HTTPS: Add an HTTPS Endpoint

We'll cover the following

Objective

- Migrate our endpoint from HTTP to HTTPS.

Steps

- Add an HTTPS endpoint.

Adding the HTTPS endpoint

We will now update our `deploy-infra.sh` script to retrieve the certificate ARN. This should go at the top of the script, and depends on the `DOMAIN` environment variable.

```
DOMAIN=the-good-parts.com
CERT=`aws acm list-certificates --region $REGION --profile awsbootstrap --output text \
        --query "CertificateSummaryList[?DomainName=='$DOMAIN'].CertificateArn | [0]"`
```

deploy-infra.sh

Line #3: Newly added environment variable holding our certificate.

We then have to pass the certificate ARN as a parameter to `main.yml`.

```
# 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 \

    Domain=$DOMAIN \
    Certificate=$CERT \
    GitHubOwner=$GH_OWNER \
    GitHubRepo=$GH_REPO \
    GitHubBranch=$GH_BRANCH \
    GitHubPersonalAccessToken=$GH_ACCESS_TOKEN \
    CodePipelineBucket=$CODEPIPELINE_BUCKET
```

.deploy-infra.sh

Line #13: The certificate ARN.

We also have to add this as a parameter in the `main.yml` template.

```
Certificate:
  Type: String
  Description: 'An existing ACM certificate ARN for your domain'
```

main.yml

Then, we also have to pass the ARN to our nested stacks by adding a parameter to the `Staging` and `Prod` resources in `main.yml`.

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

Prod:
  Type: AWS::CloudFormation::Stack
  Properties:
    TemplateURL: stage.yml
    TimeoutInMinutes: 30
    Parameters:
      EC2InstanceType: !Ref EC2InstanceType
      EC2AMI: !Ref EC2AMI
      Domain: !Ref Domain
      SubDomain: prod
      Certificate: !Ref Certificate
```

main.yml

Line #11 and #23: The certificate ARN.

Finally, we have to add an input parameter in `stage.yml` to receive the certificate ARN from `main.yml`.

```
Certificate:
  Type: String
  Description: 'An existing ACM certificate ARN for subdomain.domain'
```

stage.yml

Next, we're going to modify our security group to allow traffic on HTTPS ports 443 and 8443.

```
SecurityGroup:
  Type: AWS::EC2::SecurityGroup
  Properties:
    VpcId: !Ref VPC
    GroupDescription:
      !Sub 'Internal Security group for ${AWS::StackName}'
    SecurityGroupIngress:
      - IpProtocol: tcp
        FromPort: 8080
        ToPort: 8080
        CidrIp: 0.0.0.0/0
      - IpProtocol: tcp
        FromPort: 8443
        ToPort: 8443
        CidrIp: 0.0.0.0/0
      - IpProtocol: tcp
        FromPort: 80
        ToPort: 80
        CidrIp: 0.0.0.0/0
      - IpProtocol: tcp
        FromPort: 443
        ToPort: 443
        CidrIp: 0.0.0.0/0
      - IpProtocol: tcp
        FromPort: 22
        ToPort: 22
        CidrIp: 0.0.0.0/0
    Tags:
      - Key: Name
        Value: !Ref AWS::StackName
```

stage.yml

Line #12 and #20: Newly added HTTPS ports.

At this point, we need to modify the `UserData` section of our EC2 launch template to make the instance generate a self-signed certificate automatically when it starts up. This certificate will be used for traffic between the load balancer and the instance.

```
cat > /tmp/install_script.sh << EOF
```

```
  # START
  echo "Setting up NodeJS Environment"
  curl https://raw.githubusercontent.com/nvm-sh/nvm/v0.34.0/install.sh | bash


  # Dot source the files to ensure that variables are available within the current shell
  . /home/ec2-user/.nvm/nvm.sh
  . /home/ec2-user/.bashrc

  # Install NVM, NPM, Node.JS
  nvm alias default v12.7.0
  nvm install v12.7.0
  nvm use v12.7.0

  # Create log directory
  mkdir -p /home/ec2-user/app/logs

  # Create a self-signed TLS certificate to communicate with the load balancer
  mkdir -p /home/ec2-user/app/keys
  cd /home/ec2-user/app/keys
  openssl req -new -newkey rsa:4096 -days 365 -nodes -x509 \
              -subj "/C=/ST=/L=/O=/CN=localhost" -keyout key.pem -out cert.pem
EOF
```

stage.yml

Line #21: Generates a certificate (`cert.pem`) and private key (`key.pem`) and puts them in `/home/ec-user/app/keys`.

Next, we add a new target group so that the load balancer forwards traffic to the application's 8443 port.

```
HTTPSLoadBalancerTargetGroup:
  Type: AWS::ElasticLoadBalancingV2::TargetGroup
  Properties:
    TargetType: instance
    Port: 8443
    Protocol: HTTPS
    VpcId: !Ref VPC
    HealthCheckEnabled: true
    HealthCheckProtocol: HTTPS
    Tags:
      - Key: Name
        Value: !Ref AWS::StackName
```

stage.yml

Line #5: 8443 is the non-privileged port that our application will use to serve HTTPS requests.

Line #9: The health check will also be made on the HTTPS port.

Now, let's add a new load balancer listener for HTTPS.

```
HTTPSLoadBalancerListener:
  Type: AWS::ElasticLoadBalancingV2::Listener
  Properties:

    DefaultActions:
      - Type: forward
        TargetGroupArn: !Ref HTTPSLoadBalancerTargetGroup
    LoadBalancerArn: !Ref LoadBalancer
    Certificates:
      - CertificateArn: !Ref Certificate
    Port: 443
    Protocol: HTTPS
```

stage.yml

Line #9: The certificate ARN.

Line #10: 443 is the standard HTTPS port.

Then we need to add the new HTTPS target group to the `ScalingGroup` ASG so that the instances managed by the ASG will be added automatically behind the load balancer's HTTPS target.

```
TargetGroupARNs:
  - !Ref LoadBalancerTargetGroup
  - !Ref HTTPSLoadBalancerTargetGroup
```

stage.yml

Line #3: References the new HTTPS target group.

Next, we will also add a new entry to the `Outputs` section in `stage.yml` to return the URL for our new HTTPS endpoint.

```
HTTPSEndpoint:
  Description: The DNS name for the stage
  Value: !Sub "https://${DNS}"
```

stage.yml

Finally, we'll add two new outputs from `main.yml` for the new HTTPS endpoints.

```
Outputs:
  StagingLBEndpoint:
    Description: The DNS name for the staging LB
    Value: !GetAtt Staging.Outputs.LBEndpoint
    Export:
      Name: StagingLBEndpoint
  StagingHTTPSLBEndpoint:
    Description: The DNS name for the staging HTTPS LB
    Value: !GetAtt Staging.Outputs.HTTPSEndpoint
    Export:
      Name: StagingHTTPSLBEndpoint
  ProdLBEndpoint:
    Description: The DNS name for the prod LB
    Value: !GetAtt Prod.Outputs.LBEndpoint
    Export:
      Name: ProdLBEndpoint
  ProdHTTPSLBEndpoint:
    Description: The DNS name for the prod HTTPS LB
    Value: !GetAtt Prod.Outputs.HTTPSEndpoint
    Export:
      Name: ProdHTTPSLBEndpoint
```

main.yml

Line #17: Newly added HTTPS endpoints.

It's time to deploy our changes. This change may take longer than previous updates, because it has to spin up two new instances per stage with the updated launch script, and then terminate the old ones.

```
./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://prod.the-good-parts.com",
    "https://prod.the-good-parts.com",
    "http://staging.the-good-parts.com",
    "https://staging.the-good-parts.com"
]
```

terminal

Our HTTP endpoints should continue to respond correctly. However, if we try to reach the new HTTPS endpoints, we'll get an error, because the load balancer can't yet reach our application on port 8443.

```
curl https://prod.the-good-parts.com
<html>
<head><title>502 Bad Gateway</title></head>
```

```
<body bgcolor="white">
<center><h1>502 Bad Gateway</h1></center>


</body>
</html>
```

terminal

If you were to look for the new HTTPS target group in the AWS console, you should see no healthy hosts in the *Monitoring* tab. You can also see that the EC2 instances are being continuously created and destroyed.

This is happening because we haven't yet updated our application to serve HTTPS requests on port 8443, so our instances are failing their health checks. In the real world, it would have been better to update the application first, and only then update the infrastructure. But here, we wanted to do it in the reverse order to demonstrate the behavior of the load balancer health checks. So, let's push our infrastructure changes to GitHub, and then let's fix our application.

```
git add deploy-infra.sh main.yml stage.yml
git commit -m "Add HTTPS listener; Add cert to launch script"
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"

  }
}
```

In the next lesson, we will make our application speak HTTPS.