Deployment Pipeline Template file. Part 1.

In the last lecture we defined the parameters needed for our deployment pipeline template, lets now build the core functionality.

This template file is longer than the other we defined before, so, we are going to have it defined in 2 lectures.

First our 4 main parts:

Description, Parameters, Resources and Outputs.

Lets type in Codepipeline Deployment Pipeline template. on the Description section

The the parameters, checking the main architecture file we have, one, two, three, four, five, six we have seven parameters and we will include one more for our template.

So we have in our template file:

CodeCommitRepo, RepositoryBranch, TargetGroup, StackName, Repository, Cluster, TemplateBucket, all this seven parameters will be String type.

(Wait 4 seconds)

We will add one more parameter called TestSemaphore, will be a String type, will have an empty default value, and we will add a description for readability and understanding. This description will say: This parameter is used during build stage, to set the variable for each test we need to build. Because this will be used as a variable during the building process.

Next the resource section, We want our codepipeline to be able to read, create a modify our main architecture, so we want:

1.- Our deploy stage needs to be able to modify our ECS, ECR, IAM, CodeCommit, Autoscaling and cloudwatch.

2.- Our build stage needs to be able to create and add log events, be able to access the ECR, and be able to get and put in the Artifact Bucket.

3.- Our Codepipeline needs to be able to get and put in the S3 Artifact bucket and in the Template bucket, and of course be able to access the services: codecommit, codebuild, cloudformation and the IAM to wrap everything together.

So we will need to assume 3 roles:

Lets start with the CloudFormationExecutionRole, this is the special one we are going to need in the Deploy stage, remember we want the Deploy stage to be able to modify and get info from the ECR, ECS, the autoscaling, the cloudwatch, etc.

this resource will be an AWS::IAM::Role type, also we want this role to be saved in case of stack deletion, so we use the DeletionPolicy: Retain.

Lets start now with the resource properties, we want our role to have a name, so we use the key RoleName and using the sub function we concatenate the cfn string with the stackname.

Next for the path user group we are going to use the slash.

Now we are going to declare the assume role action, as we used in other templates.

Lastly the policies for this resource, first we are going to add the PolicyName property, to give a name to the policy.

Then the policydocument, here we are going allow all actions to the services:

ECS, ECR, IAM, CODECOMMIT, APPLICATION AUTOSCALING and CLOUDWATCH.

Lets now declare the second role, the initial structure with the AssumeRolePolicyDocument is the same. Lets go directly to the Policies.

The first Actions we are going to allow are:
- logs:CreateLogGroup
    - logs:CreateLogStream
    - logs:PutLogEvents and
    - ecr:GetAuthorizationToken

Now lets specify the S3 resource artifactbucket action allowed:
- s3:GetObject
    - s3:PutObject
    - s3:GetObjectVersion

Finnaly our last resource is the ECR repository we will need access to:

- ecr:GetDownloadUrlForLayer
    - ecr:BatchGetImage
    - ecr:BatchCheckLayerAvailability
    - ecr:PutImage
    - ecr:InitiateLayerUpload
    - ecr:UploadLayerPart
    - ecr:CompleteLayerUpload

The role resource we are going to setup will be:
CodePipelineServiceRole the initial statements will be the same we have in the other two.

The resource we want to have access is the Artifact and Template bucket, for this one we will need to:

- s3:PutObject
    - s3:GetObject
    - s3:GetObjectVersion
    - s3:GetBucketVersioning

The next will be to give access to:

- codecommit:*
    - codebuild:StartBuild
    - codebuild:BatchGetBuilds
    - cloudformation:*
    - iam:PassRole

With this we finished the roles, lets start with our artifact bucket, When you create your first pipeline, AWS CodePipeline creates or use a single Amazon S3 bucket you specify in the same region as the pipeline. Codepipeline use it to store artifacts for your pipeline as the automated release process runs.

You are probably wondering whats an artifact: AWS CodePipeline copies files or changes that will be worked upon by the actions and stages in the pipeline to the Amazon S3 bucket. These objects are referred to as artifacts, and might be the source for an action (input artifacts) or the output of an action (output artifacts). An artifact can be worked upon by more than one action. Every action has a type. Depending on the type, the action might have an input artifact, which is the artifact it consumes or works on over the course of the action run; an output artifact, which is the output of the action; or both. Every output artifact must have a unique name within the pipeline. Every input artifact for an action must match the output artifact of an action earlier in the pipeline, whether that action is immediately prior to the action in a stage or runs in a stage several stages earlier.

Lets now define our ArtifactBucket:, this will be a AWS::S3::Bucket type and we want to have a Retain DeletionPolicy.

Because we are going to use the AWS Codebuild service in the Build stage of our codepipeline, we are goint to define a CodeBuildProject resource, lets call this one:

CodeBuildProject, the type will be AWS::CodeBuild::Project. Lets define the properties needed for this Codebuild project.

The first property will be the Artifacts, here we will need to specify a location for our S3 bucket, here we are going to reference our recently created Artifact bucket and will add a S3 type.

Then we will add the Source, as usual, the first thing to add is the location, here we will use sub function to concatenate the artifact bucket and the source.zip string, and will add a S3 type.

All Code build project needs a the Build specifications or directions, lets call it Buildspec, A build spec is a collection of build commands and related settings, in YAML format, that AWS CodeBuild uses to run a build. You can include a build spec as part of the source code or you can define a build spec when you create a build project.

The first thing to do is to add phases to our codebuild project specs, The first phase will be pre_build: Represents the commands that AWS CodeBuild will run before the build. For example, you might use this phase to log in to Amazon ECR, or you might install npm dependencies.

The first command we are going to execute is an echo command to add the code build id to a file in the temporal folder of the codebuild machine.

The next command will add the repository URL to the build tag file in the temporary folder.

Next we are going to add the tag and a json build file in the temporary folder.

Lastly we want to prints a command that we can use to log in to our default Amazon ECR registry.

Lets now define our build stage, Because we will need to build a docker image from our code commit repository we are going to use the docker build command: docker build --tag "$(cat /tmp/build_tag$(TestSemaphore).out)" --file Dockerfile$(TestSemaphore)

The last phase will be our post build command, here we are going to take the image built and push it to our ECR registry.

The last property we are going to specify in our build spec will be our artifacts. Represents information about where AWS CodeBuild can find the build output and how AWS CodeBuild will prepare it for uploading to the Amazon S3 output bucket.

Next we will define the environment for our Build Machine, we are going to use a General Type compute instance, the Docker 1.12 version, will use a Linux container and in the Environment Variable we will have the AWS default region and The repository url.

To finish with our Codebuild project we specify a name and of course the service role we define at the beginning for this Codebuild to work.

In the next video we are going to finish our template