Building A Static Website in AWS (Using only Infrastructure-As-Code Tools)

Introduction

In this document I will show you how to set up a static HTML site using AWS as the platform and IaC tolls such as Terraform, Packer and Ansible. We will use the AWS Console only when absolutely necessary, such as creating credentials. Prior knowledge of Linux shell and commands are assumed.

We will use Packer and Ansible to create an AMI with the static HTML built in and served via Nginx. Then we will use Terraform to take that AMI and provision it as a load-balanced and autoscaled group in AWS which will have a Route53 public URL associated with it.

For added security we will need separate AWS access keys for Packer and Terraform

Prerequisites:

You will need Ansible, Packer and Terraform installed on your local computer. Ansible can be installed via pip (pip install ansible). Packer and Terraform can be install by downloading the zip files from Hashicorp's website and unzipping into your system's $PATH which is usually /usr/bin

Building The Image

Here we use packer to provision an AMI using Ansible. There is a simple index.html file in the same directory as the playbook.yml and packer.json
Modify the index.html to your liking.

Step 1: Create AWS Access Credentials for building the Image

Create AWS credentials for Packer, let's call it *buildImage* and limit it's permissions to only EC2 instances. There's no need for it to access other AWS resources. On the AWS IAM Console create the "buildImage" user and give it the AmazonEC2FullAccess policy. Only give it Programmatic Access. No need to give it Console Access. Also download credentials csv file.

Step 2: Set up credentials file

Create the AWS config file in your home directory $HOME/.aws/credentials and place the credentials file in it as follows:

```
[buildImage]
aws_access_key_id=<put your access key here>
aws_secret_access_key=<put your secret key here>
```

We create a separate [buildImage] profile since we want Packer and Terraform to use different profiles for accessing AWS resources. We could have used environment variables AWS_ACCESS_KEY_ID and AWS_SECRET_ACCESS_KEY but then you'd have to swap them out every time you want to run Packer or Terraform.

Step 3: Run packer

Run Packer validation first to ensure image would be built properly. cd into the buildImage directory and run packer validations.

```
packer validate packer.json
```

If there are no errors, you can then go ahead and build the image, using the command:

```
Packer build packer.json
```

If all goes well, we will see the AMI name at the end. We can now use that with Terraform.

Build the Infrastructure

Go to AWS IAM console and create another IAM user with relevant policies. Here I gave it Administrator policy for simplicity but you should use a more secure policy. Refer to the Terraform website for more information. Download it's keys as before. This time set up the keys as environment variables AWS_ACCESS_KEY_ID and AWS_SECRET_ACCESS_KEY

Terraform will now be able to authenticate to AWS with those credentials.

Now that we created the custom AMI, we can now create the infrastructure. In the buildInfra directory, run the following commands:

```
terraform validate -var 'ami=<the-ami-you-created>'
```

This will perform a "test run" showing you what changes it would make to your infrastructure, usually "create", "update" or "destroy"

If there are no errors, you can then run:

```
terraform init
terraform plan -var 'ami=<the-ami-you-created>'
```

Terraform init will set up your manifests via the relevant plugin which is AWS in this case; and terraform plan will show you what would happen if you actually run this step. If there are no errors, you can run the following

terraform apply -var 'ami=<the-ami-you-created>'

This will actually apply the changes to the infrastructure. In the end you should see the URL pointing to your application

Updating the Application/Infrastructure (Rolling Update)

To update the application, simply modify the index.html file and run the packer tool as before. Once you get your new ami, you can follow the steps as for updating your infrastructure with the new AMI as above (see 'Build the Infrastructure').

The difference here is that when you run terraform plan is it will show you that it is going to destroy 2 resources which is the launch config and the autoscale group. It's going to replace them appropriately.

Note that this will perform a **Rolling Update** because the autoscale group and launch configuration have create_before_destroy set. Also the launch configuration has no "name" attribute so terraform will create a new launch configuration for the new AMI. The autoscale group incorporates the launch configuration into the name causing it to use the new launch config. Finally, wait_for_elb_capacity in the autoscale group causes it to wait until the ELB has raised the appropriate number of new healthy instances (based upon the new AMI) before destroying the old autoscale group and old launch config.