4.1106' preview\_version: false

# Run the Playbooks

You could simply access the aws\_r\_server folder and run the main.yml playbook to install everything at once like this:

cd aws\_r\_server

ansible-playbook main.yml

But to make the installation process more flexible, I have divided the process into four individual playbooks:

ansible-playbook provision\_ec2\_instance.yml ansible-playbook install\_basic\_services.yml ansible-playbook install\_shiny\_server.yml ansible-playbook install\_rstudio\_server.yml

If you decide that you only need Shiny server or RStudio server but not the other, or you already have installed the support services you are going to use, then you can run only the playbooks you actually need.

If you want to update something in the future, like RStudio, or Shiny server (I can’t guaranty this is always going to work out of the box), you can simply change the version in the config file and

run the specific part of the playbook by taking advantage of the defined “tags”.

For example, this will only install the RStudio version defined in the config file and nothing more:

ansible-playbook install\_rstudio\_server.yml --tags "rstudio"

The available tags are:

install\_basic\_services.yml

secure: Set security settings on the server swap: Add swap memory to the server nginx: Install and configure Nginx + PHP

postgresql: Install and configure PostgreSQL

r: Install R from the CRAN repository install\_shiny\_server.yml

shiny-server: Install shiny-server

configure\_shiny: Configure shiny-server install\_rstudio\_server.yml

rstudio: Install RStudio server

configure\_rstudio: Configure RStudio server

You can also go the other way and skip specific parts of the playbooks by using the --skip- tags option, for example, if you don’t need PostgreSQL, you can avoid installing it by running the playbook this way:

ansible-playbook install\_basic\_services.yml --skip-tags "postgresql"

After successfully running all the playbooks, you will have a fully functional installation ready to be used, so you could simply open an RStudio session at http://your\_server\_ip/rstudio and/or publish your Shiny apps in the /srv/shiny-server folder and access them at http://your\_server\_ip/shiny/your\_app\_name.

# Final Notes

To finish I just want to make you aware that there are other options for getting RStudio and Shiny server on AWS that you might consider to be simpler, like using a premade AMI or containers, but they are not as flexible and customizable as defining your own infrastructure in code with Ansible. Obviously, the example in this article is my conception of a basic installation for an R-based data science server but you can use it as a starting point to custom tailor your own infrastructure and make your workflow more efficient as you become more proficient with Ansible.