How to set up a Private Docker Registry using AWS S3

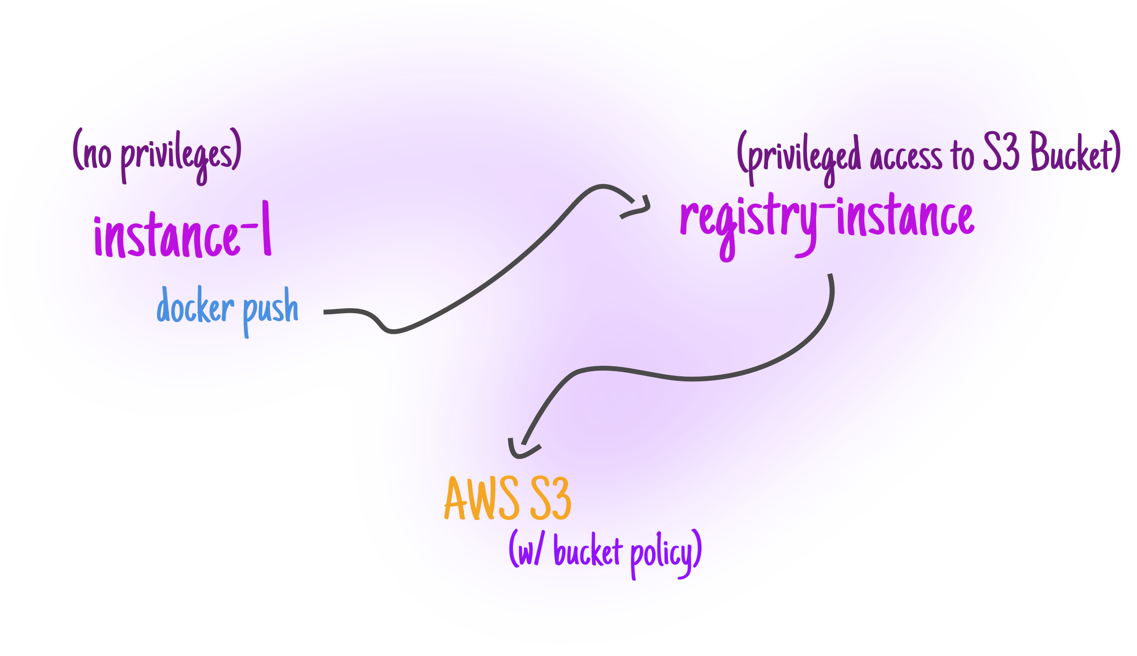
Go through the process of creating a Terraform configuration for deploying a Docker registry to an instance making use of IAM roles.

by Ciro S. Costa - Jun 23, 2018

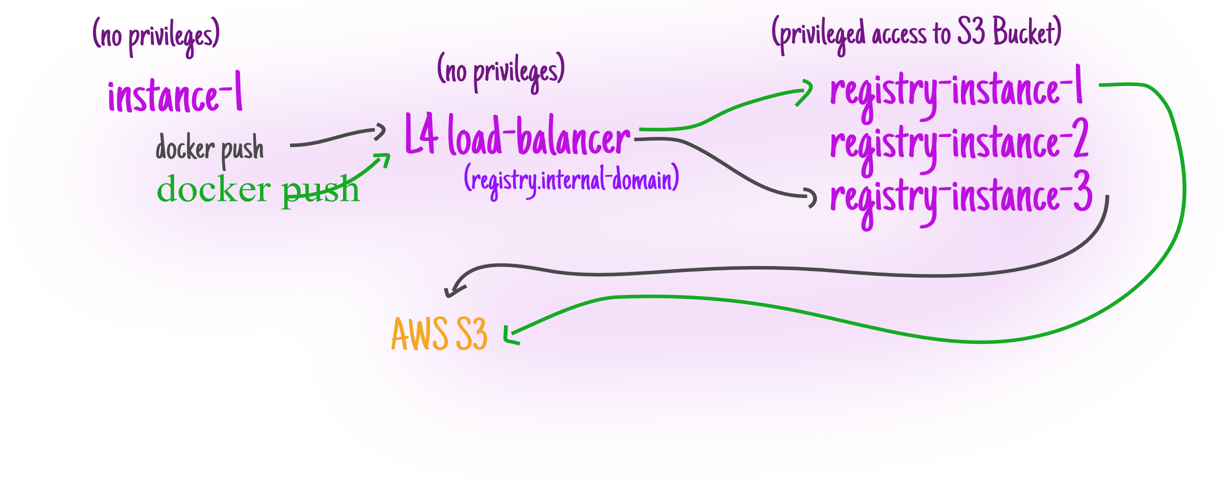
tags: aws , docker

Hey,

Some time ago I needed to check whether AWS EC2 instance profiles would work fine with Docker Registry, and guess what? It does!



What interests me the most in such setup is how it facilitates a scenario where you can have a highly available internal registry, having a load-balancer in front of a set of registries and granting privileges to only those who really need it.



Given that the pushes to the registry (and pulls from it) usually involve large amounts of data being transferred, with a well-done load-balancing we can scale the throughput with ease.

Here's a setup that makes use of `terraform` and the official Docker Registry image to achieve the first scenario.

The terraform setup

By taking a top-down approach, the infrastructure resources that Terraform is meant to create is composed of only two top-level resources:

- the instance that will hold the docker registry; and
- the S3 bucket that is meant to store the registry data.

The bucket can be created using the policy specified in the official registry documentation:

```
# Create a AWS S3 bucket that is encrypted by default
# at the server side using the name provided by the
# `bucket` variable.
#
# Given that we're not specifying an ACL, by default
# the `private` canned ACL is used, which means that
# only the owner gets FULL_CONTROL access (and no
# one else).
resource "aws_s3_bucket" "encrypted" {
  bucket        = "${var.bucket}"
  force_destroy = true

  server_side_encryption_configuration {
    rule {
      apply_server_side_encryption_by_default {
        sse_algorithm = "AES256"
      }
    }
  }
}

# Set up the bucket policy to allow only a
# specific set of operations on both the root
# of the bucket as well as its subdirectories.
#
# Here we also explicitly set who's able to have
# such capabilities - those that make use of the
# role that we defined in `permissions.tf`.
resource "aws_s3_bucket_policy" "main" {
  bucket = "${var.bucket}"

  policy = <<POLICY
{
    "Statement": [
        {
            "Action": [
                "s3:PutObject",
                "s3:GetObject",
                "s3:DeleteObject",
                "s3:ListMultipartUploadParts",
                "s3:AbortMultipartUpload"
            ],
            "Effect": "Allow",
            "Principal": {
                "AWS": "${aws_iam_role.main.arn}"
            },
            "Resource": "arn:aws:s3:::${var.bucket}/*",
            "Sid": "AddCannedAcl"
        },
        {
            "Action": [
                "s3:ListBucket",
                "s3:GetBucketLocation",
                "s3:ListBucketMultipartUploads"
            ],
            "Effect": "Allow",
            "Principal": {
                "AWS": "${aws_iam_role.main.arn}"
            },
            "Resource": "arn:aws:s3:::${var.bucket}"
        }
    ],
    "Version": "2012-10-17"
}
POLICY
}
```

Naturally, there's one piece there in the policies that we didn't create yet at this point - the role to which we grant access to.

This role is the one that we're going to ship in the form of an instance profile to the EC2 instances such that they can have access to the S3 buckets.

```
# The bucket-root policy defines the API actions that are
# allowed to take place on the bucket root directory.
#
# Given that by default nothing is allowed, here we list
# the actions that are meant to be allowed.
#
# This list of actions that are required by the Docker Registry
# can be found in the official Docker Registry documentation.
data "aws_iam_policy_document" "bucket-root" {
  statement {
    effect = "Allow"

    actions = [
      "s3:ListBucket",
      "s3:GetBucketLocation",
      "s3:ListBucketMultipartUploads",
    ]

    resources = [
      "arn:aws:s3:::${var.bucket}",
    ]
  }
}

# The bucket-subdirs policy defines the API actions that
# are allowed to take place on the bucket subdirectories.
#
# Given that by default nothing is allowed, here we list
# the actions that are meant to be allowed.

# Just like the policy for the root directory of the bucket,
# this list of necessary actions to be allowed can be found in
# the official Docker Registry documentation.
data "aws_iam_policy_document" "bucket-subdirs" {
  statement {
    effect = "Allow"

    actions = [
      "s3:PutObject",
      "s3:GetObject",
      "s3:DeleteObject",
      "s3:ListMultipartUploadParts",
      "s3:AbortMultipartUpload",
    ]

    resources = [
      "arn:aws:s3:::${var.bucket}/*",
    ]
  }
}

# Define a base policy document that dictates which principals
# the roles that we attach to this policy are meant to affect.
#
# Given that we want to grant the permissions to an EC2 instance,
# the pricipal is not an account, but a service: `ec2`.
#
# Here we allow the instance to use the AWS Security Token Service
# (STS) AssumeRole action as that's the action that's going to
# give the instance the temporary security credentials needed
# to sign the API requests made by that instance.
data "aws_iam_policy_document" "default" {
  statement {
    effect = "Allow"

    actions = [
      "sts:AssumeRole",
    ]

    principals {
      type = "Service"

      identifiers = [
        "ec2.amazonaws.com",
      ]
    }
  }
}

# Creates an IAM role with the trust policy that enables the process
# of fetching temporary credentials from STS.
#
# By tying this trust policy with access policies, the resulting
# temporary credentials generated by STS have only the permissions
# allowed by the access policies.
resource "aws_iam_role" "main" {
  name               = "default"
  assume_role_policy = "${data.aws_iam_policy_document.default.json}
}

# Attaches an access policy to that role.
resource "aws_iam_role_policy" "bucket-root" {
  name   = "bucket-root-s3"
  role   = "${aws_iam_role.main.name}"
  policy = "${data.aws_iam_policy_document.bucket-root.json}"
}

# Attaches an access policy to that role.
resource "aws_iam_role_policy" "bucket-subdirs" {
  name   = "bucket-subdirs-s3"
  role   = "${aws_iam_role.main.name}"
  policy = "${data.aws_iam_policy_document.bucket-subdirs.json}"
}

# Creates the instance profile that acts as a container for that
# role that we created that has the trust policy that is able
# to gather temporary credentials using STS and specifies the
# access policies to the bucket.
#
# This instance profile can then be provided to the aws_instance
# resource to have it at launch time.
resource "aws_iam_instance_profile" "main" {
  name = "instance-profile"
  role = "${aws_iam_role.main.name}"
}
```

With the permissions created, as well as the bucket, what's left now is creating the instance with its corresponding networking and AMI:

```
provider "aws" {
  region  = "${var.region}"
  profile = "${var.profile}"
}

# We could create an extra VPC, properly set a subnet and
# have the whole thing configured (internet gateway, an updated
# routing table etc).
#
# However, given that this is only a sample, we can make use of
# the default VPC (assuming you didn't delete your default VPC
# in your region, you can too).
data "aws_vpc" "main" {
  default = true
}

# Provide the public key that we want in our instance so we can
# SSH into it using the other side (private) of it.
resource "aws_key_pair" "main" {
  key_name_prefix = "sample-key"
  public_key      = "${file("./keys/key.rsa.pub")}"
}

# Create a security group that allows anyone to access our
# instance's port 5000 (where the main registry functionality
# lives).
#
# Naturally, you'd not do this if you're deploying a private
# registry - something you could do is allow the internal cidr
# and not 0.0.0.0/0.
resource "aws_security_group" "allow-registry-ingress" {
  name = "allow-registry-ingress"

  description = "Allows ingress SSH traffic and egress to any addres
  vpc_id      = "${data.aws_vpc.main.id}"

  ingress {
    from_port   = 5000
    to_port     = 5000
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  tags {
    Name = "allow_registry-ingress"
  }
}

# Allow SSHing into the instance
resource "aws_security_group" "allow-ssh-and-egress" {
  name = "allow-ssh-and-egress"

  description = "Allows ingress SSH traffic and egress to any addres
  vpc_id      = "${data.aws_vpc.main.id}"

  ingress {
    from_port   = 22
    to_port     = 22
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  egress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }

  tags {
    Name = "allow_ssh-and-egress"
  }
}

# Template the registry configuration with the desired
# bucket and region information.
#
# Note.: no credentials are needed given that the golang
# aws sdk is going to retrieve them from the instance
# role that we set to the machine.
data "template_file" "registry-config" {
  template = "${file("./registry.yml.tpl")}"

  vars {
    bucket = "${var.bucket}"
    region = "${var.region}"
  }
}

# Template the instance initialization script with information
# regarding the region and bucket that the user configured.
data "template_cloudinit_config" "init" {
  gzip          = true
  base64_encode = true

  part {
    content_type = "text/cloud-config"

    content = <<EOF
#cloud-config
write_files:
  - content: ${base64encode("${data.template_file.registry-config.re
    encoding: b64
    owner: root:root
    path: /etc/registry.yml
    permissions: '0755'
EOF
  }

  part {
    content_type = "text/x-shellscript"
    content      = "${file("./instance-init.sh")}"
  }
}

# Create an instance in the default VPC with a specified
# SSH key so we can properly SSH into it to verify whether
# everything is worked as intended.
resource "aws_instance" "main" {
  instance_type        = "t2.micro"
  ami                  = "${data.aws_ami.ubuntu.id}"
  key_name             = "${aws_key_pair.main.id}"
  iam_instance_profile = "${aws_iam_instance_profile.main.name}"
  user_data            = "${data.template_cloudinit_config.init.rend

  vpc_security_group_ids = [
    "${aws_security_group.allow-ssh-and-egress.id}",
    "${aws_security_group.allow-registry-ingress.id}",
  ]
}

output "public-ip" {
  description = "Public IP of the instance created"
  value       = "${aws_instance.main.public_ip}"
}
```

Now, aside from the Terraform setup that has already been done at this point, the next step is configuring the registry itself.

As we're using cloud-init's `write_files` to send our templated registry configuration, this is how such configuration looks like:

```yaml
# -*- mode: yaml -*-
# vi: set ft=yaml :
#
# Template of a registry configuration.
#
# It expects the folling variables:
# - region: AWS region where the S3 bucket is deployed to; and
# - bucket: the name of the bucket to push registry data to.
#
# Without modifying the default command line arguments of the
# registry image, put this configuration under
# `/etc/docker/registry/config.yml`.
version: 0.1
log:
  level: "debug"
  formatter: "json"
  fields:
    service: "registry"
storage:
  cache:
    blobdescriptor: "inmemory"
  s3:
    region: "${region}"
    bucket: "${bucket}"
http:
  addr: ":5000"
  secret: "a-secret"
  debug:
    addr: ":5001"
    # In the `master` branch there's already native support for
    # Prometheus metrics.
    #
    # If you want to make use of this feature:
    # 1. clone `github.com/docker/distribution`;
    # 2. build the image (`docker build -t registry .`);
    # 3. reference the recently built image in the initialization.
    prometheus:
      enabled: true
      path: "/metrics"
  headers:
    X-Content-Type-Options: [ "nosniff" ]
health:
  storagedriver:
    enabled: true
    interval: "10s"
    threshold: 3
```

Update: Note that there I specified a Prometheus endpoint - that's because in a future release the Docker registry will be able to export metrics to prometheus; if you're curious about it, checkout how to retrieve metrics using prometheus in which I tie these metrics with the newest Grafana Prometheus heatmap support that just landed in `Grafana v5.1` .

And that's it! Create an instance initialization script and there you go! You have a registry backed by S3.

Such script doesn't need to be complicated - something like the following would do it:

```
# Run docker registry with the configuration provided
# at `/etc/registry.yml` as a readonly volume mount
# to the registry container.
#
# Here I make use of the host networking mode (no extra
# network namespace is created), but you're free to make
# use of port forwarding as you wish.
docker run \
        --detach \
        --volume /etc/registry.yml:/etc/docker/registry/config.yml:r
        --network host \
        registry
```

Note that all the setup is independent of a particular container scheduler.

In the case of a scheduler like Kubernetes or Swarm you could provide the proper labels to the instance and make sure that the registry container gets scheduled to such instance type.

The whole code that's been described here can be found in the cirocosta/sample-docker-registry-aws repository.

If you have any questions or found something odd, feel free to get in touch! I'm cirowrc on Twitter.

Have a good one!

finis

Recommended articles

If you've gotten some knowledge from this article, these are some others that you might take advatange of as well!

- A practical look at basic AWS Networking with Terraform
- Blocking ingress traffic to Docker swarm worker machines
- Blocking EC2 Metadata service from Docker containers in AWS
- Compiling and running your own forked Docker release
- Adding privileged containers to Docker Swarm mode

Stay in touch!

From time to time I'll deliver some content to you.

The emails are not automatic - it's all about things I thought were worth sharing that I'd personally like to receive.

JOIN THE GROUP

If you're into Twitter, reach me at @cirowrc.

About Tags Advertise Twitter GitHub LinkedIn