Scaling: Add an Auto Scaling Group

We'll cover the following

Objective

- Replace explicit EC2 instances with Auto Scaling.

Steps

- Add an Auto Scaling Group.

Thus far, we have created our two EC2 instances explicitly. Doing this means that we need to update the CloudFormation template and do an infrastructure deployment just to add, remove, or replace an instance.

In this section, we’ll replace our two EC2 instances with an auto scaling group (ASG). We can then easily increase or decrease the number of hosts running our application by simply changing the value of desired instances for our ASG. In addition, we will configure our ASG to place our instances evenly across our two availability zones. This ensures that if one availability zone has an outage, our application would still have half of its capacity online.

The new ASG will also:

- Automatically replace an instance if the instance fails a health check.
- Attempt to respond to availability zone outages by temporarily adding additional instances to the remaining healthy zone.
- Automatically inform CodeDeploy about new instances so that we won’t have to trigger a deployment manually anymore when a new EC2 instance comes

online.

Multi-phase deployments

In a production system, we have to assume that we have users continually sending requests to the system all the time. As such, when we make infrastructure or software changes, it is important to do so in such a way that causes no disruption. We must consider the effect of every change and stage the changes so that the users do not experience any loss of service. We also need to be able to roll back a change if we discover that it's not doing quite what we wanted, again without affecting our users.

CloudFormation does a good job of doing this staging for us when it can determine the relationship between resources. But not all resources have obvious dependencies, and not all resources are strictly CloudFormation resources.

For example, in this section we'll be adding an ASG to manage our hosts and also remove the `Instance` and `Instance2` definitions that we had created for our two instances. If we were to make both of these changes simultaneously, we would likely have a gap in service between when the two hosts are terminated and when the new ASG hosts come online. Instead, we'll first add the new capacity via the ASGs, and then we will remove the old capacity only after we have checked that the new hosts are online.

The Amazon Builder's Library has a great article on making changes in deliberate phases: Ensuring rollback safety during deployments.

Adding our ASG

Adding an ASG to our CloudFormation template requires a relatively small amount of configuration. We need to reference our launch template, load balancer, and networking resources.

```
ScalingGroup:
  Type: AWS::AutoScaling::AutoScalingGroup
  UpdatePolicy:
    AutoScalingRollingUpdate:
      MinInstancesInService: "1"
      MaxBatchSize: "1"
      PauseTime: "PT15M"
      WaitOnResourceSignals: "true"
      SuspendProcesses:
        - HealthCheck
        - ReplaceUnhealthy
        - ReplaceUnhealthy
        - AZRebalance
        - AlarmNotification
        - ScheduledActions
  Properties:
    AutoScalingGroupName: !Sub 'ASG_${AWS::StackName}'
    AvailabilityZones:
      - !Select [ 0, !GetAZs '' ]
      - !Select [ 1, !GetAZs '' ]
    MinSize: 2
    MaxSize: 6
    HealthCheckGracePeriod: 0
    HealthCheckType: ELB
    LaunchTemplate:
      LaunchTemplateId: !Ref InstanceLaunchTemplate
      Version: !GetAtt InstanceLaunchTemplate.LatestVersionNumber
    TargetGroupARNs:
      - !Ref LoadBalancerTargetGroup
    MetricsCollection:
      -
        Granularity: "1Minute"
        Metrics:
          - "GroupMaxSize"
          - "GroupInServiceInstances"
    VPCZoneIdentifier:
      - !Ref SubnetAZ1
      - !Ref SubnetAZ2
    Tags:
      - Key: Name
        Value: !Ref AWS::StackName
        PropagateAtLaunch: "true"
```

main.yml

Line #3 and #8: The `WaitOnResourceSignal` works for ASGs in the same way that the `CreationPolicy` worked on individual instances. Our launch script will get the ASG's logical ID when querying its tag, and will pass that to the `cfn-signal` command, which in turn will signal to the ASG that the instance has launched successfully.

Line #20: We have two availability zones, so we'll set the minimum number of hosts to two to get one instance in each.

Line #23: Our ASG will use our load balancer's health check to assess the health of its instances.

Line #24: All instances that the ASG launches will be created as per our launch template.

Line #28: The ASG will add all launched instances to the load balancer's target group.

Line #35: The VPC and subnets into which the ASG will launch the instances.

Line #33: The VPC and subnets into which the ASG will launch the instances.

Line #41: Specifying `PropagateAtLaunch` ensures that this tag will be copied to all instances that are launched as part of this ASG.

> If we set our desired capacity via CloudFormation, then we should never change it manually in the AWS console. If we do change it manually, CloudFormation might fail the next deployment if it finds that the current desired capacity doesn't match what it was last set to.
>
> If you expect that you may need to make changes manually, then leave the desired capacity unset in CloudFormation.

Next, we will modify our deployment group to reference the new ASG. This will make the ASG tell CodeDeploy to deploy our application to every new instance that gets added to the ASG.

```yaml
  StagingDeploymentGroup:
    Type: AWS::CodeDeploy::DeploymentGroup
    Properties:
      DeploymentGroupName: staging
      AutoScalingGroups:
        - !Ref ScalingGroup
      ApplicationName: !Ref DeploymentApplication
      DeploymentConfigName: CodeDeployDefault.AllAtOnce
      ServiceRoleArn: !GetAtt DeploymentRole.Arn
      Ec2TagFilters:
        - Key: aws:cloudformation:stack-name
          Type: KEY_AND_VALUE
          Value: !Ref AWS::StackName
```

main.yml

Line #10: When the deployment group uses an ASG, we won't need `Ec2TagFilters` anymore. We will remove this property when we decommission our explicit `Instance` and `Instance2`.

Now it's time to run our `deploy-infra.sh` script to deploy our new ASG.

```
./deploy-infra.sh


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

Waiting for changeset to be created..
```

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



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

Waiting for changeset to be created..
Waiting for stack create/update to complete
Successfully created/updated stack - awsbootstrap
[
    "http://awsbo-LoadB-13F2DS4LKSCVO-10652175.us-east-1.elb.amazonaws.com:80"
]
```

terminal

Then, if we hit the load balancer endpoint, we should see our requests spread across our two explicit instances, as well as across two new instances that our ASG has spun up.

```
for run in {1..20}; do curl -s http://awsbo-LoadB-13F2DS4LKSCVO-10652175.us-east-1.elb.amazonaws.c
4 Hello World from ip-10-0-113-245.ec2.internal
5 Hello World from ip-10-0-50-202.ec2.internal
6 Hello World from ip-10-0-61-251.ec2.internal
5 Hello World from ip-10-0-68-58.ec2.internal
```

terminal

At this point, let's push our ASG changes to GitHub.

```
git add main.yml
git commit -m "Add ASG"
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...

```
const { hostname } = require('os');
const http = require('http');
const message = `Hello World from ${hostname()}\n`;
const port = 8080;

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end(message);
});
  server.listen(port, hostname, () => {
    console.log(`Server running at http://${hostname()}:${port}/`);
});
```

Now, we will remove our created instances in the next lesson.