# Starting from Scratch: Basic Web Application

**We'll cover the following**

* [Objective](https://www.educative.io/module/lesson/the-good-parts-of-aws/B6xBpkNwkmQ#Objective)
* [Steps](https://www.educative.io/module/lesson/the-good-parts-of-aws/B6xBpkNwkmQ#Steps)
* [Creating our application](https://www.educative.io/module/lesson/the-good-parts-of-aws/B6xBpkNwkmQ#Creating-our-application)
* [Pushing our code to GitHub](https://www.educative.io/module/lesson/the-good-parts-of-aws/B6xBpkNwkmQ#Pushing-our-code-to-GitHub)

In this part, we will walk you through getting a basic web application running in the cloud on AWS. We will start with a blank project, and build the application and its infrastructure step by step. Each step focuses on a single aspect of the infrastructure, and we will try to explain in detail what’s happening, and why.

All the source code shown in this guide is available on GitHub. Each commit represents a code checkpoint from the book, and you can find it all here: [AWS Bootstrap](https://github.com/good-parts/aws-bootstrap).

When interacting with AWS, we will use both the [AWS console](https://console.aws.amazon.com/) and the [AWS CLI](https://aws.amazon.com/cli). In addition, we will also make use of the following tools:

• [GitHub](https://github.com/) as the source code repository for our application and infrastructure code.

• [node.js](https://nodejs.org/) and [npm](https://www.npmjs.com/" \t "_blank) to build our application.

• [git](https://git-scm.com/) for version control.

• [curl](https://curl.haxx.se/docs/manual.html) to interact with our application.

## Objective

* Get a simple web application running on a single EC2 instance.

## Steps

* Write a basic “hello world” web application.

In this section, we will create a tiny web application and we will get it running on an EC2 instance in an AWS account. We will start by performing all the steps manually, and we will automate them in later sections. Our application is not very interesting, and the way it will be hosted is far from ideal, but it will allow us to spend the rest of this book building a well-contained AWS setup, step by step.

## Creating our application

We will need git and npm installed. But don’t worry, these are already installed on our platform. So you can straight away use them in our terminal widget.

Now, let’s create our bare-bones application, along with a git repository to store it.

1

2

3

mkdir·aws-bootstrap·&&·cd·aws-bootstrap

git·init

npm·init·-y





terminal

Our application is going to listen for HTTP requests on port 8080 and respond with “Hello World”. The entire application will be in one file, server.js.

1

2

3

4

5

6

7

8

9

10

11

12

13

const·{·hostname·}·=·require('os');

const·http·=·require('http');

const·message·=·'Hello·World\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}/`);

});





server.js

⚠️ **Line #4:** we’ll run the server on port 8080 because port numbers below 1024 require root privileges.

We can run our application directly with the node command.

1

2

node server.js

Server running at http://localhost:8080/





terminal

And we can test it with curl from another terminal window.

1

2

curl localhost:8080

Hello World





terminal

All these commands are already executed in the below widget. Hit on the RUN button to start the server.

###### /

server.js

**server.js**

1

2

3

4

5

6

7

8

9

10

11

12

const { hostname } = require('os');

const http = require('http');

const message = 'Hello World\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}/`);

});





Run

Save

Reset

**Your app can be found at:**[https://ed-6069882977779712.educative.run](https://ed-6069882977779712.educative.run/)

Next, let’s use a process manager to monitor our application so that it automatically restarts if it crashes. To do this, we need to modify our package.json file.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

{

  "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.log ",

  "stop": "node ./node\_modules/pm2/bin/pm2 stop hello\_aws",

  "build": "echo 'Building...'"  },

  "dependencies": {

    "pm2": "^4.2.0"

  }

}





package.json

**Line #11:** Takes a dependency on pm2, a node process manager.

**Line #7:** From now on, we’ll use npm start to start our application via pm2.

**Line #7:** pm2 will monitor it under the name hello\_aws and send its stdout to ../logs/app.log

**Line #8:** We’ll use npm stop to tell pm2 to stop our application.

**Line #9:** A dummy build step. Your actual production build process goes here.

Now, we need to use npm to get the new dependency we added in package.json.

1

npm install





terminal

In package.json we specified that the application’s logs are sent to ../logs. Having the directory for logs outside the application’s directory will be important when we deploy this with CodeDeploy, since it prevents the logs directory from being deleted with every deployment. For now, let’s create the directory manually on our local machine.

1

mkdir ../logs





terminal

And now we should be able to start our application through the process manager.

1

2

3

4

5

6

7

8

9

npm start

[PM2] Applying action restartProcessId on app [hello\_aws](ids: [ 0 ])

[PM2] [hello\_aws](0) ✓

[PM2] Process successfully started

┌────┬────────────┬───────┬──────┬───────────┬──────────┬──────────┐

│ id │ name       │ mode  │ ↺    │ status    │ cpu      │ memory   │

├────┼────────────┼───────┼──────┼───────────┼──────────┼──────────┤

│ 0  │ hello\_aws  │ fork  │ 0    │ online    │ 0%       │ 8.7mb    │





terminal

Testing via curl should once again show our “Hello World” message.

1

2

curl localhost:8080

Hello World





terminal

Again, all these commands are already executed in the widget below. Hit on the RUN button to start the server.

###### /

server.js

package.json

**server.js**

1

2

3

4

5

6

7

8

9

10

11

12

const { hostname } = require('os');

const http = require('http');

const message = 'Hello World\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}/`);

});





Run

Save

Reset

**Your app can be found at:**[https://ed-6069882977779712.educative.run](https://ed-6069882977779712.educative.run/)

And with everything working, we can now commit all our changes to git.

1

2

git add server.js package.json package-lock.json

git commit -m "Create basic hello world web application"





terminal

## Pushing our code to GitHub

If you don’t already have a GitHub account, [create one](https://github.com/) (it’s free for simple projects like this).

Next, create a [new GitHub repository](https://github.com/new). We named ours aws-bootstrap. The repository needs to be public for now, but we’ll be able to make it private once we set up CodeBuild.

Now that we have a GitHub repository, let’s push our local changes to GitHub.

1

2

git remote add origin https://github.com/<username>/aws-bootstrap.git

git push -u origin master





terminal

**Line #1:** replace <username> with your GitHub username.

Again all these git commands are executed. You just need to set your Github username as the environment variable first.

**NOTE:** When you run the code, it will ask you your username and password before pushing the files to Github. **So don’t miss that!**

This code requires the following environment variables to execute:

username

Not Specified...

Github\_Token

Not Specified...

Edit

###### /

server.js

package.json

**server.js**

1

2

3

4

5

6

7

8

9

10

11

12

const { hostname } = require('os');

const http = require('http');

const message = 'Hello World\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}/`);

});





Run

Save

Reset

**Your app can be found at:**[https://ed-6069882977779712.educative.run](https://ed-6069882977779712.educative.run/)

In the next lesson, we will host our application and manually install it on an EC2 instance.

**Starting from Scratch: Manual AWS Infrastructure**

Creation of AWS infrastructure manually to host our application is performed in this lesson. We then will install our web application on an EC2 instance.

**We'll cover the following**

* [Objective](https://www.educative.io/module/lesson/the-good-parts-of-aws/B1E7D0q6pyN#Objective)
* [Steps](https://www.educative.io/module/lesson/the-good-parts-of-aws/B1E7D0q6pyN#Steps)
* [Hosting our application](https://www.educative.io/module/lesson/the-good-parts-of-aws/B1E7D0q6pyN#Hosting-our-application)
* [Running our application](https://www.educative.io/module/lesson/the-good-parts-of-aws/B1E7D0q6pyN#Running-our-application)

**Objective**

* Get a simple web application running on a single EC2 instance.

**Steps**

* Manually create basic AWS infrastructure to host our application.
* Manually install our application on an EC2 instance.

**Hosting our application**

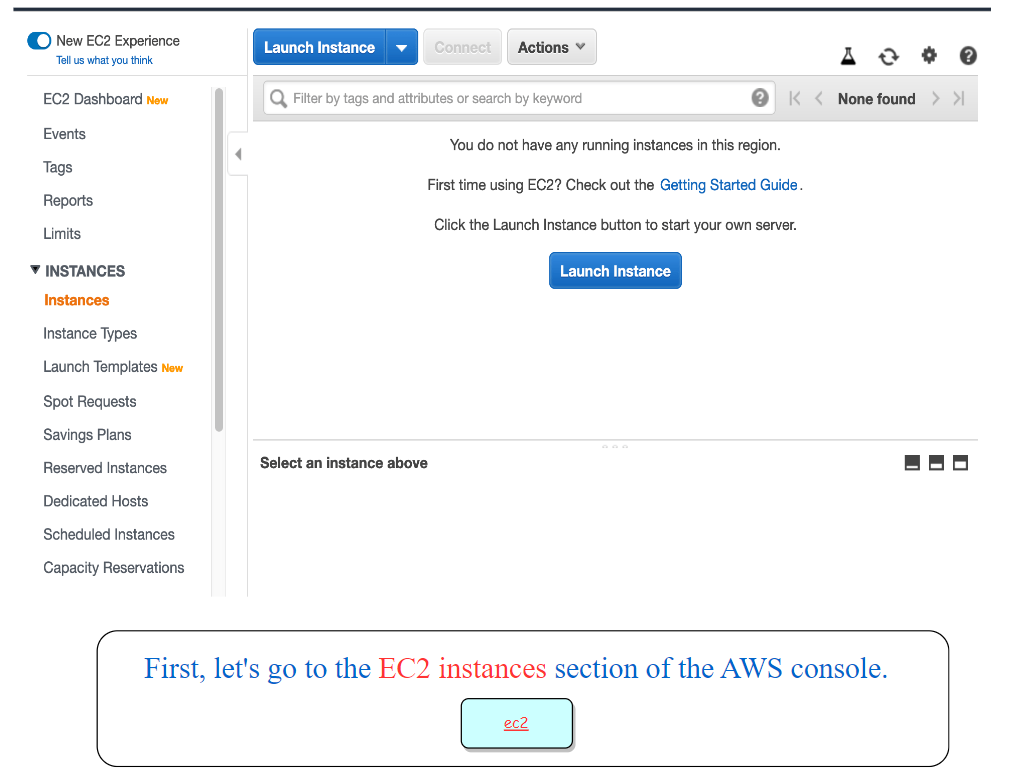
If you don’t already have an AWS account, [you will need to create one](https://portal.aws.amazon.com/billing/signup).

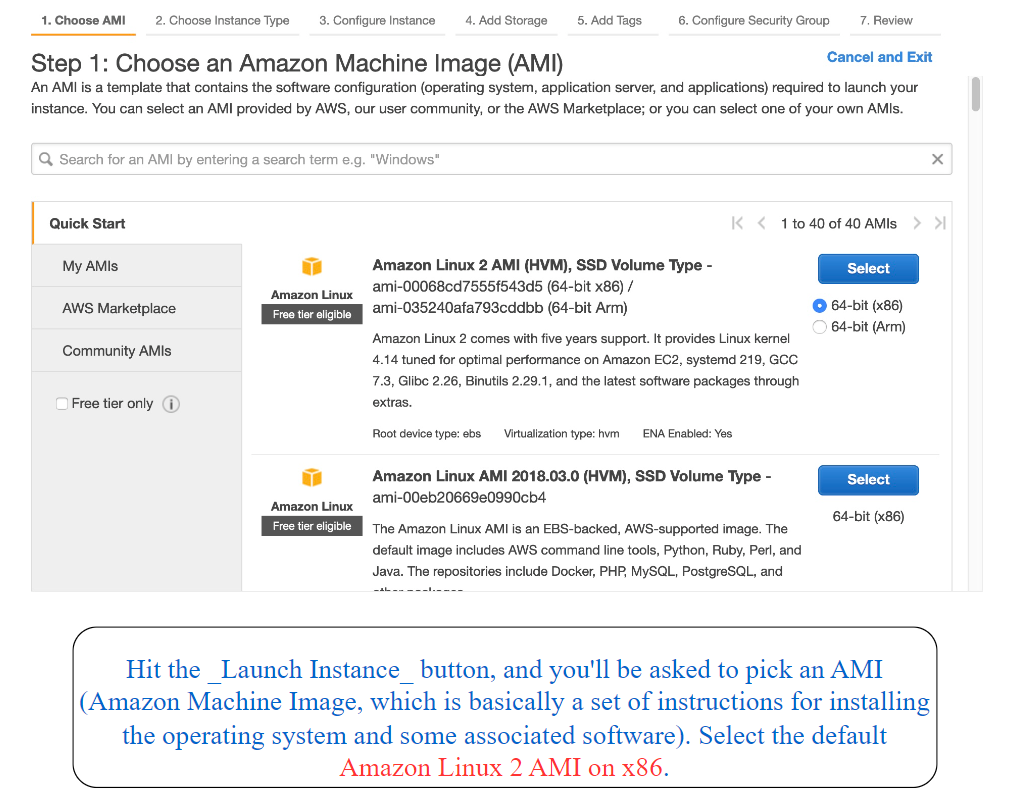
⚠️You should never use your AWS root account credentials other than to create an Administrator user for your account. AWS has [several best practices](https://docs.aws.amazon.com/IAM/latest/UserGuide/best-practices.html) for managing accounts and credentials.

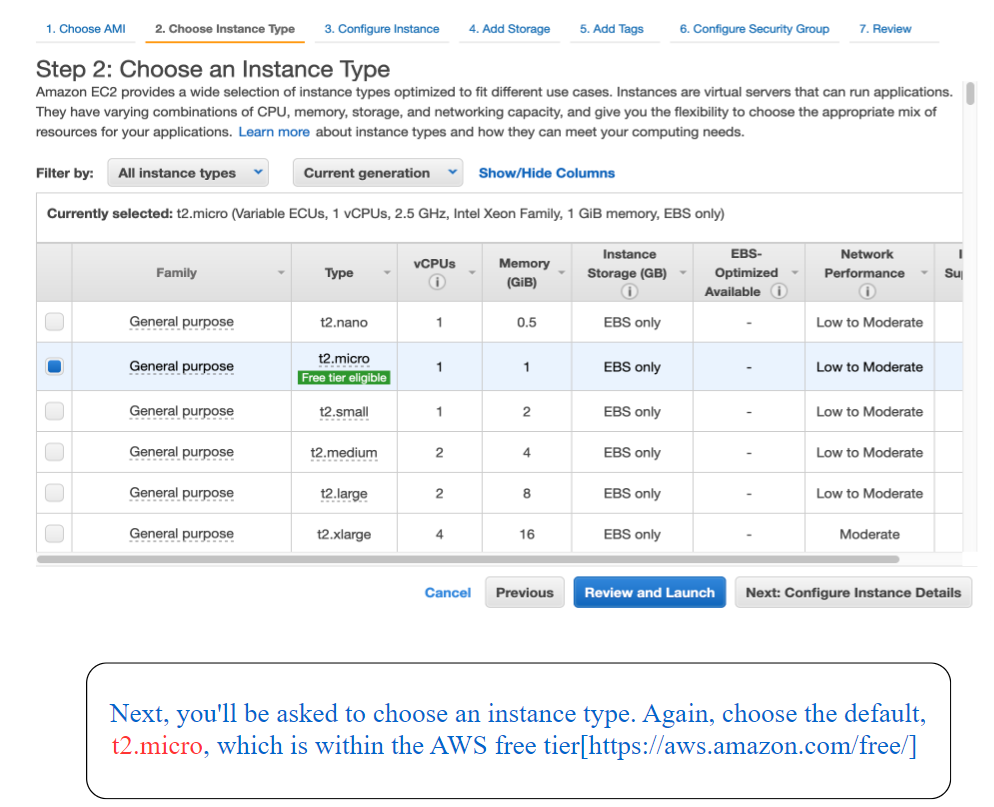
If you are creating a new AWS account or you don’t yet have an Administrator user, use the root account to [create an Administrator user](https://docs.aws.amazon.com/IAM/latest/UserGuide/getting-started_create-admin-group.html) now, and then use only that user.

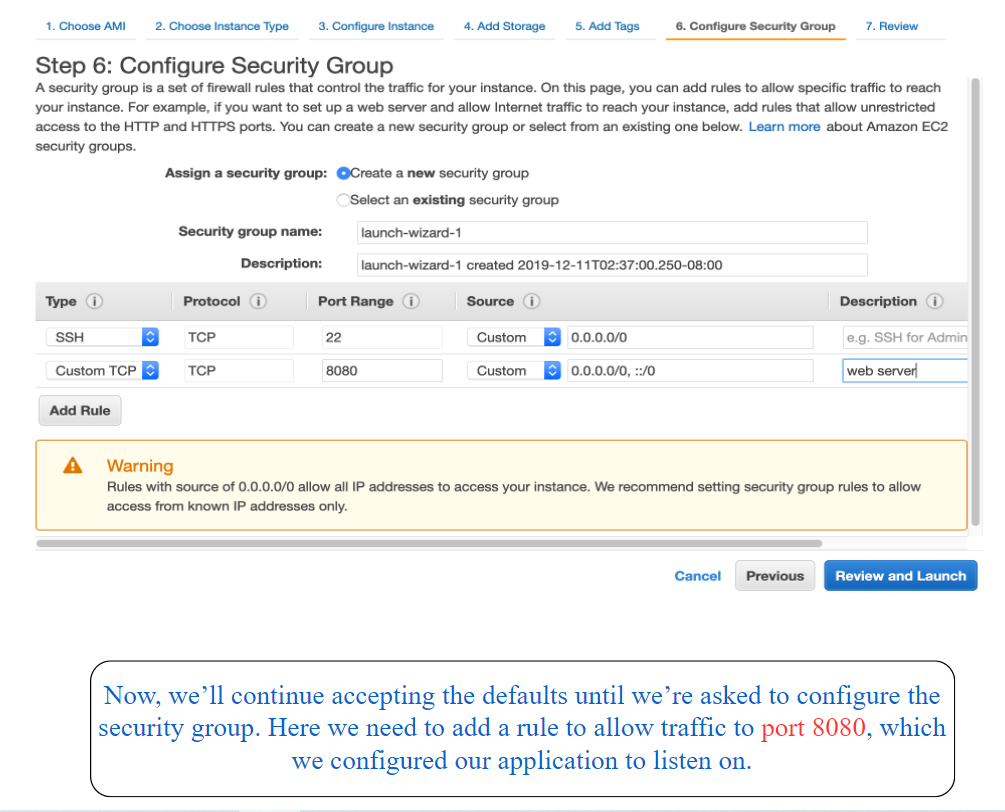
🔍 We’ve chosen to provide links to the AWS console in the US East (N. Virginia) (us-east-1) region. Feel free to choose a region closer to your physical location. Everything described here is available in all AWS regions.

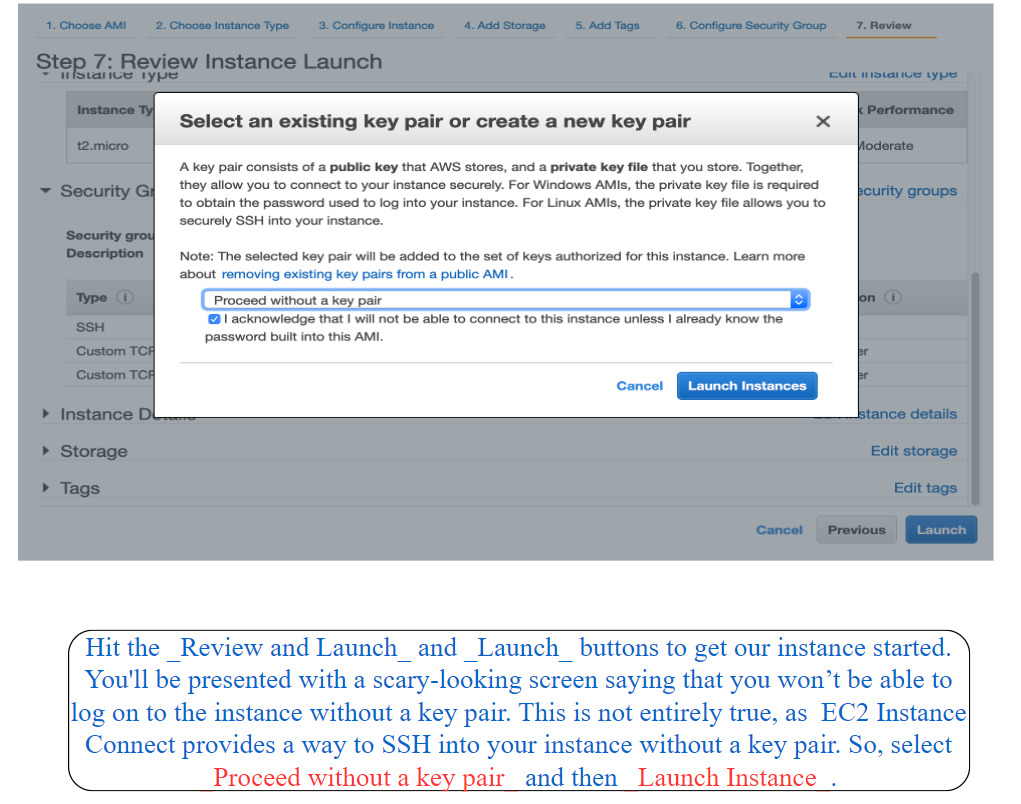
Now we’re going to use the AWS console to create the minimal resources necessary to get our application running in our AWS account.

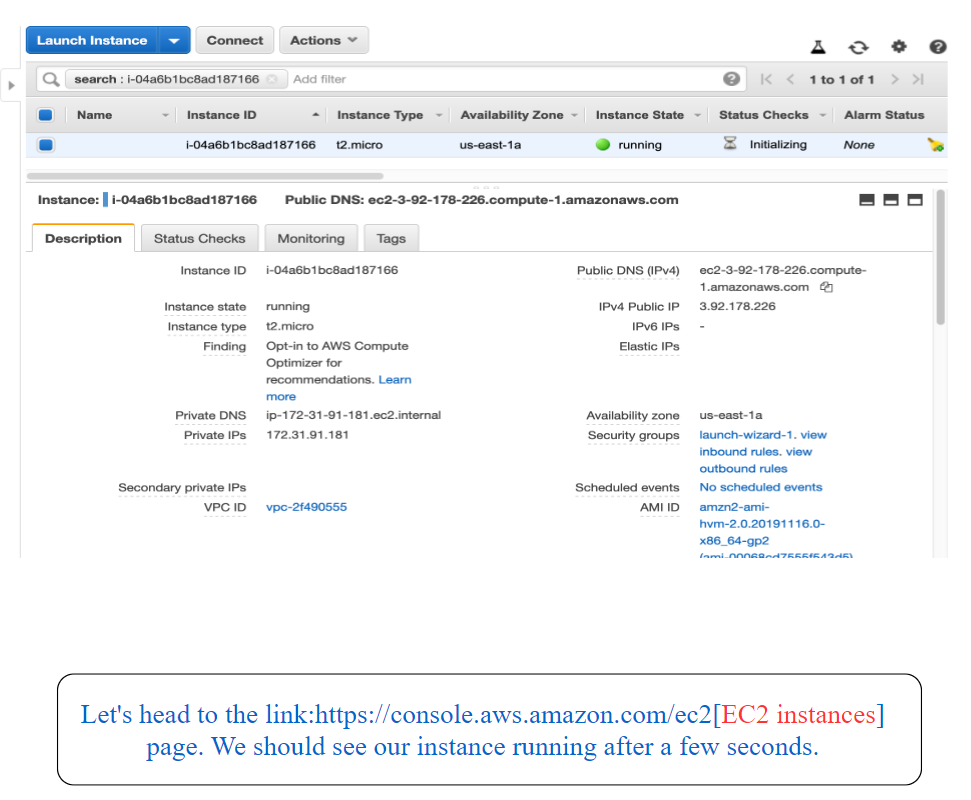








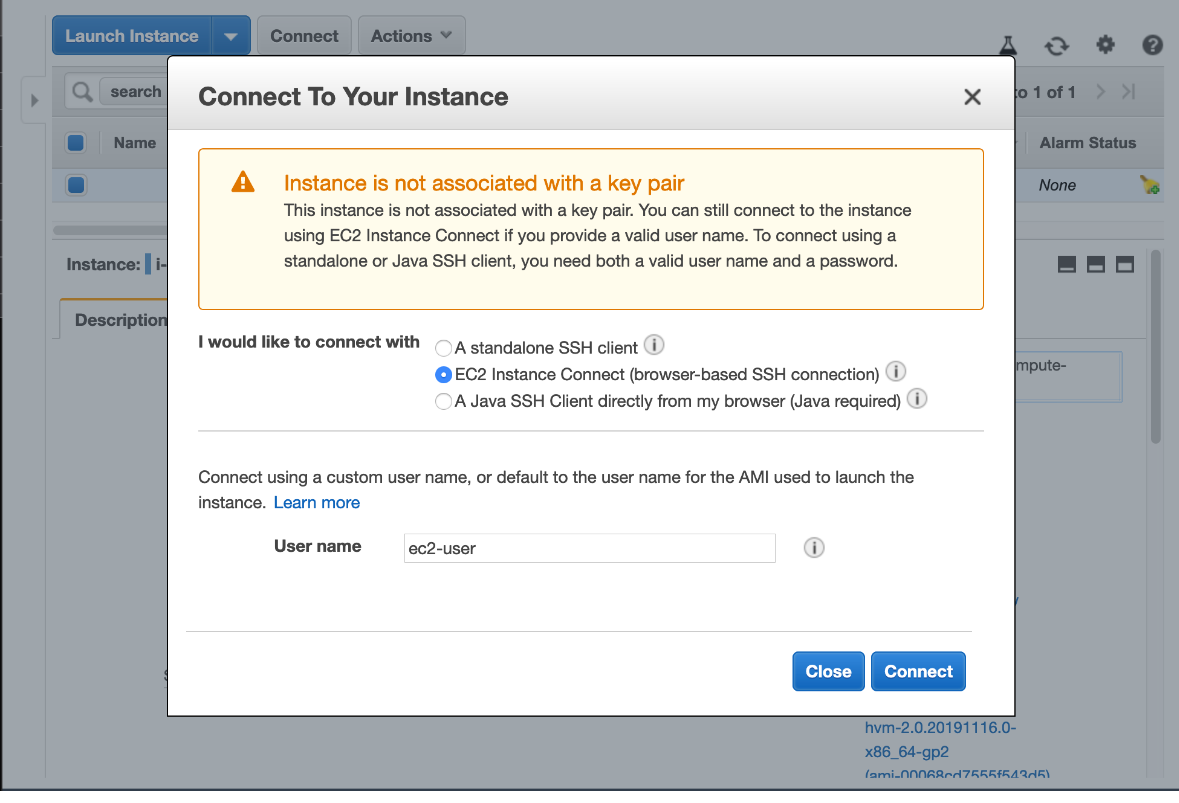




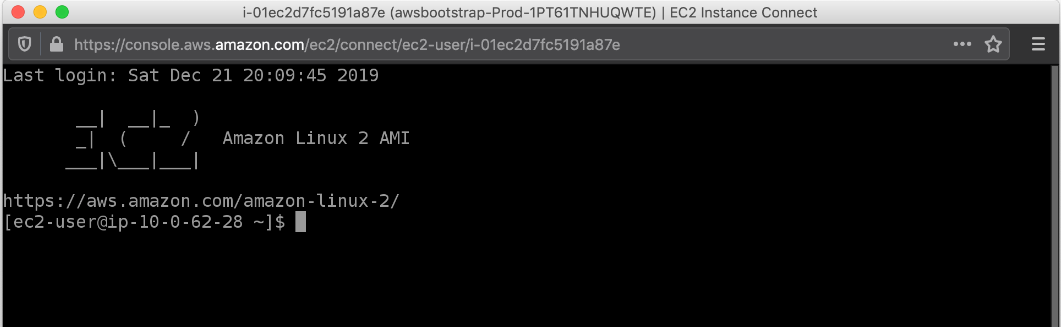
Note the Public DNS (IPv4) field shown in the console. This address will allow us to reach our instance via the internet.

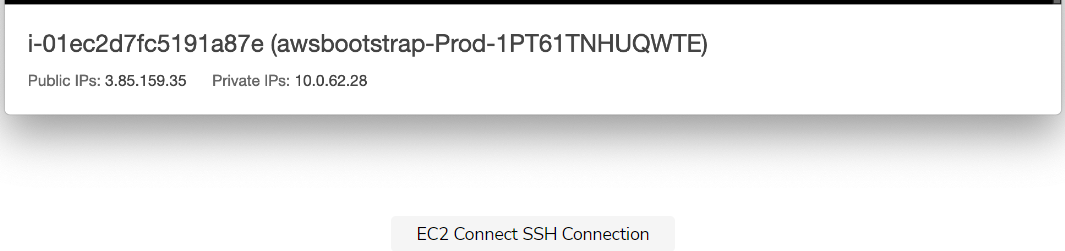
## Running our application

For this exercise, we’re going to SSH to our instance from the browser. In the EC2 instances view, select our instance and then press the Connect button at the top of the page to start the connection. Then select EC2 Instance Connect and hit Connect at the bottom of the dialogue.



A new browser window will pop up with an open SSH connection to our instance.





Now that we have an SSH shell open on the host, we’re going to update the installed packages manually (for now). Then we will install our application and its dependencies.

1

2

3

4

sudo yum -y update

curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.34.0/install.sh | bash

. ~/.nvm/nvm.sh

nvm install node





terminal

**Line #1:** Updates the installed yum packages.

**Line #2:** We’ll use [NVM](link:https://github.com/nvm-sh/nvm/blob/master/README.md) to install node.

**Line #3:** Makes sure NVM is available.

**Line #4:** Installs node via NVM.

Now that our dependencies are in place, we can proceed to install our application.

1

2

3

4

5

6

7

8

9

mkdir logs

curl -sL https://github.com/<username>/aws-bootstrap/archive/master.zip --output master.zip

unzip master.zip

mv aws-bootstrap-master app

cd app

npm install

npm start

curl localhost:8080

Hello World





terminal

**Line #2:** Replace <username> with your GitHub user name.

**Line #2:** Here we use GitHub’s archive to download the latest version of our application. This works only with public repositories.

At this point, if you copy the Public DNS (IPv4) from *Figure 6* (See the *sixth* slide in the above slideshow), you should be able to point your web browser to http://<public\_dns>:8080 and see the “Hello World” message from our application.

Congratulations! We now have our application running in the cloud. However, our infrastructure is not yet fault-tolerant or scalable. And it’s not particularly easy to configure again, either. But we’ll get there, step by step, in the following sections.

📝 At this point, you may also want to set up [billing alerts with CloudWatch](https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/monitor_estimated_charges_with_cloudwatch.html) to help you monitor your AWS charges and to remind you if you forget to decommission any experiments you have performed using your AWS account.

In the next lesson, we will recreate our infrastructure using CloudFormation.

**Infrastructure as Code: Create CloudFormation Stack**

CloudFormation will be used in this lesson to recreate the same instrastructure we set up in the previous lessons. We are going to automate the process of instrastructure creation, instead of setting everything up manually through the AWS console.

**We'll cover the following**

* [Objective](https://www.educative.io/module/lesson/the-good-parts-of-aws/YV8n13wo0lA#Objective)
* [Steps](https://www.educative.io/module/lesson/the-good-parts-of-aws/YV8n13wo0lA#Steps)
* [Configuring the AWS CLI](https://www.educative.io/module/lesson/the-good-parts-of-aws/YV8n13wo0lA#Configuring-the-AWS-CLI)
* [Infrastructure as code](https://www.educative.io/module/lesson/the-good-parts-of-aws/YV8n13wo0lA#Infrastructure-as-code)
  + [Parameters](https://www.educative.io/module/lesson/the-good-parts-of-aws/YV8n13wo0lA#Parameters)
  + [Resources](https://www.educative.io/module/lesson/the-good-parts-of-aws/YV8n13wo0lA#Resources)
  + [Outputs](https://www.educative.io/module/lesson/the-good-parts-of-aws/YV8n13wo0lA#Outputs)

**Objective**

* Recreate our infrastructure using CloudFormation.

**Steps**

* Configure the AWS CLI.
* Create a CloudFormation Stack.

In this section, we’ll recreate the same infrastructure we set up in the previous section, but this time, we’ll use CloudFormation to automate the process, instead of setting everything up manually through the AWS console.

**Configuring the AWS CLI**

We’re going to use the AWS CLI to access AWS resources from the command line rather than the AWS console. If you don’t already have it installed, follow [the official directions for your system](https://docs.aws.amazon.com/cli/latest/userguide/cli-chap-install.html). Then, configure a profile named awsbootstrap using a newly generated Access Key ID and Secret Access Key, as described in [the AWS documentation](https://docs.aws.amazon.com/cli/latest/userguide/cli-chap-configure.html).

aws configure --profile awsbootstrap

AWS Access Key ID [None]: AKIAIOSFODNN7EXAMPLE

AWS Secret Access Key [None]: wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY

Default region name [None]: us-east-1

Default output format [None]: json

terminal

We can test our AWS CLI configuration by listing the EC2 instances in our account.

aws ec2 describe-instances --profile awsbootstrap

terminal

Now that we have the AWS CLI working, we could write a bunch of scripts that call the various AWS API commands to automate the process for setting up our infrastructure. But that would be very brittle and complicated. Luckily, there’s a better way.

## Infrastructure as code

Infrastructure as code is the idea of using the same processes and tools to update your infrastructure as you do for your application code. We will now start defining our infrastructure into files that can be linted, schema-checked, version controlled, and deployed without manual processes. Within AWS, the tool for this is CloudFormation.

We’ll use the AWS CLI to submit infrastructure updates to CloudFormation. Although we could interact with CloudFormation directly from the AWS CLI, it is easier to write a script containing the necessary parameters. We’ll call the script deploy-infra.sh and use it to deploy changes to our CloudFormation stack. A stack is what CloudFormation calls the collection of resources that are managed together as a unit.

#!/bin/bash

STACK\_NAME=awsbootstrap

REGION=us-east-1

CLI\_PROFILE=awsbootstrap

EC2\_INSTANCE\_TYPE=t2.micro

# 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 main.yml \

--no-fail-on-empty-changeset \

--capabilities CAPABILITY\_NAMED\_IAM \

--parameter-overrides \

EC2InstanceType=$EC2\_INSTANCE\_TYPE

deploy-infra.sh

**Line #3:** The stack name is the name that CloudFormation will use to refer to the group of resources it will manage.

**Line #4:** The region to deploy to.

**Line #5:** We use the awsbootstrap profile that we created in the previous section.

**Line #7:** An instance type in the free tier.

**Line #15:** The main.yml file is the CloudFormation template that we will use to define our infrastructure.

**Line #18:** These correspond to the input parameters in the template that we’ll write next.

Before we move on, let’s make our helper script executable.

chmod +x deploy-infra.sh

terminal

Now it’s time to start creating our CloudFormation template. It uses the following three top-level sections.

### Parameters

These are the input parameters for the template. They give us the flexibility to change some settings without having to modify the template code.

### Resources

This is the bulk of the template. Here is where we define and configure the resources that CloudFormation will manage for us.

### Outputs

These are like return values for the template. We use them to make it easy to find some of the resources that CloudFormation will create for us.

We’re going to name our template file main.yml. There will be other template files later, but they will all be referenced from here. This file will become quite large, so let’s start by sketching out its high-level structure.

AWSTemplateFormatVersion: 2010-09-09

Parameters:

Resources:

Outputs:

main.yml

Next, let’s fill in the input parameters to accept the instance type. The parameter names we’ll use in the template will need to match the parameters we used in the deploy-infra.sh script. For now, we have EC2InstanceType. We will add other parameters throughout the book. EC2AMI is a bit special. We use the AWS SSM provided value that specifies the [most up-to-date AMI](https://aws.amazon.com/blogs/compute/query-for-the-latest-amazon-linux-ami-ids-using-aws-systems-manager-parameter-store/).

Parameters:

EC2InstanceType:

Type: String

EC2AMI:

Type: 'AWS::SSM::Parameter::Value<AWS::EC2::Image::Id>'

Default: '/aws/service/ami-amazon-linux-latest/amzn2-ami-hvm-x86\_64-gp2'

main.yml

**On line #5:** This is a special parameter type that allows our template to get the latest AMI without having to specify the exact version.

The first resource that we’re going to define is our security group. This functions like a firewall for the EC2 instance that we’ll create. We need to add a rule to allow TCP traffic to port 8080 (to reach our application) and to port 22 (for SSH access).

Resources:

SecurityGroup

Type: AWS::EC2::SecurityGroup

Properties:

GroupDescription: !Sub 'Internal Security group for ${AWS::StackName}'

SecurityGroupIngress:

- IpProtocol: tcp

FromPort: 8080

ToPort: 8080

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

main.yml

**Line #6:** !Sub is a [CloudFormation function](https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/intrinsic-function-reference-sub.html) that performs string interpolation. Here, we interpolate the stack name into the security group description.

**Line #6:** AWS::StackName is a [CloudFormation pseudo parameter](https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/pseudo-parameter-reference.html). There are [many other useful ones](https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/pseudo-parameter-reference.html).

**Line #16:** Tags are great. There are [many ways to use them](https://aws.amazon.com/answers/account-management/aws-tagging-strategies/). At the very least, it makes sense to tag most resources with the stack name if you are going to have multiple stacks in the same AWS account.

**Line #18:** !Ref is a [CloudFormation function](https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/intrinsic-function-reference-ref.html) for referring to other resources in your stack.

The next resource we’ll create is an IAM role, which our EC2 instance will use to define its permissions. At this point, our application doesn’t need much, as it isn’t using any AWS services yet. For now, we will grant our instance role full access to AWS CloudWatch, but there are [many other managed polices](https://docs.aws.amazon.com/IAM/latest/UserGuide/access_policies_managed-vs-inline.html), which you can choose based on what permissions your application needs.

Resources:

SecurityGroup: ...

InstanceRole:

Type: "AWS::IAM::Role"

Properties:

AssumeRolePolicyDocument:

Version: "2012-10-17"

Statement:

Effect: Allow

Principal:

Service:

- "ec2.amazonaws.com"

Action: sts:AssumeRole

ManagedPolicyArns:

- arn:aws:iam::aws:policy/CloudWatchFullAccess

Tags:

- Key: Name

Value: !Ref AWS::StackName

Next, we’ll create an instance profile to tie our IAM role to the EC2 instance that we’ll create.

Resources:

SecurityGroup: ...

InstanceRole: ...

InstanceProfile:

Type: "AWS::IAM::InstanceProfile"

Properties:

Roles:

- Ref: InstanceRole

main.yml

Now it’s time to create our final resource, the EC2 instance itself.

Resources:

SecurityGroup: ...

InstanceRole: ...

InstanceProfile: ...

Instance:

Type: AWS::EC2::Instance

CreationPolicy:

ResourceSignal:

Timeout: PT15M

Count: 1

Metadata:

AWS::CloudFormation::Init:

config:

packages:

yum:

wget: []

unzip: []

Properties:

ImageId: !Ref EC2AMI

InstanceType: !Ref EC2InstanceType

IamInstanceProfile: !Ref InstanceProfile

Monitoring: true

SecurityGroupIds:

- !GetAtt SecurityGroup.GroupId

UserData:

# ...

Tags:

- Key: Name

Value: !Ref AWS::StackName

main.yml

**Line #8:** This tells CloudFormation to wait for a signal before marking the new instance as created (we’ll see how in the install script).

**Line #15:** Here we define some prerequisites that CloudFormation will install on our instance (the wget and unzip utilities). We’ll need them to install our application.

**Line #20:** The AMI ID that we take as a template parameter.

**Line #21:** The EC2 instance type that we take as a template parameter.

**Line #25:**!GetAtt is a [CloudFormation function](https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/intrinsic-function-reference-getatt.html) that can reference attributes from other resources.

**Line #27:** See the next code listing for how to fill in this part.

Next, let’s fill in the UserData section for the EC2 instance. This allows us to [run commands on our instance when it launches](https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/user-data.html).

UserData:

Fn::Base64: !Sub |

#!/bin/bash -xe

# send script output to /tmp so we can debug boot failures

exec > /tmp/userdata.log 2>&1

# Update all packages

yum -y update

# Get latest cfn scripts; https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/best-practices.html#cfninit

yum install -y aws-cfn-bootstrap

# Have CloudFormation install any files and packages from the metadata

/opt/aws/bin/cfn-init -v --stack ${AWS::StackName} --region ${AWS::Region} --resource 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

# Download latest code, unzip it into /home/ec2-user/app

wget https://github.com/<username>/aws-bootstrap/archive/master.zip

unzip master.zip

mv aws-bootstrap-master app

# Create log directory

mkdir -p /home/ec2-user/app/logs

# Run server

cd app

npm install

npm start

EOF

chown ec2-user:ec2-user /tmp/install\_script.sh && chmod a+x /tmp/install\_script.sh

sleep 1; su - ec2-user -c "/tmp/install\_script.sh"

# Signal to CloudFormation that the instance is ready

/opt/aws/bin/cfn-signal -e $? --stack ${AWS::StackName} --region ${AWS::Region} --resource Instance

main.yml

**Line #6:** The output of the UserData script will be written to /tmp/userdata.log. Look there if you need to debug any launch issues.

**Line #15:** This is where the wget and unzip utilities will get installed.

**Line #17:** This script replaces the manual setup we ran in the previous section.

**Line #32:** Replace <username> with your GitHub username.

**Line #46:** Runs the install script as the ec2-user.

**Line #49:** Signals to CloudFormation that the instance setup is complete.

In the next lesson, we will deploy our CloudFormation stack.

**Infrastructure as Code: Deploy CloudFormation Stack**

**We'll cover the following**

* [Objective](https://www.educative.io/module/lesson/the-good-parts-of-aws/7nj347qwW1G#Objective)
* [Steps](https://www.educative.io/module/lesson/the-good-parts-of-aws/7nj347qwW1G#Steps)
* [Deploying](https://www.educative.io/module/lesson/the-good-parts-of-aws/7nj347qwW1G#Deploying)
* [Github access token](https://www.educative.io/module/lesson/the-good-parts-of-aws/7nj347qwW1G#Github-access-token)
* [Run and push the code](https://www.educative.io/module/lesson/the-good-parts-of-aws/7nj347qwW1G#Run-and-push-the-code)

**Objective**

* Recreate our infrastructure using CloudFormation.

**Steps**

* Deploy the CloudFormation Stack.

**Deploying**

Now it’s time to deploy our infrastructure. Let’s run the deploy-infra.sh command. We can check the status of our stack from the [CloudFormation console](https://console.aws.amazon.com/cloudformation/). The events tab shows which resources are being created, modified, or destroyed.

When successful, the script should show us the URL for reaching our application.

./deploy-infra.sh

Waiting for changeset to be created..

Waiting for stack create/update to complete

Successfully created/updated stack - awsbootstrap

[

"http://ec2-35-174-3-173.compute-1.amazonaws.com:8080"

]

Terminal

And now we can test that our application is up and running with curl.

curl ec2-35-174-3-173.compute-1.amazonaws.com:8080

Hello World

Terminal

Now we can commit our infrastructure code to GitHub to checkpoint our progress.

git add deploy-infra.sh main.yml

git commit -m "Create infrastructure via CloudFormation"

git push

terminal

## Github access token

**NOTE:** Before we run the code, we will require a Github access token so you don’t need to put in your username and password whenever you access Github through the terminal widget. To generate an access token, go to <https://github.com/settings/tokens/new> and click Generate new token. Give it repo and admin:repo\_hook permissions, and click Generate token.

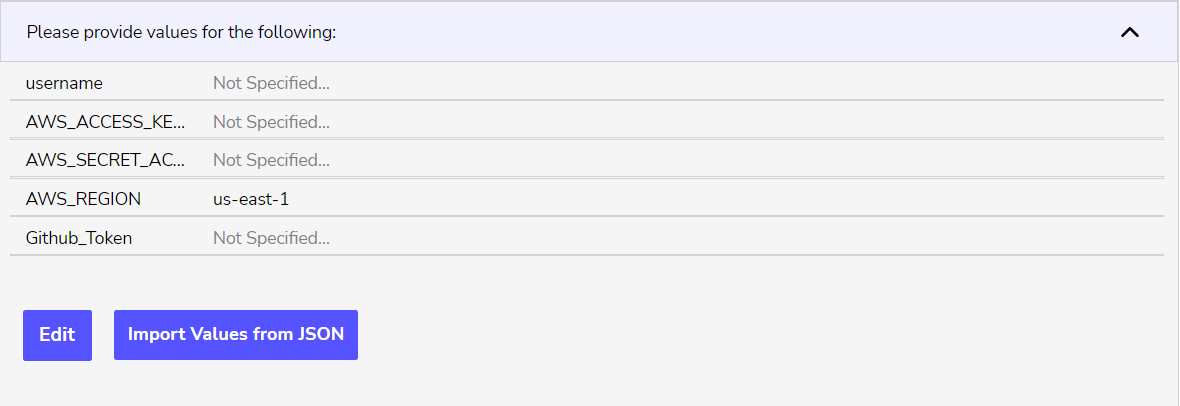
**NOTE:** You just have to put your Github token **ONCE** in the terminal widget, it will persist that information in all the other terminal widgets automatically.

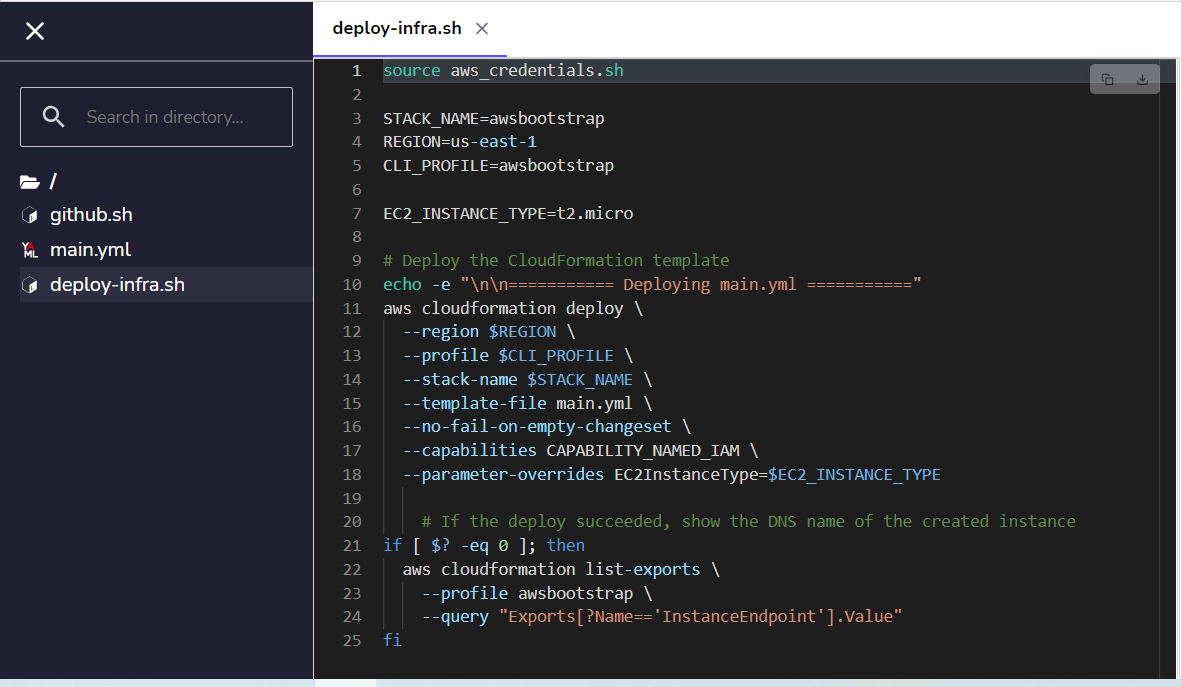
## Run and push the code

All of the files have already been placed. You just have to run the code.

**NOTE:** It will automatically push the code to your Github too.

**NOTE:** In deploy-infra.sh file, on **Line #1** (which is also highlighted in the code), another file by the name of aws\_credentials is getting executed to set your AWS credentials but you can’t see it. Similarly in github.sh, on **Line #1**, github\_credentials.sh is getting executed to set your Github credentials. We have hidden these two files so your main focus is on the code, not to set up the credentials.



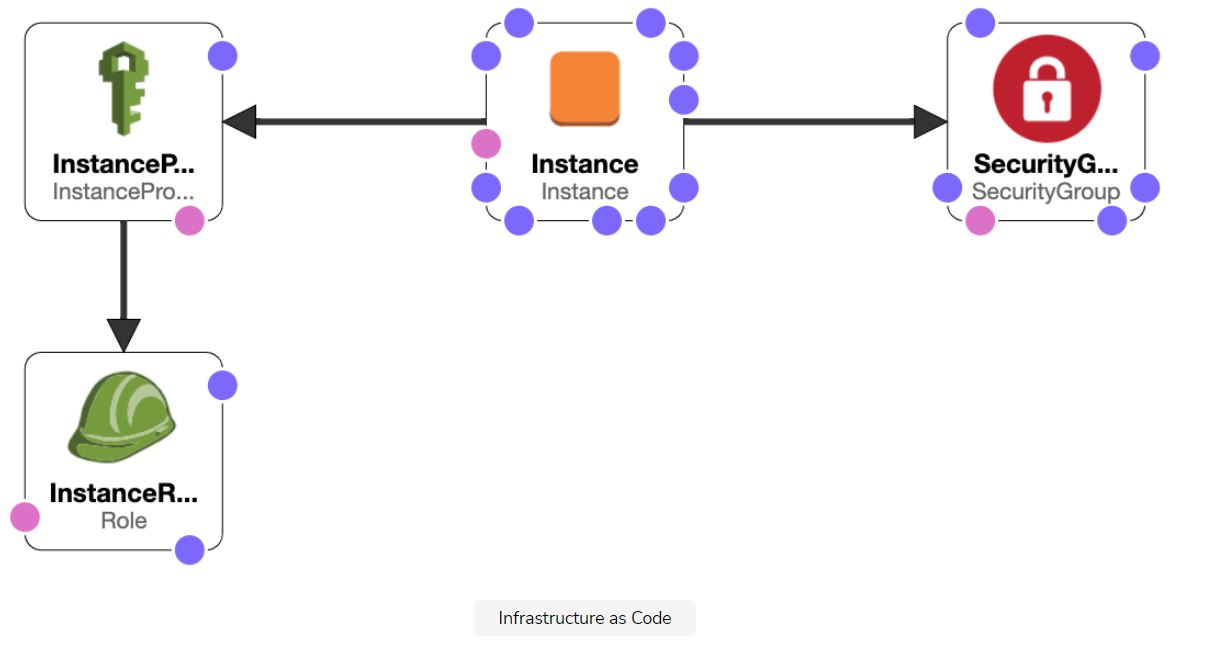


We now have our application running in the cloud, with its basic infrastructure managed through code. However, if we make a change to our application, our EC2 instance won’t be updated.

🔍 Only one action at a time can be in progress for a given CloudFormation stack. If you get an error that says your stack “is in [?] state and cannot be updated”, then wait until the stack has finished its current update and try again.

⚠️ If there is an error with the creation of your stack, you may get a message saying that your stack “is in ROLLBACK\_COMPLETE state and cannot be updated.” When this happens, you will not be able to deploy again. CloudFormation does this to give you a chance to inspect the error that caused the deployment to fail. Once you’ve addressed the issue, you’ll need to delete the stack and redeploy it.

In order to get a pictorial view of our developed CloudFormation stack so far, below is the design view which shows the resources we created and their relationships.



In the next lesson, we will make our instance receive a new version of our application automatically as soon as a change is pushed to GitHub.

**Automatic Deployments: CodeBuild**

Getting GitHub credentials, creating an S3 bucket for build artifacts, and telling CodeBuild to pull the changes from GitHub will be demonstrated in this lesson.

**We'll cover the following**

* [Objective](https://www.educative.io/module/lesson/the-good-parts-of-aws/gkXPRyyKmjY#Objective)
* [Steps](https://www.educative.io/module/lesson/the-good-parts-of-aws/gkXPRyyKmjY#Steps)
* [GitHub access token](https://www.educative.io/module/lesson/the-good-parts-of-aws/gkXPRyyKmjY#GitHub-access-token)
* [S3 bucket for build artifacts](https://www.educative.io/module/lesson/the-good-parts-of-aws/gkXPRyyKmjY#S3-bucket-for-build-artifacts)
* [Start and stop scripts](https://www.educative.io/module/lesson/the-good-parts-of-aws/gkXPRyyKmjY#Start-and-stop-scripts)
* [The build specification](https://www.educative.io/module/lesson/the-good-parts-of-aws/gkXPRyyKmjY#The-build-specification)
* [The deployment specification](https://www.educative.io/module/lesson/the-good-parts-of-aws/gkXPRyyKmjY#The-deployment-specification)

**Objective**

* Automatically update our application when a change gets pushed to GitHub.

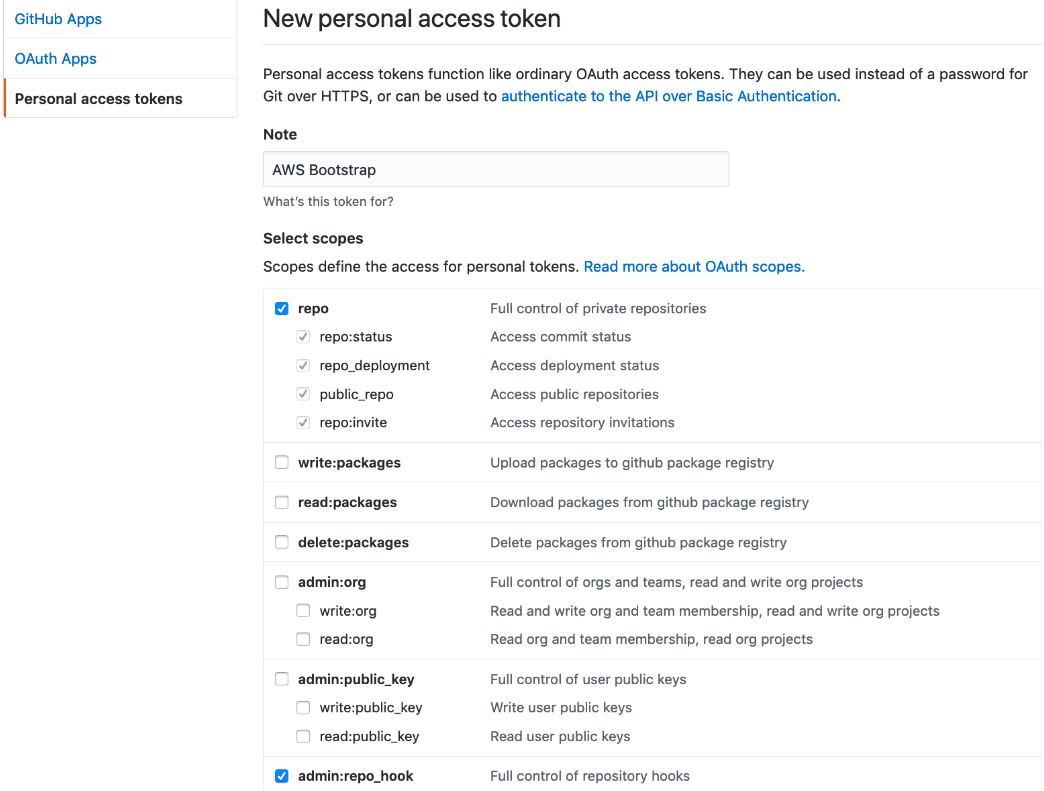
**Steps**

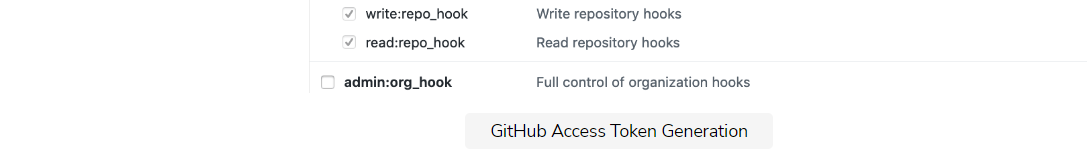
* Get GitHub credentials.
* Creating S3 bucket for build artifacts.
* CodeBuild to pull changes from GitHub.

In this section, we’re going to use CodeBuild, CodeDeploy, and CodePipeline so that our application gets updated automatically as soon as we push a change to GitHub.

**GitHub access token**

We will need a GitHub access token to let CodeBuild pull changes from GitHub. To generate an access token, go to <https://github.com/settings/tokens/new> and click *Generate new token*. Give it *repo* and *admin:repo\_hook* permissions, and click *Generate token*.





Tokens and passwords are sensitive information and should not be checked into source repositories. There are sophisticated ways to store them, but for now we’ll put our new token in a local file that we can later read into an environment variable.

mkdir -p ~/.github

echo "aws-bootstrap" > ~/.github/aws-bootstrap-repo

echo "<username>" > ~/.github/aws-bootstrap-owner

echo "<token>" > ~/.github/aws-bootstrap-access-token

terminal

**Line #3:** Replace <username> with your GitHub username.

**Line #4:** Replace <token> with your GitHub access token.

## S3 bucket for build artifacts

CodePipeline requires an S3 bucket to store artifacts built by CodeBuild. We chose to create this bucket outside of our main CloudFormation template because CloudFormation is unable to delete S3 buckets unless they’re empty. This limitation becomes very inconvenient during development, because you would have to delete the S3 bucket manually every time you tear down your CloudFormation stack. Therefore, we like to put resources such as these in a separate CloudFormation template called setup.yml.

AWSTemplateFormatVersion: 2010-09-09

Parameters:

CodePipelineBucket:

Type: String

Description: 'The S3 bucket for CodePipeline artifacts.'

Resources:

CodePipelineS3Bucket:

Type: AWS::S3::Bucket

DeletionPolicy: Retain

Properties:

BucketName: !Ref CodePipelineBucket

PublicAccessBlockConfiguration:

BlockPublicAcls: true

BlockPublicPolicy: true

IgnorePublicAcls: true

RestrictPublicBuckets: true

BucketEncryption:

ServerSideEncryptionConfiguration:

- ServerSideEncryptionByDefault:

SSEAlgorithm: AES256

setup.yml

Now let’s edit our deploy-infra.sh script to define the S3 bucket name for our CodePipeline.

AWS\_ACCOUNT\_ID=`aws sts get-caller-identity --profile awsbootstrap \

--query "Account" --output text`

CODEPIPELINE\_BUCKET="$STACK\_NAME-$REGION-codepipeline-$AWS\_ACCOUNT\_ID"

Terminal

**Line #1:** This is a way to programmatically get the AWS account ID from the AWS CLI.

**Line #3:** S3 bucket names must be globally unique across all AWS customers. Adding our account ID to the bucket name helps prevent name conflicts.

Then we need to deploy setup.yml from our deploy-infra.sh script, just before we deploy main.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

deploy-infra.sh

## Start and stop scripts

Next, we need to create a couple of simple scripts to tell CodeDeploy how to start and stop our application.

#!/bin/bash -xe

source /home/ec2-user/.bash\_profile

cd /home/ec2-user/app/release

npm run start

start-service.sh

**Line #2:** Makes sure any user-specific software that we’ve installed (e.g., npm via nvm) is available.

**Line #3:** Changes into the working directory in which our application expects to be run.

**Line #4:** Runs the start script we put in package.json.

#!/bin/bash -xe

source /home/ec2-user/.bash\_profile

[ -d "/home/ec2-user/app/release" ] && \

cd /home/ec2-user/app/release && \

npm stop

stop-service.sh

## The build specification

Next, we need to tell CodeBuild how to build our application. To do this, CodeBuild has a [specification](https://docs.aws.amazon.com/codebuild/latest/userguide/build-spec-ref.html), which we use in a file named buildspec.yml.

version: 0.2

phases:

install:

runtime-versions:

nodejs: 10

pre\_build:

commands:

# run 'npm install' using versions in package-lock.json

- npm ci

build:

commands:

- npm run build

artifacts:

files:

- start-service.sh

- stop-service.sh

- server.js

- package.json

- appspec.yml

- 'node\_modules/\*\*/\*'

buildspec.yml

## The deployment specification

Now, we need to tell CodeDeploy what to do with the build artifacts created by CodeBuild. To do this, CodeDeploy also has a [specification](https://docs.aws.amazon.com/codedeploy/latest/userguide/reference-appspec-file.html), which we use in a file named appspec.yml.

version: 0.0

os: linux

files:

# unzip the build artifact in ~/app

- source: /

destination: /home/ec2-user/app/release

permissions:

# change permissions from root to ec2-user

- object: /home/ec2-user/app/release

pattern: "\*\*"

owner: ec2-user

group: ec2-user

hooks:

ApplicationStart:

# start the application

- location: start-service.sh

timeout: 300

runas: ec2-user

ApplicationStop:

# stop the application

- location: stop-service.sh

timeout: 300

runas: ec2-user

appspec.yml

At this point, let’s commit what we have so far to GitHub.

git add appspec.yml buildspec.yml start-service.sh stop-service.sh deploy-infra.sh setup.yml

git commit -m "Add codebuild / codedeploy spec"

git push

terminal

In the next lesson, we are going to install CodeDeploy agent on our EC2 instance.

**Automatic Deployments: Install CodeDeploy Agent on EC2**

We will be installing CodeDeploy agent on our EC2 instance in this lesson.

**We'll cover the following**

* [Objective](https://www.educative.io/module/lesson/the-good-parts-of-aws/xorBXnxv4O9#Objective)
* [Steps](https://www.educative.io/module/lesson/the-good-parts-of-aws/xorBXnxv4O9#Steps)
* [The deployment pipeline](https://www.educative.io/module/lesson/the-good-parts-of-aws/xorBXnxv4O9#The-deployment-pipeline)

**Objective**

* Automatically update our application when a change gets pushed to GitHub.

**Steps**

* Install the CodeDeploy agent on our EC2 instance.

**The deployment pipeline**

Now it’s time to create the deployment pipeline in CloudFormation. Let’s start by setting up a few environment variables in our deploy-infra.sh script with information about our GitHub credentials.

# Generate a personal access token with repo and admin:repo\_hook

# permissions from https://github.com/settings/tokens

GH\_ACCESS\_TOKEN=$(cat ~/.github/aws-bootstrap-access-token)

GH\_OWNER=$(cat ~/.github/aws-bootstrap-owner)

GH\_REPO=$(cat ~/.github/aws-bootstrap-repo)

GH\_BRANCH=master

deploy-infra.sh

And then we can pass these variables to our main.yml script as parameters.

# 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 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

To do that, we also need to update the Parameters section in main.yml to receive the GitHub information.

Parameters:

EC2InstanceType:

Type: String

EC2AMI:

Type: 'AWS::SSM::Parameter::Value<AWS::EC2::Image::Id>'

Default: '/aws/service/ami-amazon-linux-latest/amzn2-ami-hvm-x86\_64-gp2'

CodePipelineBucket:

Type: String

Description: 'The S3 bucket for CodePipeline artifacts.'

GitHubOwner:

Type: String

Description: 'The username of the source GitHub repo.'

GitHubRepo:

Type: String

Description: 'The source GitHub repo name (without the username).'

GitHubBranch:

Type: String

Default: master

Description: 'The source GitHub branch.'

GitHubPersonalAccessToken:

Type: String

NoEcho: true

Description: 'A GitHub personal access token with "repo" and "admin:repo\_hook" permissions.'

main.yml

Next, we need to add a new managed policy to allow our EC2 instance to access CodeDeploy.

InstanceRole:

Type: "AWS::IAM::Role"

Properties:

AssumeRolePolicyDocument:

Version: "2012-10-17"

Statement:

Effect: Allow

Principal:

Service:

- "ec2.amazonaws.com"

Action: sts:AssumeRole

ManagedPolicyArns:

- arn:aws:iam::aws:policy/CloudWatchFullAccess

- arn:aws:iam::aws:policy/service-role/AmazonEC2RoleforAWSCodeDeploy

Tags:

- Key: Name

Value: !Ref AWS::StackName

main.yml

**Line #14:** Allows our EC2 instance to access CodeDeploy.

We also need to create a new IAM role to allow the CodeBuild, CodeDeploy, and CodePipeline services to access our AWS resources.

DeploymentRole:

Type: AWS::IAM::Role

Properties:

AssumeRolePolicyDocument:

Version: "2012-10-17"

Statement:

Effect: Allow

Principal:

Service:

- codepipeline.amazonaws.com

- codedeploy.amazonaws.com

- codebuild.amazonaws.com

Action: sts:AssumeRole

ManagedPolicyArns:

- arn:aws:iam::aws:policy/PowerUserAccess

main.yml

Then we can define our CodeBuild project.

BuildProject:

Type: AWS::CodeBuild::Project

Properties:

Name: !Ref AWS::StackName

ServiceRole: !GetAtt DeploymentRole.Arn

Artifacts:

Type: CODEPIPELINE

Environment:

Type: LINUX\_CONTAINER

ComputeType: BUILD\_GENERAL1\_SMALL

Image: aws/codebuild/standard:2.0

Source:

Type: CODEPIPELINE

main.yml

Next, we can define our CodeDeploy application. This lets CodeDeploy know that our deployment target is EC2.

DeploymentApplication:

Type: AWS::CodeDeploy::Application

Properties:

ApplicationName: !Ref AWS::StackName

ComputePlatform: Server

main.yml

**Line #5:** In this case, Server means EC2.

To complete the CodeDeploy setup, we also need to define a deployment group. For now, we’re going to have one deployment group called Staging. This will be our pre-production environment. We will add another deployment group for production when we get to the [Production](https://www.educative.io/pageeditor/10370001/5943367834796032/5107468213420032) section.

StagingDeploymentGroup:

Type: AWS::CodeDeploy::DeploymentGroup

DependsOn: Instance

Properties:

DeploymentGroupName: staging

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 #7:** For pre-production, we can choose to deploy as fast as possible. We’ll do this differently in production.

**Line #9:** These filters define how CodeDeploy will find the EC2 instances to deploy to.

And finally, we just need to define our pipeline. Let’s do that in the next lesson.

**Automatic Deployments: Create a CodePipeline**

**We'll cover the following**

* [Objective](https://www.educative.io/module/lesson/the-good-parts-of-aws/R8Mq8wDYQwz#Objective)
* [Steps](https://www.educative.io/module/lesson/the-good-parts-of-aws/R8Mq8wDYQwz#Steps)
* [Defining our pipeline](https://www.educative.io/module/lesson/the-good-parts-of-aws/R8Mq8wDYQwz#Defining-our-pipeline)

**Objective**

* Automatically update our application when a change gets pushed to GitHub.

**Steps**

* Create a CodePipeline.

**Defining our pipeline**

The pipeline comes in three stages:

1. The *Source* stage pulls the latest code from GitHub.
2. The *Build* stage builds the latest code with CodeBuild according to our buildspec.yml file.
3. The *Deploy* stage deploys the build artifacts from CodeBuild to the EC2 instances referenced in the deployment group, and starts the application according to our appspec.yml file.
4. Pipeline:
5. Type: AWS::CodePipeline::Pipeline
6. Properties:
7. Name: !Ref AWS::StackName
8. ArtifactStore:
9. Location: !Ref CodePipelineBucket
10. Type: S3
11. RoleArn: !GetAtt DeploymentRole.Arn
12. Stages:
13. - Name: Source
14. Actions:
15. - Name: Source
16. ActionTypeId:
17. Category: Source
18. Owner: ThirdParty
19. Version: 1
20. Provider: GitHub
21. OutputArtifacts:
22. - Name: Source
23. Configuration:
24. Owner: !Ref GitHubOwner
25. Repo: !Ref GitHubRepo
26. Branch: !Ref GitHubBranch
27. OAuthToken: !Ref GitHubPersonalAccessToken
28. PollForSourceChanges: false
29. RunOrder: 1
30. - Name: Build
31. Actions:
32. - Name: Build
33. ActionTypeId:
34. Category: Build
35. Owner: AWS
36. Version: 1
37. Provider: CodeBuild
38. InputArtifacts:
39. - Name: Source
40. OutputArtifacts:
41. - Name: Build
42. Configuration:
43. ProjectName: !Ref BuildProject
44. RunOrder: 1
45. - Name: Staging
46. Actions:
47. - Name: Staging
48. InputArtifacts:
49. - Name: Build
50. ActionTypeId:
51. Category: Deploy
52. Owner: AWS
53. Version: 1
54. Provider: CodeDeploy
55. Configuration:
56. ApplicationName: !Ref DeploymentApplication
57. DeploymentGroupName: !Ref StagingDeploymentGroup
58. RunOrder: 1

main.yml

**Line #25:** We don’t need to poll for changes because we’ll set up a webhook to trigger a deployment as soon as GitHub receives a change.

Now, let’s create the webhook that will trigger our pipeline as soon as a change is pushed to GitHub.

PipelineWebhook:

Type: AWS::CodePipeline::Webhook

Properties:

Authentication: GITHUB\_HMAC

AuthenticationConfiguration:

SecretToken: !Ref GitHubPersonalAccessToken

Filters:

- JsonPath: $.ref

MatchEquals: 'refs/heads/{Branch}'

TargetPipeline: !Ref Pipeline

TargetAction: Source

Name: !Sub 'webhook-${AWS::StackName}'

TargetPipelineVersion: !GetAtt Pipeline.Version

RegisterWithThirdParty: true

main.yml

We also need to make some changes to our EC2 instance to get the CodeDeploy agent installed on it.

Instance:

Type: AWS::EC2::Instance

CreationPolicy:

ResourceSignal:

Timeout: PT5M

Count: 1

Metadata:

AWS::CloudFormation::Init:

config:

packages:

yum:

ruby: []

files:

/home/ec2-user/install:

source: !Sub "https://aws-codedeploy-${AWS::Region}.s3.amazonaws.com/latest/install"

mode: "000755" # executable

commands:

00-install-cd-agent:

command: "./install auto"

cwd: "/home/ec2-user/"

Properties:

ImageId: !Ref EC2AMI

InstanceType: !Ref EC2InstanceType

IamInstanceProfile: !Ref InstanceProfile

Monitoring: true

SecurityGroupIds:

- !GetAtt SecurityGroup.GroupId

UserData:

# ...

Tags:

- Key: Name

Value: !Ref AWS::StackName

main.yml

**Line #12:** The CodeDeploy agent requires ruby.

**Line #14:** Downloads the CodeDeploy agent install script to /home/ec2-user/install and makes it executable.

**Line #18:** Installs the CodeDeploy agent.

**Line #29:** See the next code listing for how to fill in this part.

Let’s update the UserData section next. We need to remove the bits where we were downloading our application from GitHub because CodeDeploy will do that for us now.

UserData:

Fn::Base64: !Sub |

#!/bin/bash -xe

# send script output to /tmp so we can debug boot failures

exec > /tmp/userdata.log 2>&1

# Update all packages

yum -y update

# Get latest cfn scripts; https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/best-practices.html#cfninit

yum install -y aws-cfn-bootstrap

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

EOF

chown ec2-user:ec2-user /tmp/install\_script.sh && chmod a+x /tmp/install\_script.sh

sleep 1; su - ec2-user -c "/tmp/install\_script.sh"

# Have CloudFormation install any files and packages from the metadata

/opt/aws/bin/cfn-init -v --stack ${AWS::StackName} --region ${AWS::Region} --resource Instance

# Signal to CloudFormation that the instance is ready

/opt/aws/bin/cfn-signal -e $? --stack ${AWS::StackName} --region ${AWS::Region} --resource Instance

main.yml

And with all of that done, we can deploy our infrastructure updates. But first, we need to delete our stack from the CloudFormation console, because the changes we’ve made will not trigger CloudFormation to tear down our EC2 instance and start a new one. So, let’s delete our stack, and recreate it by running the deploy-infra.sh script.

./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

[

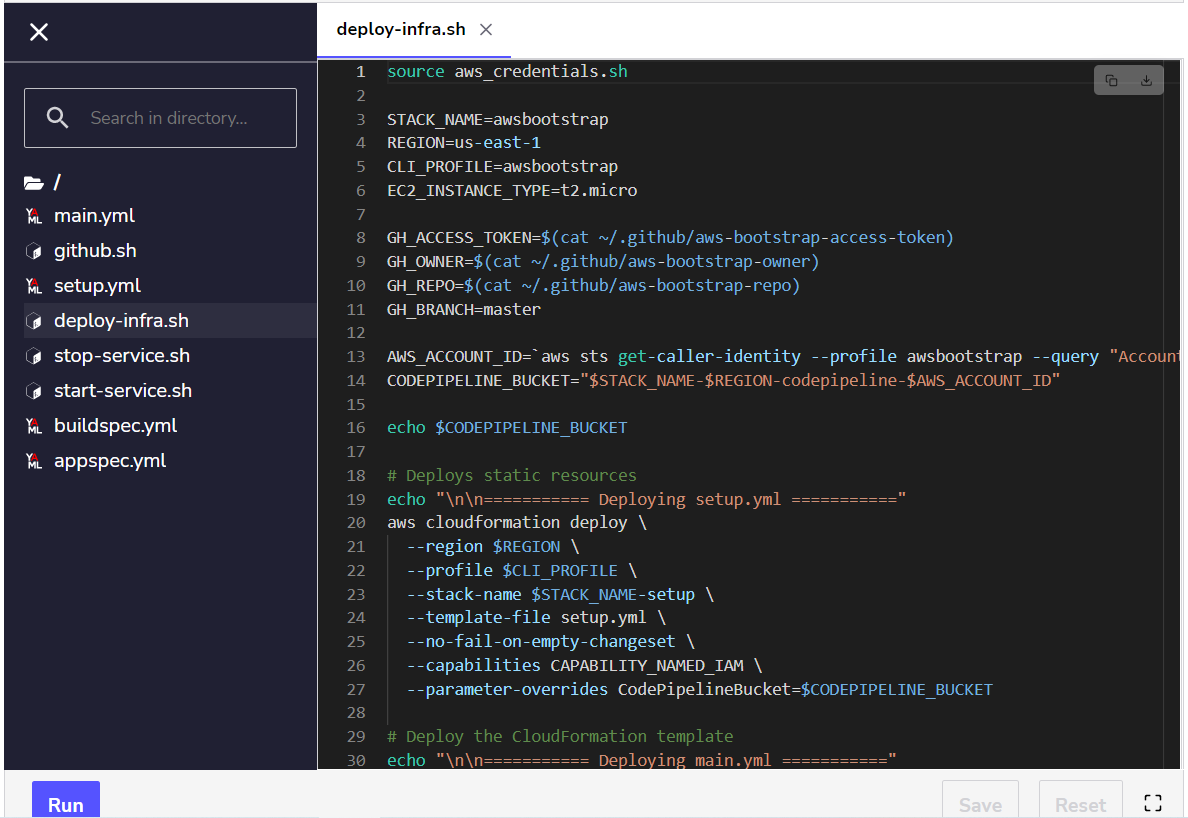
"http://ec2-3-93-145-152.compute-1.amazonaws.com:8080"

]

Terminal

**NOTE:** Let’s run the code and also push all our infrastructure changes to our GitHub repository. Check out the github.sh file for that.





source aws\_credentials.sh

STACK\_NAME=awsbootstrap

REGION=us-east-1

CLI\_PROFILE=awsbootstrap

EC2\_INSTANCE\_TYPE=t2.micro

GH\_ACCESS\_TOKEN=$(cat ~/.github/aws-bootstrap-access-token)

GH\_OWNER=$(cat ~/.github/aws-bootstrap-owner)

GH\_REPO=$(cat ~/.github/aws-bootstrap-repo)

GH\_BRANCH=master

AWS\_ACCOUNT\_ID=`aws sts get-caller-identity --profile awsbootstrap --query "Account" --output text`

CODEPIPELINE\_BUCKET="$STACK\_NAME-$REGION-codepipeline-$AWS\_ACCOUNT\_ID"

echo $CODEPIPELINE\_BUCKET

# Deploys static resources

echo "\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

# Deploy the CloudFormation template

echo "\n\n=========== Deploying main.yml ==========="

aws cloudformation deploy \

--region $REGION \

--profile $CLI\_PROFILE \

--stack-name $STACK\_NAME \

--template-file 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

# If the deploy succeeded, show the DNS name of the created instance

if [ $? -eq 0 ]; then

aws cloudformation list-exports \

--profile awsbootstrap \

--query "Exports[?Name=='InstanceEndpoint'].Value"

Fi

At this point, our EC2 instance should be up and running with the CodeDeploy agent running on it. But the CodeDeploy agent doesn’t automatically deploy the application when it gets installed. For now, we can trigger the first deployment manually by hitting *Release Change* in the [CodePipeline console](https://console.aws.amazon.com/codesuite/codepipeline/pipelines" \t "_blank). When we get to the [Scaling](https://www.educative.io/courses/good-parts-of-aws/m2v7M6Bp0qr) section, we will have our EC2 instances deploy the application automatically as soon as they start.

As soon as the deployment completes, we should be able to see the “Hello World” message when we visit the URL we got after running deploy-infra.sh.

We can now test our automatic deployments by making a change to the “Hello World” message in our application. Let’s change it to “Hello Cloud” and push the changes to GitHub.

1

const message = 'Hello Cloud\n';

server.js

git add server.js

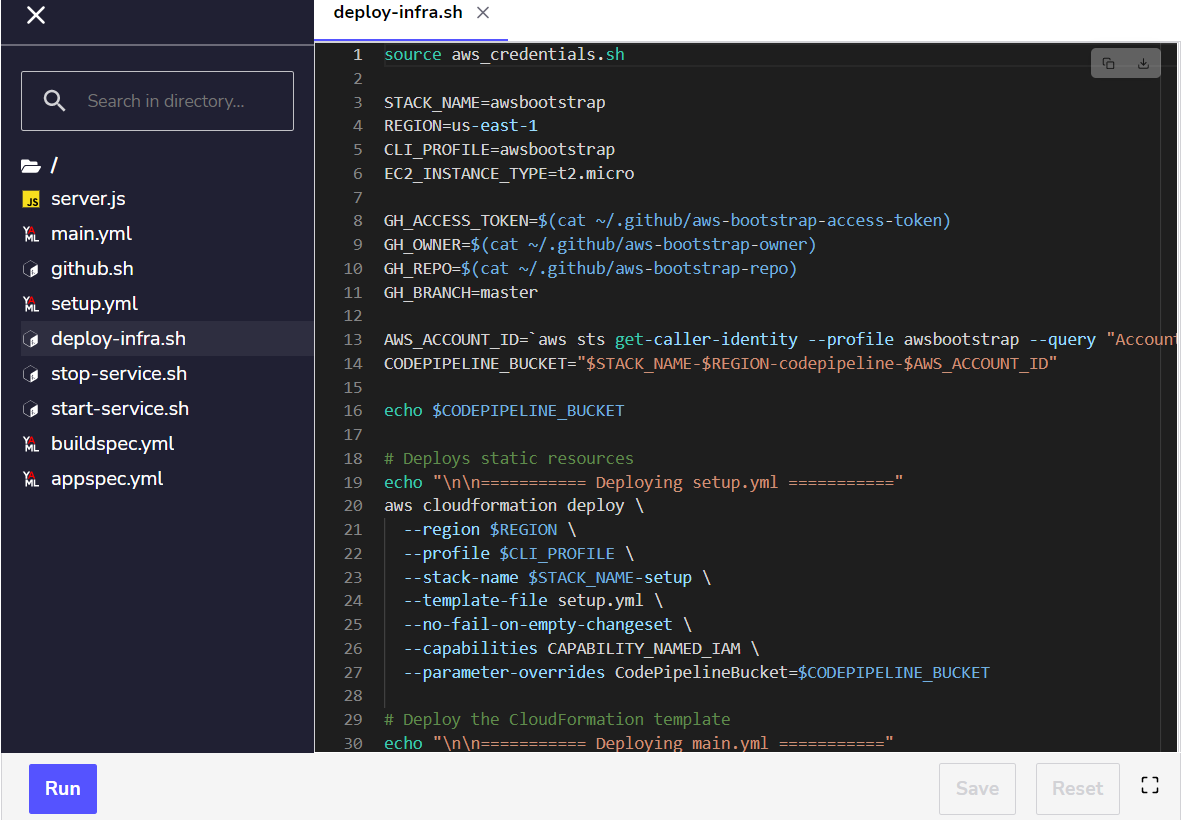
git commit -m "Change Hello World to Hello Cloud"

git push

terminal

**NOTE:** We have added server.js this time too. So before running the code, do change the message from Hello World\n to Hello Cloud\n on **Line #3**. Then you can curl to see if it works.





source aws\_credentials.sh

STACK\_NAME=awsbootstrap

REGION=us-east-1

CLI\_PROFILE=awsbootstrap

EC2\_INSTANCE\_TYPE=t2.micro

GH\_ACCESS\_TOKEN=$(cat ~/.github/aws-bootstrap-access-token)

GH\_OWNER=$(cat ~/.github/aws-bootstrap-owner)

GH\_REPO=$(cat ~/.github/aws-bootstrap-repo)

GH\_BRANCH=master

AWS\_ACCOUNT\_ID=`aws sts get-caller-identity --profile awsbootstrap --query "Account" --output text`

CODEPIPELINE\_BUCKET="$STACK\_NAME-$REGION-codepipeline-$AWS\_ACCOUNT\_ID"

echo $CODEPIPELINE\_BUCKET

# Deploys static resources

echo "\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

# Deploy the CloudFormation template

echo "\n\n=========== Deploying main.yml ==========="

aws cloudformation deploy \

--region $REGION \

--profile $CLI\_PROFILE \

--stack-name $STACK\_NAME \

--template-file 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

# If the deploy succeeded, show the DNS name of the created instance

if [ $? -eq 0 ]; then

aws cloudformation list-exports \

--profile awsbootstrap \

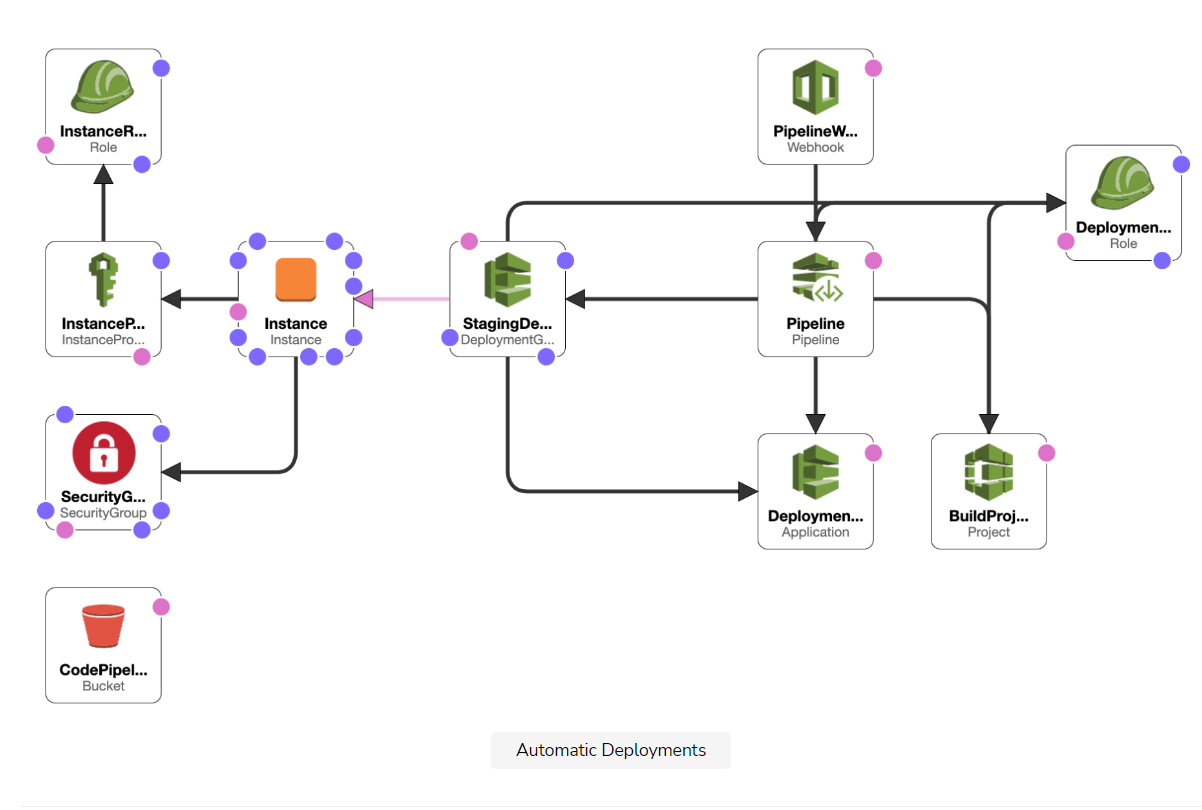
--query "Exports[?Name=='InstanceEndpoint'].Value"

Fi

As soon as we push the changes to GitHub, we can watch the deployment progress in the [CodePipeline console](https://console.aws.amazon.com/codesuite/codepipeline/pipelines" \t "_blank). As soon as the deployment reaches the Staging phase, we should see “Hello Cloud” when we refresh the URL.

Our application is now getting updated automatically as soon as a change gets pushed to GitHub. And since we’re now using GitHub access tokens, we can also mark our repository as private.

In order to get a pictorial view of our developed cloudformation stack so far, below is the design view which shows the resources we created and their relationships.



In the next lesson, we will run our application on more than one EC2 instance.

**Load Balancing: Add a second EC2 Instance**

**We'll cover the following**

* [Objective](https://www.educative.io/module/lesson/the-good-parts-of-aws/YVPPW3wxNPO#Objective)
* [Steps](https://www.educative.io/module/lesson/the-good-parts-of-aws/YVPPW3wxNPO#Steps)
* [Adding a second instance](https://www.educative.io/module/lesson/the-good-parts-of-aws/YVPPW3wxNPO#Adding-a-second-instance)

**Objective**

* Run our application on more than one EC2 instance.

**Steps**

* Add a second EC2 instance.

Currently, our application is running on a single EC2 instance. To allow our application to scale beyond the capacity of a single instance, we need to introduce a load balancer that can direct traffic to multiple instances.

**Adding a second instance**

We could naively add a second instance by simply duplicating the configuration we have for our existing instance. But that would create a lot of configuration duplication. Instead, we’re going to pull the bulk of the EC2 configuration into an EC2 launch template, and then we’ll simply reference the launch template from both instances.

We can almost copy our EC2 instance configuration into a new launch template resource as is, but there are slight differences between the two specifications. In addition, we’ll also need to change the cfn-init and cfn-signal calls at the end of the UserData script to dynamically determine the instance ID at runtime.

InstanceLaunchTemplate:

Type: AWS::EC2::LaunchTemplate

Metadata:

AWS::CloudFormation::Init:

config:

packages:

yum:

ruby: []

jq: []

files:

/home/ec2-user/install:

source: !Sub "https://aws-codedeploy-${AWS::Region}.s3.amazonaws.com/latest/install"

mode: "000755" # executable

commands:

00-install-cd-agent:

command: "./install auto"

cwd: "/home/ec2-user/"

Properties:

LaunchTemplateName: !Sub 'LaunchTemplate\_${AWS::StackName}'

LaunchTemplateData:

ImageId: !Ref EC2AMI

InstanceType: !Ref EC2InstanceType

IamInstanceProfile:

Arn: !GetAtt InstanceProfile.Arn

Monitoring:

Enabled: true

SecurityGroupIds:

- !GetAtt SecurityGroup.GroupId

UserData:

# ...

main.yml

**Line #30:** See the next code listing for how to fill in this part.

Now let’s update the UserData script.

UserData:

Fn::Base64: !Sub |

#!/bin/bash -xe

# send script output to /tmp so we can debug boot failures

exec > /tmp/userdata.log 2>&1

# Update all packages

yum -y update

# Get latest cfn scripts; https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/best-practices.html#cfninit

yum install -y aws-cfn-bootstrap

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

EOF

chown ec2-user:ec2-user /tmp/install\_script.sh && chmod a+x /tmp/install\_script.sh

sleep 1; su - ec2-user -c "/tmp/install\_script.sh"

# Have CloudFormation install any files and packages from the metadata

/opt/aws/bin/cfn-init -v --stack ${AWS::StackName} --region ${AWS::Region} --resource InstanceLaunchTemplate

# Query the EC2 metadata service for this instance's instance-id

export INSTANCE\_ID="`wget -q -O - http://169.254.169.254/latest/meta-data/instance-id`"

# Query EC2 describeTags method and pull our the CFN Logical ID for this instance

export LOGICAL\_ID=`aws --region ${AWS::Region} ec2 describe-tags \

--filters "Name=resource-id,Values=${!INSTANCE\_ID}" \

"Name=key,Values=aws:cloudformation:logical-id" \

| jq -r ".Tags[0].Value"`

# Signal to CloudFormation that the instance is ready

/opt/aws/bin/cfn-signal -e $? --stack ${AWS::StackName} --region ${AWS::Region} --resource ${!LOGICAL\_ID}

main.yml

**Line #39:** We’re using the [Instance Metadata service](https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/instancedata-data-retrieval.html) to get the instance id.

**Line #41:** Here, we’re getting the tags associated with this instance. The aws:cloudformation:logical-id tag is automatically attached by CloudFormation. Its value is what we pass to cfn-signal to signal a successful launch.

**Line #41:** Note the usage of ${!INSTANCE\_ID}. Since this is inside a CloudFormation !Sub, if we used ${INSTANCE\_ID}, CloudFormation would have tried to do the substitution itself. Adding the ! tells CloudFormation to [rewrite it](https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/intrinsic-function-reference-sub.html) for bash to interpret.

Now, we can change our instance resource to reference the new launch template.

Instance:

Type: AWS::EC2::Instance

CreationPolicy:

ResourceSignal:

Timeout: PT5M

Count: 1

Properties:

LaunchTemplate:

LaunchTemplateId: !Ref InstanceLaunchTemplate

Version: !GetAtt InstanceLaunchTemplate.LatestVersionNumber

Tags:

- Key: Name

Value: !Ref AWS::StackName

main.yml

**Line #10:** Each time we update our launch template, it will get a new version number. We always want to use the latest.

Adding a second instance is now as easy as creating a new instance resource that references the same launch template.

Instance2:

Type: AWS::EC2::Instance

CreationPolicy:

ResourceSignal:

Timeout: PT5M

Count: 1

Properties:

LaunchTemplate:

LaunchTemplateId: !Ref InstanceLaunchTemplate

Version: !GetAtt InstanceLaunchTemplate.LatestVersionNumber

Tags:

- Key: Name

Value: !Ref AWS::StackName

main.yml

We also need to add an inline IAM policy to allow the UserData script to access the EC2 DescribeTags API. Let’s modify the InstanceRole resource to add it.

InstanceRole:

Type: "AWS::IAM::Role"

Properties:

AssumeRolePolicyDocument:

Version: "2012-10-17"

Statement:

Effect: Allow

Principal:

Service:

- "ec2.amazonaws.com"

Action: sts:AssumeRole

ManagedPolicyArns:

- arn:aws:iam::aws:policy/CloudWatchFullAccess

- arn:aws:iam::aws:policy/service-role/AmazonEC2RoleforAWSCodeDeploy

Policies:

- PolicyName: ec2DescribeTags

PolicyDocument:

Version: 2012-10-17

Statement:

- Effect: Allow

Action: 'ec2:DescribeTags'

Resource: '\*'

Tags:

- Key: Name

Value: !Ref AWS::StackName

main.yml

**Line #16:** This policy allows our instance to query EC2 for tags.

Now, we can change the output of our CloudFormation template to return a URL for both instances.

Outputs:

InstanceEndpoint1:

Description: The DNS name for the created instance

Value: !Sub "http://${Instance.PublicDnsName}:8080"

Export:

Name: InstanceEndpoint1

InstanceEndpoint2:

Description: The DNS name for the created instance

Value: !Sub "http://${Instance2.PublicDnsName}:8080"

Export:

Name: InstanceEndpoint2

main.yml

Finally, let’s change our deploy-infra.sh script to give us these URLs.

# If the deploy succeeded, show the DNS name of the created instance

if [ $? -eq 0 ]; then

aws cloudformation list-exports \

--profile awsbootstrap \

--query "Exports[?starts\_with(Name,'InstanceEndpoint')].Value"

Fi

deploy-infra.sh

If we run the deploy-infra.sh script now, we should see a pair of URLs when the deployment finishes.

./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://ec2-52-91-223-254.compute-1.amazonaws.com:8080",

"http://ec2-3-93-145-152.compute-1.amazonaws.com:8080"

]

Terminal

Our old instance should have been terminated, and two new instances should have started in its place.

At this point, let’s also checkpoint our progress into GitHub.

git add main.yml deploy-infra.sh

git commit -m "Add a second instance"

git push

terminal

Next, let’s modify our application a little bit to allow us to see how our requests get routed. We can do this by simply including the hostname in the response message.

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}/`);

});

server.js

**Line #3:** Changes the message to include the hostname.

Let’s push this change to GitHub and wait for the deployment to finish.

git add server.js

git commit -m "Add hostname to hello world message"

git push

terminal

We can follow the deployment progress from the [CodePipeline console](https://console.aws.amazon.com/codesuite/codepipeline/pipelines" \t "_blank). Once the deployment is complete, we can verify our change by making a request to both URLs.

curl http://ec2-52-91-223-254.compute-1.amazonaws.com:8080

Hello World from ip-10-0-113-245.ec2.internal

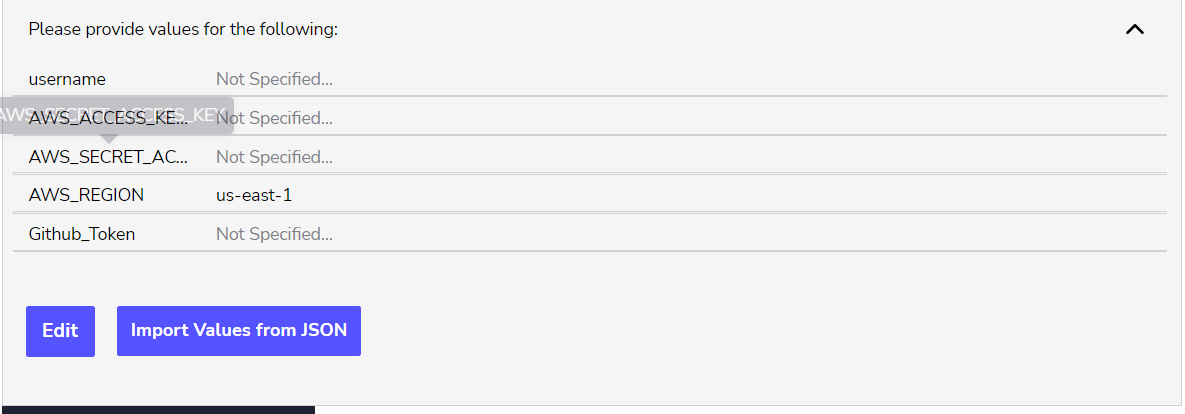
curl http://ec2-3-93-145-152.compute-1.amazonaws.com:8080

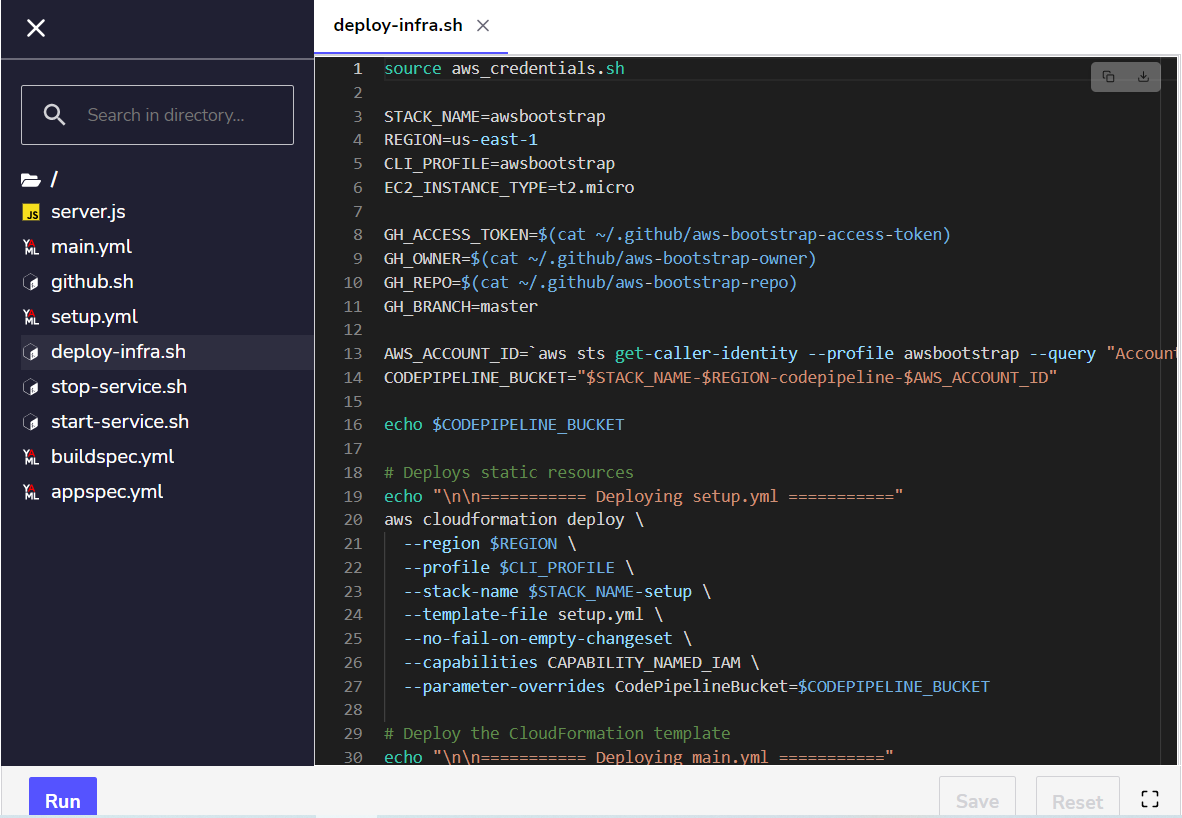
Hello World from ip-10-0-61-251.ec2.internal

Terminal

🔍 The hostname that the server is printing is not the same as the Public DNS assigned to the instance. It is actually a private domain name that EC2 assigns to each instance. You can find this private domain in the EC2 console under the Private DNS field.

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





source aws\_credentials.sh

STACK\_NAME=awsbootstrap

REGION=us-east-1

CLI\_PROFILE=awsbootstrap

EC2\_INSTANCE\_TYPE=t2.micro

GH\_ACCESS\_TOKEN=$(cat ~/.github/aws-bootstrap-access-token)

GH\_OWNER=$(cat ~/.github/aws-bootstrap-owner)

GH\_REPO=$(cat ~/.github/aws-bootstrap-repo)

GH\_BRANCH=master

AWS\_ACCOUNT\_ID=`aws sts get-caller-identity --profile awsbootstrap --query "Account" --output text`

CODEPIPELINE\_BUCKET="$STACK\_NAME-$REGION-codepipeline-$AWS\_ACCOUNT\_ID"

echo $CODEPIPELINE\_BUCKET

# Deploys static resources

echo "\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

# Deploy the CloudFormation template

echo "\n\n=========== Deploying main.yml ==========="

aws cloudformation deploy \

--region $REGION \

--profile $CLI\_PROFILE \

--stack-name $STACK\_NAME \

--template-file 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

# If the deploy succeeded, show the DNS name of the created instance

if [ $? -eq 0 ]; then

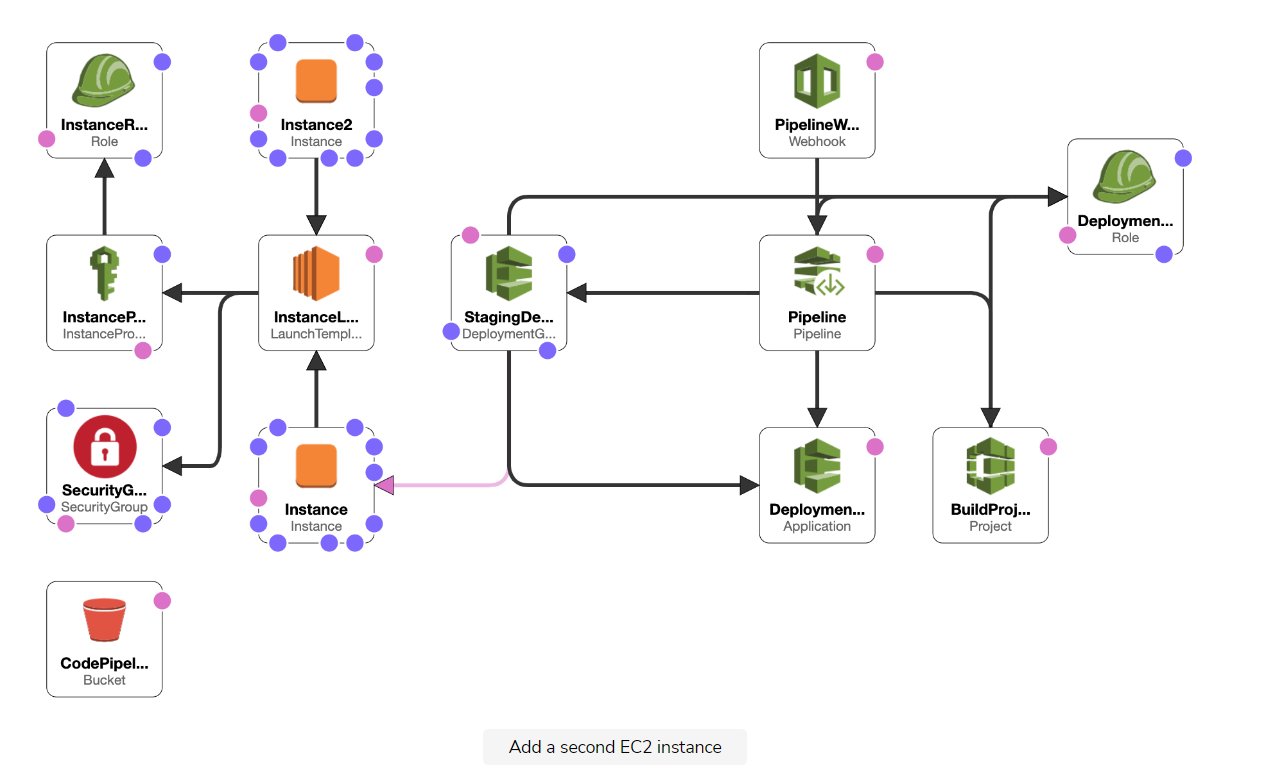
aws cloudformation list-exports \

--profile awsbootstrap \

--query "Exports[?starts\_with(Name,'InstanceEndpoint')].Value"

Fi

In order to get a pictorial view of our developed cloudformation stack so far, below is the design view which shows the resources we created and their relationships.



Let’s go ahead and add a load balancer to have a single endpoint for our application in the next part of this lesson.