SAVIOR Deployment Information

Version 1, April 2018

Wole Omitowoju

Wole.omitowoju@nextcentury.com

# Introduction

This document describes the deployment of the Savior system. The Savior system is made up of

* A Desktop Frontend. This is the component that the user logs into to launch an application with a virtue.
* A Virtue-Admin Server. This is the component that creates the different creates the different virtual machines that make up a virtue. The desktop frontend communicates with the virtue-admin server to launch an application.
* Sensing and response infrastructure

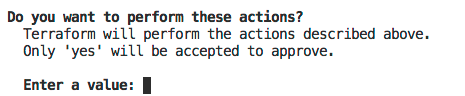
# Creating the AWS Infrastructure

The AWS infrastructure for Savior can be created using terraform. The terraform git project is in github - <https://github.com/NextCenturyCorporation/virtue-tf-apl>. The virtue-tf-apl terraform project creates the following resources in aws:

* Single VPC – “VIRTUE”
* A swarm of three sensors in three separate ec2 instances (m4.large)
* Virtue-admin server in a single ec2 instance (t2.xlarge).
* Route 53 resources
* Elastic ip address for virtue-admin server
* A gateway – “main IGW”
* A single subnet - “Public\_1a”
* Security groups

To create the various resources, you will need to run the terraform project. Checkout the project from github and do the following:

* Make sure you have an AWS profile set with the name APL that contains your access keys.
* From the virtue-tf-apl folder, run terraform apply. This will begin to setup the different resources that makeup the Savior system. You will be prompted with the following to accept the creation of the various resources:



# Deploying Sensor Infrastructure

The sensor ec2 instances are provisioned with the sensor infrastructure components already installed on each of the three instances. The terraform code that creates the ec2 instances is shown below. The AMI used for these instances are already stored in the AMI repo.

.

resource "aws\_instance" "twoSix\_virtue\_1" {

ami = "ami-4ee34631"

instance\_type = "m4.large"

key\_name = "twosix\_ec2"

subnet\_id = "${module.vpc.public\_1a\_id}"

vpc\_security\_group\_ids = ["${aws\_security\_group.default\_sg.id}", "${ aws\_security\_group.virtue\_internalports\_dev\_sg.id}"]

root\_block\_device {

volume\_type = "gp2"

volume\_size = "120"

delete\_on\_termination = true

}

tags {

Name = "Virtue Sensor Monitor 1"

}

lifecycle {

prevent\_destroy = false

}

}

resource "aws\_instance" "twoSix\_virtue\_2" {

ami = "ami-cce247b3"

instance\_type = "m4.large"

key\_name = "twosix\_ec2"

subnet\_id = "${module.vpc.public\_1a\_id}"

vpc\_security\_group\_ids = ["${aws\_security\_group.default\_sg.id}", "${ aws\_security\_group.virtue\_internalports\_dev\_sg.id}"]

root\_block\_device {

volume\_type = "gp2"

volume\_size = "120"

delete\_on\_termination = true

}

tags {

Name = "Virtue Sensor Monitor 2"

}

lifecycle {

prevent\_destroy = false

}

}

resource "aws\_instance" "twoSix\_virtue\_3" {

ami = "ami-cce247b3"

instance\_type = "m4.large"

key\_name = "twosix\_ec2"

subnet\_id = "${module.vpc.public\_1a\_id}"

vpc\_security\_group\_ids = ["${aws\_security\_group.default\_sg.id}", "${ aws\_security\_group.virtue\_internalports\_dev\_sg.id}"]

root\_block\_device {

volume\_type = "gp2"

volume\_size = "120"

delete\_on\_termination = true

}

tags {

Name = "Virtue Sensor Monitor 3"

}

lifecycle {

prevent\_destroy = false

}

}

Log into the sensor ec2 instance with name "Virtue Sensor Monitor 1" to deploy the sensor api.

Before deploying to the swarm or building the swarm network, *source* the swarm\_setup.sh script to prep the host environment:

. .savior/bin/swarm\_setup.sh

Instead of directly invoking the docker-compose command, we'll deploy the API as described by the docker-compose-swarm.yml compose file using the docker stack interface to the Swarm.

Deploy everything with:

sudo docker stack deploy --compose-file docker-compose-swarm.yml savior-api

You can check what's running in the service with:

sudo docker stack services savior-api

You can generally check that things are running smoothly by looking for errors in the API logs:

sudo docker service logs -f savior-api\_api

For debugging the current state of services, you can get a non-truncated PS result from the stack with:

sudo docker stack ps savior-api --no-trunc

Tear down the stack with:

sudo docker stack rm savior-api

Tear down the network

sudo docker network rm apinet

# Deploying Virtue-Admin Server

The virtue-admin server is responsible to creating ec2/xen instances that make up a virtue. The desktop frontend communicates with the virtue-admin server to launch various application within a virtue.

The ec2 instance that runs the virtue-admin is created by the following terraform:

resource "aws\_instance" "ncc\_virtue-admin" {

ami = "ami-4f87273"

instance\_type = "t2.xlarge"

key\_name = "vrtu"

subnet\_id = "${module.vpc.public\_1a\_id}"

vpc\_security\_group\_ids = ["${aws\_security\_group.default\_sg.id}", "${ aws\_security\_group.virtue\_internalports\_dev\_sg.id}", "${ aws\_security\_group.virtue\_admin\_server\_internal\_sg.id}", "${ aws\_security\_group.virtue\_admin\_server\_external\_sg.id}"]

root\_block\_device {

volume\_type = "gp2"

volume\_size = "120"

delete\_on\_termination = true

}

disable\_api\_termination = true

tags {

Name = "Virtue BackEnd Admin"

}

lifecycle {

prevent\_destroy = false

}

}

The virtue-admin server is created from an AMI preinstalled with the virtue-admin java application.

To start the virtue-admin

cd /opt/savior/virtue-admin-0.0.1-SNAPSHOT

Run

nohup bin/virtue-admin

# Connecting the Desktop the virtue-admin Server

The Savior desktop runs on the end user system. It needs to be configured to communicate with the virtue-admin server.

To connect to the server, edit the savior-user.properties file of the Savior destkotp and change the savior.api.path.base parameter to point to the address of the virtue-admin server e.g.

savior.api.path.base=http://ec2-34-197-219-203.compute-1.amazonaws.com:8080/

If the system is configured properly, you will see a list of virtues configured for the the user displayed in the desktop application.

# Appendix – Route 53

Below is the terraform configuration for route 53

resource "aws\_route53\_zone" "savior" {

name = "savior.internal"

vpc\_id = "${module.vpc.vpc\_id}"

}

resource "aws\_route53\_record" "sensing-api" {

zone\_id = "${aws\_route53\_zone.savior.zone\_id}"

name = "sensing-api.savior.internal"

type = "CNAME"

ttl = "300"

records = ["${aws\_instance.twoSix\_virtue\_1.private\_dns}"]

}

resource "aws\_route53\_record" "sensing-ca" {

zone\_id = "${aws\_route53\_zone.savior.zone\_id}"

name = "sensing-ca.savior.internal"

type = "CNAME"

ttl = "300"

records = ["${aws\_instance.twoSix\_virtue\_1.private\_dns}"]

}

resource "aws\_route53\_record" "sensing-kafka" {

zone\_id = "${aws\_route53\_zone.savior.zone\_id}"

name = "sensing-kafka.savior.internal"

type = "CNAME"

ttl = "300"

records = ["${aws\_instance.twoSix\_virtue\_1.private\_dns}"]

}

# APPENDIX – VPC Configuration

The VPC terraform configuration

module "vpc" {

source = "./vpc"

vpc\_cidr = "10.0.0.0/16"

vpc\_public\_1a\_subnet = "10.0.4.0/23"

vpc\_name = "VIRTUE"

}

resource "aws\_route\_table" "public\_routes" {

vpc\_id = "${module.vpc.vpc\_id}"

route {

cidr\_block = "0.0.0.0/0"

gateway\_id = "${module.vpc.igw\_id}"

}

}

resource "aws\_route\_table\_association" "public\_1a\_routes" {

subnet\_id = "${module.vpc.public\_1a\_id}"

route\_table\_id = "${aws\_route\_table.public\_routes.id}"

}