Production: Create Staging Stack

We'll cover the following

Objective

- Create separate environments for staging.

Steps

- Extract common resources out of `main.yml`.
- Create a separate stack for staging.

Creating a nested stack for staging

Next, we're going to move all the resources we've created for our staging environment into a separate staging stack. We'll do this by extracting the staging resources into a file called `stage.yml`. (Here, 'stage' refers to a deployment step, not to the staging environment itself.)

To perform this split, it's easier if we start by copying the whole `main.yml` file.

```
cp main.yml stage.yml
```

terminal

Now, let's delete the following resources from `stage.yml`:

- `DeploymentRole`

- `BuildProject`

- `DeploymentApplication`
- `StagingDeploymentGroup`
- `Pipeline`
- `PipelineWebhook`

And let’s delete everything that is not in the above list from `main.yml`.

> In the main.yml file, you need to update the parameter EC2AMI type to `String` and remove the `Default` type.

We also need to delete the following input parameters from `stage.yml`:

- `CodePipelineBucket`
- `GitHubOwner`
- `GitHubRepo`
- `GitHubBranch`
- `GitHubPersonalAccessToken`

Next, we’re going to add a nested stack named `Staging` to `main.yml` as an instance of our new `stage.yml` template.

```
Staging:
  Type: AWS::CloudFormation::Stack
  Properties:
    TemplateURL: stage.yml
    TimeoutInMinutes: 30
    Parameters:
      EC2InstanceType: !Ref EC2InstanceType
      EC2AMI: !Ref EC2AMI
```

main.yml

Now we need to add outputs in `stage.yml` so that the `main.yml` stack knows about the load balancer endpoints and the ASG of the stage.

```
Outputs:
  LBEndpoint:
    Description: The DNS name for the LB
    Value: !Sub "http://${LoadBalancer.DNSName}:80"
  ScalingGroup:
    Description: The ScalingGroup for this stage
    Description: The ScalingGroup for this stage
    Value: !Ref ScalingGroup
```

stage.yml

> We don't need `Export` properties in `stage.yml`. This is because `stage.yml` will be referenced only by the `main.yml` stack, and parent stacks can access the output variables of nested stacks directly.

Now we need to change the `StagingDeploymentGroup` resource to refer to the output from the staging nested stack.

```
StagingDeploymentGroup:
  Type: AWS::CodeDeploy::DeploymentGroup
  Properties:
    DeploymentGroupName: staging
    AutoScalingGroups:
    - !GetAtt Staging.Outputs.ScalingGroup
    ApplicationName: !Ref DeploymentApplication
    DeploymentConfigName: CodeDeployDefault.AllAtOnce
    ServiceRoleArn: !GetAtt DeploymentRole.Arn
```

main.yml

Line #6: Refers to the ASG that the staging stack returns.

We also need to change the endpoint that `main.yml` returns.

```
StagingLBEndpoint:
  Description: The DNS name for the staging LB
  Value: !GetAtt Staging.Outputs.LBEndpoint
  Export:
    Name: StagingLBEndpoint
```

main.yml

At this point, we need to deal with one of CloudFormation's quirks. Nested stacks must be referenced as S3 URLs. To deal with this, we can use CloudFormation packaging to help us upload and transform our templates.

But first, we'll need an S3 bucket to store our packaged templates. This is another thing that can go into our `setup.yml` template, so let's add the input parameter first.

```
CloudFormationBucket:
  Type: String
  Description: 'The S3 bucket for CloudFormation templates.'
```

setup.yml

And then let’s add the resource for the S3 bucket.

```
CloudFormationS3Bucket:
  Type: AWS::S3::Bucket
  DeletionPolicy: Retain
  Properties:
    BucketName: !Ref CloudFormationBucket
    PublicAccessBlockConfiguration:
      BlockPublicAcls: true
      BlockPublicPolicy: true
      IgnorePublicAcls: true
      RestrictPublicBuckets: true
    BucketEncryption:
      ServerSideEncryptionConfiguration:
        - ServerSideEncryptionByDefault:
            SSEAlgorithm: AES256
```

setup.yml

Next, we’ll add an environment variable in `deploy-infra.sh` to define the S3 bucket name for the packaged CloudFormation templates.

```
CFN_BUCKET="$STACK_NAME-cfn-$AWS_ACCOUNT_ID"
```

deploy-infra.sh

And finally, we’re going to pass the bucket name as a parameter when we deploy `setup.yml`.

```
# Deploys static resources
echo -e "\n\n=========== Deploying setup.yml ==========="
aws cloudformation deploy \
  --region $REGION \
  --profile $CLI_PROFILE \
  --stack-name $STACK_NAME-setup \
  --template-file setup.yml \
  --no-fail-on-empty-changeset \
  --capabilities CAPABILITY_NAMED_IAM \
  --parameter-overrides \
    CodePipelineBucket=$CODEPIPELINE_BUCKET \
    CloudFormationBucket=$CFN_BUCKET
```

deploy-infra.sh

Line #12: Pass in the bucket used to store packaged CloudFormation resources.

Between the deploy commands for `setup.yml` and `main.yml` in our `deploy-

Between the deploy commands for `setup.yml` and `main.yml` in our `deploy-infra.sh` script, we also need to add a new set of commands to package our nested stacks.

```
# Package up CloudFormation templates into an S3 bucket
echo -e "\n\n=========== Packaging main.yml ==========="
mkdir -p ./cfn_output

PACKAGE_ERR="$(aws cloudformation package \
  --region $REGION \
  --profile $CLI_PROFILE \
  --template main.yml \
  --s3-bucket $CFN_BUCKET \
  --output-template-file ./cfn_output/main.yml 2>&1)"

if ! [[ $PACKAGE_ERR =~ "Successfully packaged artifacts" ]]; then
  echo "ERROR while running 'aws cloudformation package' command:"
  echo $PACKAGE_ERR
  exit 1
fi
```

deploy-infra.sh

Line #11: This will write the packaged CloudFormation template to `/cfn_output/main.yml`.

We now need to change the deploy command for `main.yml` in `deploy-infra.sh` to refer to the packaged template file.

```
# Deploy the CloudFormation template
echo -e "\n\n=========== Deploying main.yml ==========="
aws cloudformation deploy \
  --region $REGION \
  --profile $CLI_PROFILE \
  --stack-name $STACK_NAME \
  --template-file ./cfn_output/main.yml \
  --no-fail-on-empty-changeset \
  --capabilities CAPABILITY_NAMED_IAM \
  --parameter-overrides \
    EC2InstanceType=$EC2_INSTANCE_TYPE \
    GitHubOwner=$GH_OWNER \
    GitHubRepo=$GH_REPO \
    GitHubBranch=$GH_BRANCH \
    GitHubPersonalAccessToken=$GH_ACCESS_TOKEN \
    CodePipelineBucket=$CODEPIPELINE_BUCKET
```

deploy-infra.sh

Line #7: The output of the `aws cloudformation package` command.

Finally, we need to change the section of `deploy-infra.sh` that prints the endpoint

URLs so that it catches both our staging endpoint, as well as the forthcoming prod endpoint.

```
# If the deploy succeeded, show the DNS name of the endpoints
if [ $? -eq 0 ]; then
  aws cloudformation list-exports \
    --profile awsbootstrap \
    --query "Exports[?ends_with(Name,'LBEndpoint')].Value"
fi
```

deploy-infra.sh

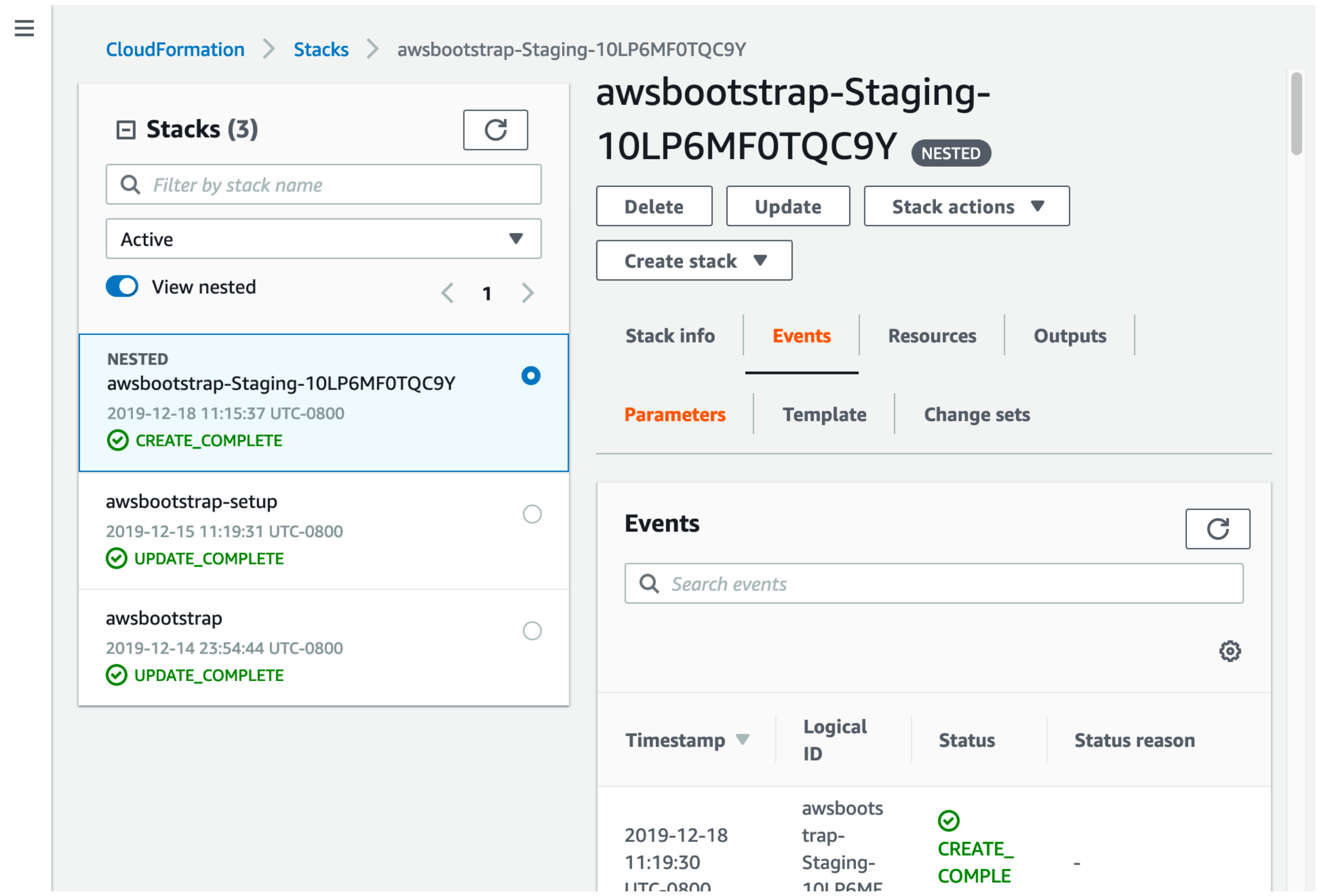
Now it’s time to deploy our changes.

```
./deploy-infra.sh


=========== Deploying setup.yml ===========

Waiting for changeset to be created..

No changes to deploy. Stack awsbootstrap-setup is up to date


=========== Packaging main.yml ===========


=========== Deploying main.yml ===========

Waiting for changeset to be created..
Waiting for stack create/update to complete
Successfully created/updated stack - awsbootstrap
[
    "http://awsbo-LoadB-1SN04P0UGU5RV-1429520787.us-east-1.elb.amazonaws.com:80"
]
```

terminal

Within a few minutes we should have all the resources recreated as they were, but organized under our new staging stack.

At this point, if we go to the CloudFormation console we should see three stacks.


CloudFormation > Stacks > awsbootstrap-Staging-10LP6MF0TQC9Y

Stacks (3)

Filter by stack name

Active

View nested 1

NESTED
awsbootstrap-Staging-10LP6MF0TQC9Y
2019-12-18 11:15:37 UTC-0800
CREATE_COMPLETE

awsbootstrap-setup
2019-12-15 11:19:31 UTC-0800
UPDATE_COMPLETE

awsbootstrap
2019-12-14 23:54:44 UTC-0800
UPDATE_COMPLETE

awsbootstrap-Staging-10LP6MF0TQC9Y NESTED

Delete | Update | Stack actions | Create stack

Stack info | Events | Resources | Outputs | Parameters | Template | Change sets

Events

Search events

Timestamp	Logical ID	Status	Status reason
2019-12-18 11:19:30 UTC-0800	awsbootstrap-Staging-10LP6MF	CREATE_COMPLE	-

New Staging nested stack

`awsbootstrap-setup`

A root stack containing our S3 buckets for CodePipeline and CloudFormation.

`awsbootstrap`

A root stack for our application containing our deployment resources and our staging nested stack.

`awsbootstrap-Staging-XYZ`

Our new nested staging stack containing all the application resources.

Let's verify that everything is still working.

```
for run in {1..20}; do curl -s http://awsbo-LoadB-1SN04P0UGU5RV-1429520787.us-east-1.elb.amazonaws
10 Hello World from ip-10-0-102-103.ec2.internal in awsbootstrap-Staging-10LP6MF0TQC9Y
10 Hello World from ip-10-0-61-182.ec2.internal in awsbootstrap-Staging-10LP6MF0TQC9Y
```

terminal

And now it's time to commit our changes to GitHub.

```
git add main.yml stage.yml setup.yml deploy-infra.sh
git commit -m "Split out staging nested stack"
git push
```

terminal

> **Note:** All the code has been already added and we are pushing it on our repository as well.

This code requires the following API keys to execute:

username	Not Specified...
AWS_ACCESS_KE...	Not Specified...
AWS_SECRET_AC...	Not Specified...
AWS_REGION	us-east-1
Github_Token	Not Specified...

```
{
  "name": "aws-bootstrap",
  "version": "1.0.0",
  "description": "",
  "main": "server.js",
  "scripts": {
    "start": "node ./node_modules/pm2/bin/pm2 start ./server.js --name hello_aws --log ../logs/app
    "stop": "node ./node_modules/pm2/bin/pm2 stop hello_aws",
    "build": "echo 'Building...'"
  },
  "dependencies": {
    "pm2": "^4.2.0"
  }
}
```

Now in the next lesson, we will add our prod stack.