Running a MEAN web application in Docker containers on AWS

Written by: Vishal Kumar

June 19, 2015

11 min read

The rate of adoption of Docker as a containerized solution is soaring. A lot of companies are now using Docker containers to run apps. In a lot of scenarios, using Docker containers can be a better approach than spinning up a full-blown virtual machine.

In this post, I'll break down all the steps I took to successfully install and run a web application built on the MEAN stack (MongoDB, Express, AngularJS, and Node.js). I hosted the application in Docker containers on Amazon Web Services (AWS).

Also, I ran the MongoDB database and the web application in separate containers. There are **lots of benefits to this approach** of having isolated environments:

1. Since each container has its own runtime environment, it's easy to modify the environment of one application component without affecting other parts. We can change the installed software or try out different versions of the softwares, until we figure out the best possible setup for that specific component.
2. Since our application components are isolated, security issues are easy to deal with. If a container is attacked or a malicious script ends up being inadvertently run as part of an update, our other containers are still safe.
3. Since it is easy to switch out and change the connected containers, testing becomes a lot easier. For example, if we want to test our web application with different sets of data, we can do that easily by connecting it to different containers set up for different database environments.

MEAN Web Framework

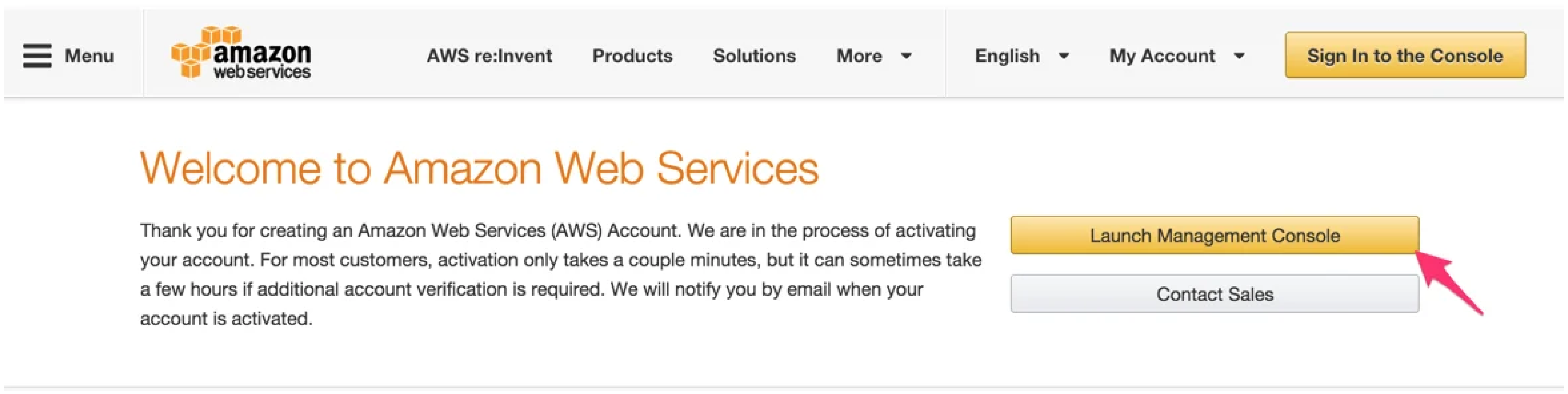
The web application that we're going to run is the framework code for MEAN.JS. This full-stack JavaScript solution builds fast, robust, and maintainable production web applications using MongoDB, Express, AngularJS, and Node.js.

Another great advantage of MEAN.JS is that we can use Yeoman Generators to create the scaffolding for our application in minutes. It also has CRUD generators which I have used heavily when adding new features to the application. The best part is that it is already well set up to support Docker deployment. It comes with a Dockerfile that can be built to create the container image, although we will use a prebuilt image to do it even faster (more on this later).

Running Docker on an Amazon Instance

You might already be aware that you can use basic AWS services free for a full year. The following steps will walk you through how to configure and run a virtual machine on AWS along with Docker service:

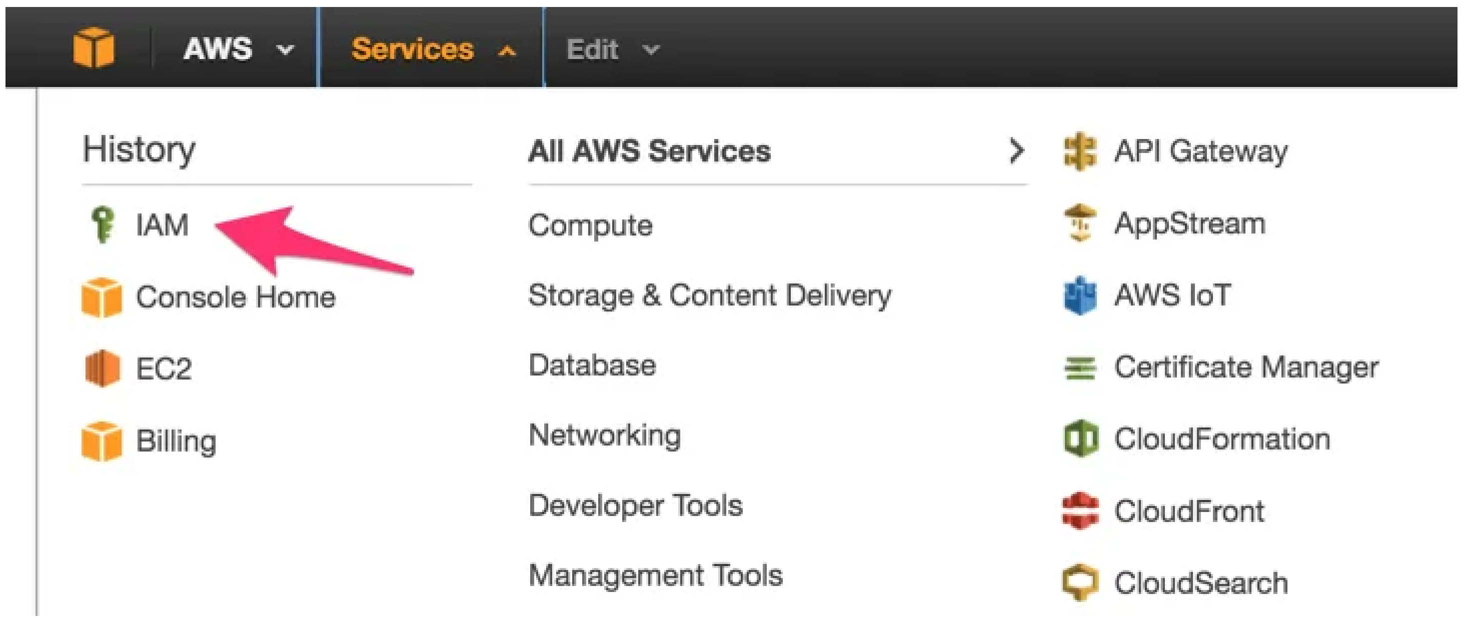
1. To begin, create your free account on AWS.amazon.com. Make sure to choose the Basic (Free) support plan. You will be redirected the AWS welcome page.
2. On this page, click on *Launch Management Console*.



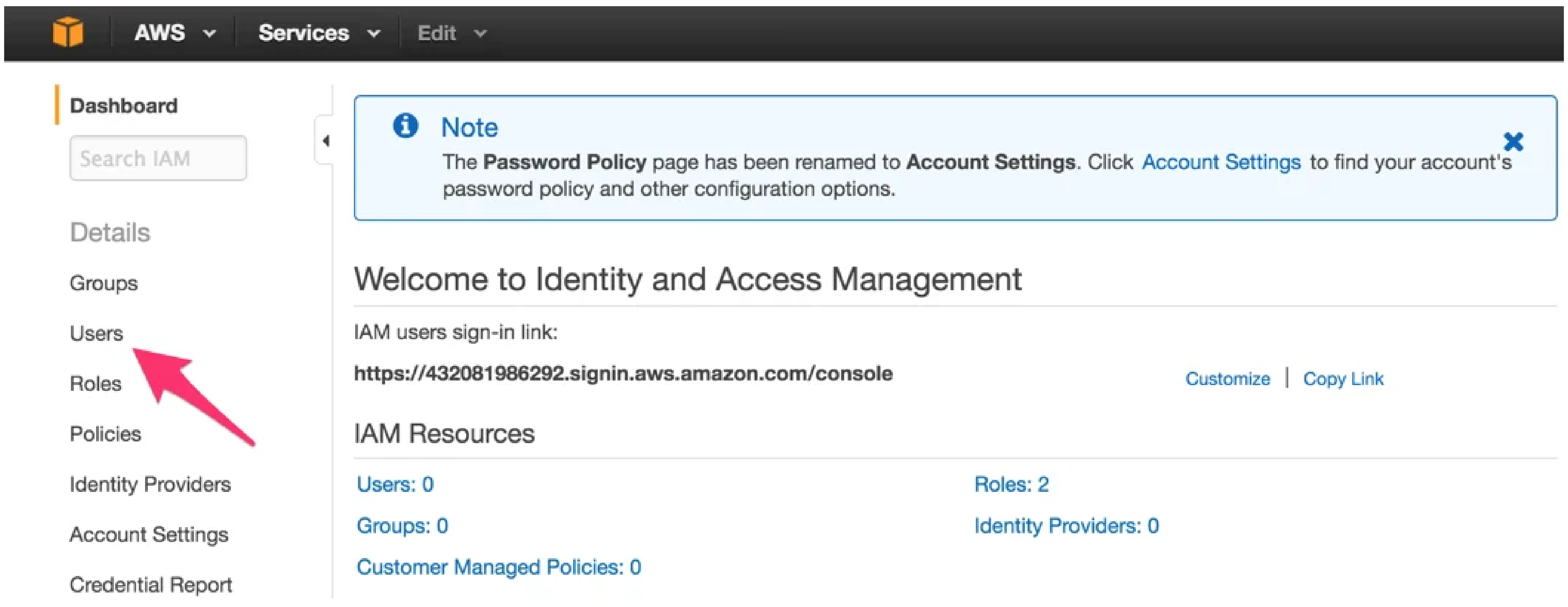
Welcome to Amazon Web Services

Thank you for creating an Amazon Web Services (AWS) Account. We are in the process of activating your account. For most customers, activation only takes a couple minutes, but it can sometimes take a few hours if additional account verification is required. We will notify you by email when your account is activated.

1. We will first go through the steps to create a user with the required credentials to manage our instance. Go to *Services* and then *IAM*.



1. Go to the *Users* link on the left.



Note

The **Password Policy** page has been renamed to **Account Settings**. Click Account Settings to find your account's password policy and other configuration options.

Welcome to Identity and Access Management

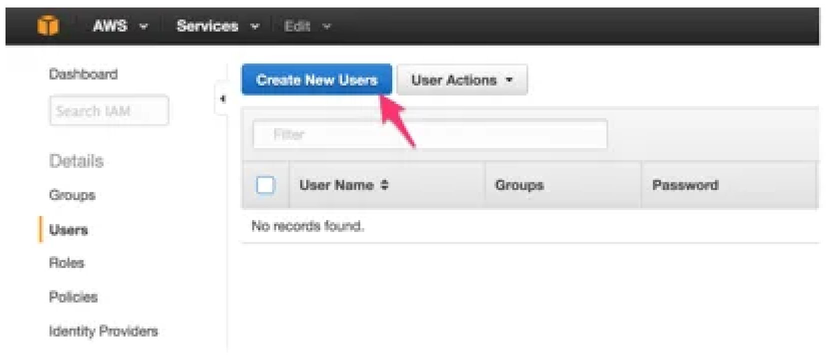
IAM users sign-in link:

https://432081986292.signin.aws.amazon.com/console

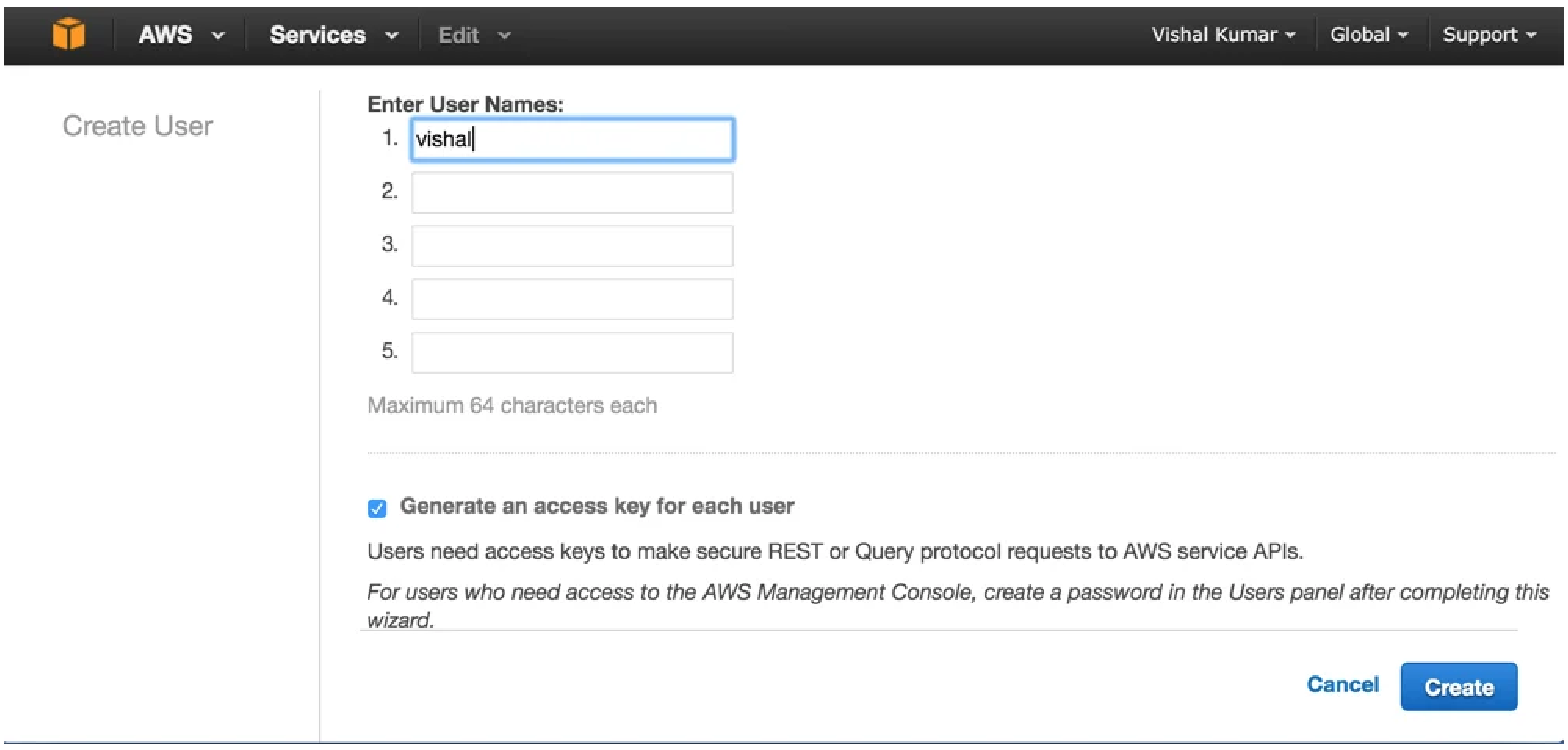
IAM Resources

Users: 0
Groups: 0
Customer Managed Policies: 0
Roles: 2
Identity Providers: 0

1. Click *Create New Users*.



1. Create a new user by providing a username. Make sure you have the checkbox to *Generate an access key for each user* checked.



Enter User Names:

1. vishal
2.
3.
4.
5.

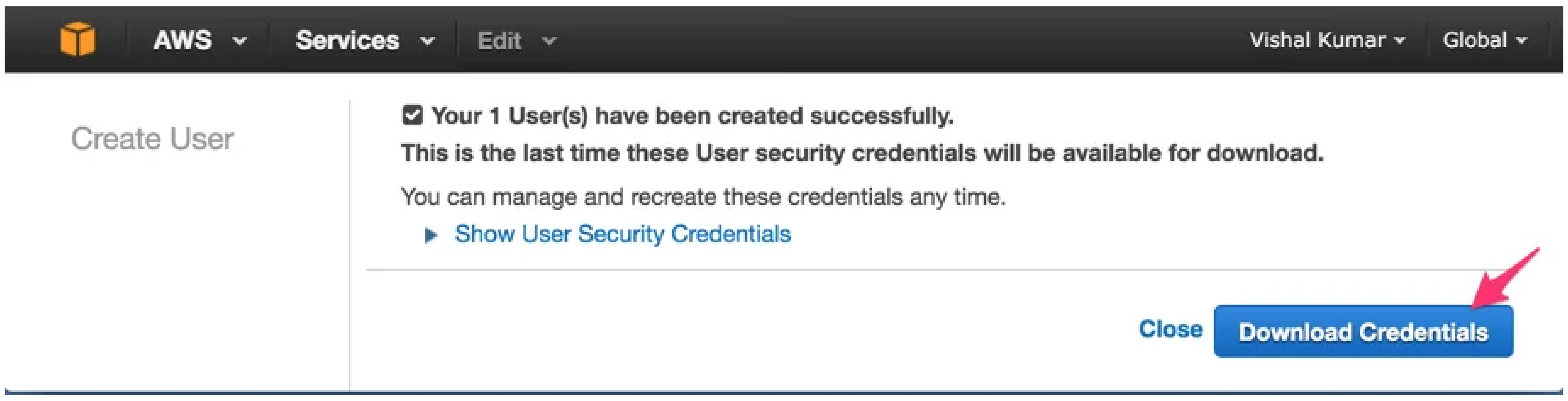
Maximum 64 characters each

☑ **Generate an access key for each user**

Users need access keys to make secure REST or Query protocol requests to AWS service APIs.

For users who need access to the AWS Management Console, create a password in the Users panel after completing this wizard.

1. Once the user is created, you will get the option to *Download Credentials* for this user. Download the file. This file will have the *Access Key Id* and the *Secret Access Key* for this user. We will need to use these credentials to connect remotely to AWS.



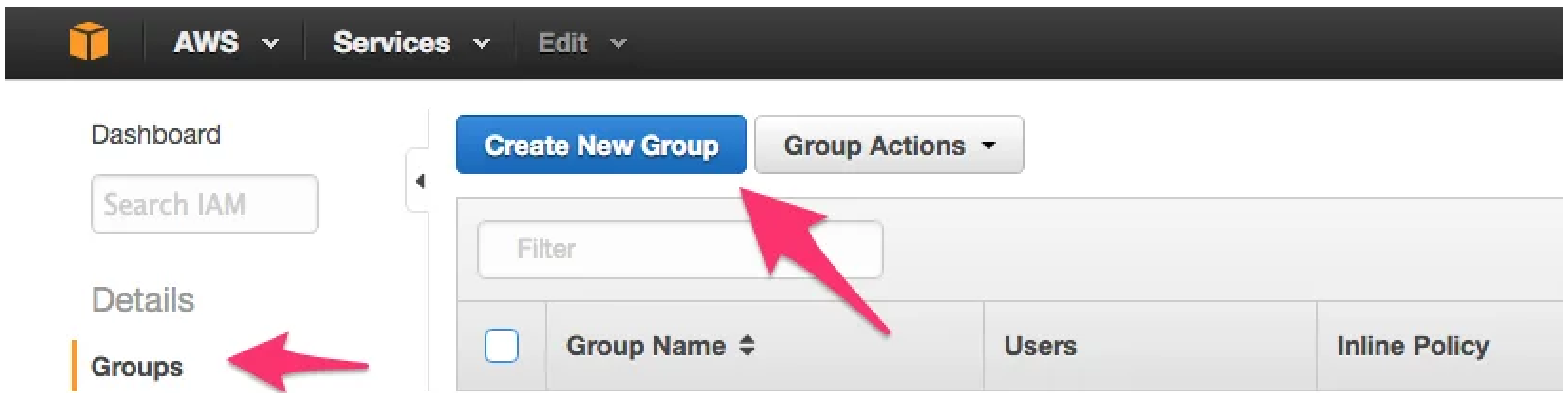
Your 1 User(s) have been created successfully.
This is the last time these User security credentials will be available for download.
You can manage and recreate these credentials any time.
▶ Show User Security Credentials

1. The recommended approach for using credentials when connecting remotely to AWS is to keep the credentials in a file at the path `~/.aws/credentials`. Create this path if it does not exist in your local computer and then create a file with the name *credentials*. The content of this file should look like this:

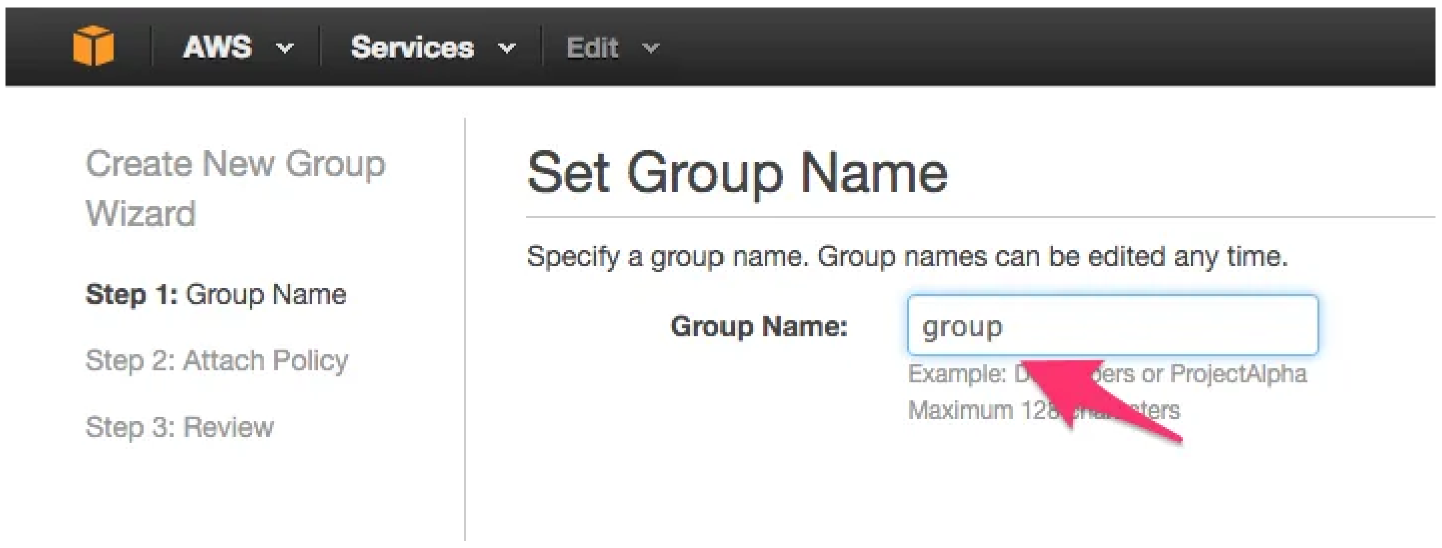
```
[default]
aws_access_key_id = [access key from the downloaded credential file]
aws_secret_access_key = [secret access key from the downloaded credential
file]
```

After creating the user and storing his credentials locally, we will also need to give the required permissions to this user in AWS.

1. Go to *Services->IAM->Groups*. Click *Create New Group*.



1. Go through the wizard steps to create the new user group. Enter the groupname and click *Next*.

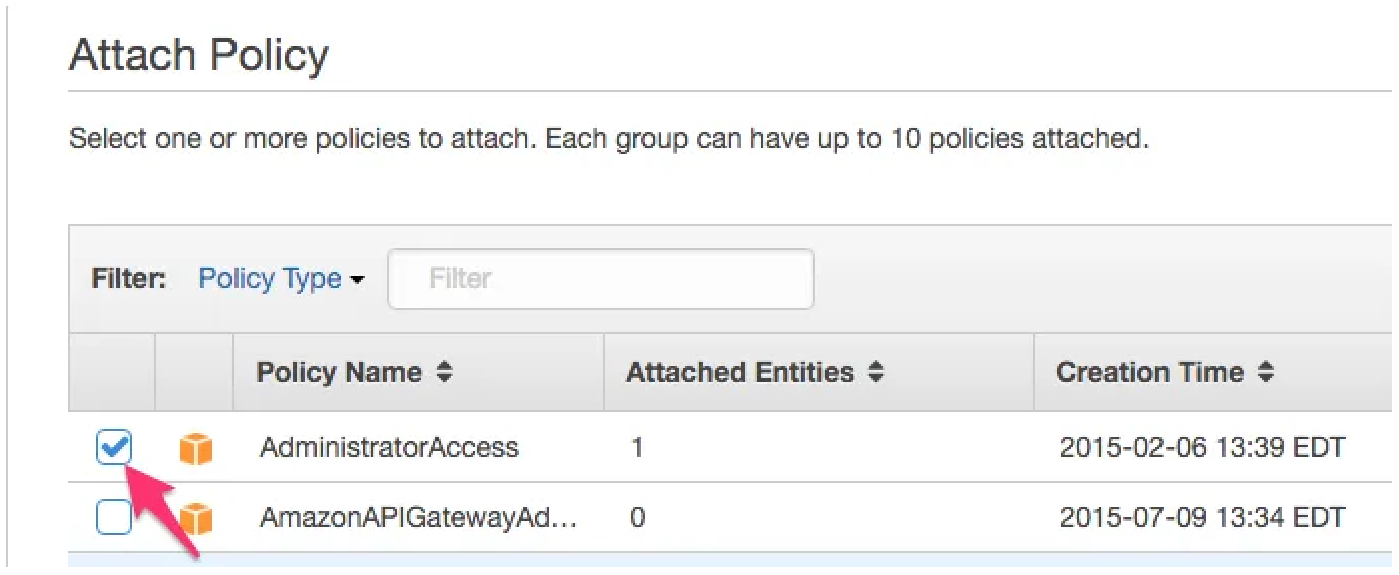


Set Group Name

Specify a group name. Group names can be edited any time.

Group Name: group

1. You will then see the option to attach a Policy. Check the first option, *Administrator Access*, and click *Next*.



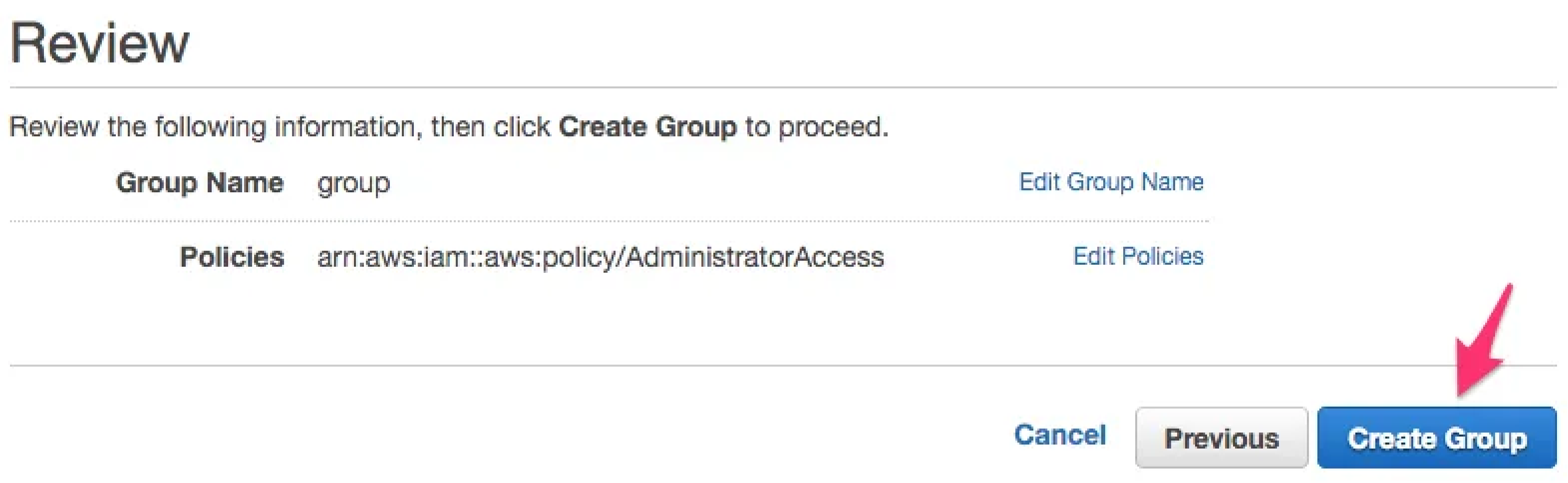
Attach Policy

Select one or more policies to attach. Each group can have up to 10 policies attached.

Filter: Policy Type ▾ Filter

		Policy Name	Attached Entities	Creation Time
✓		AdministratorAccess	1	2015-02-06 13:39 EDT
		AmazonAPIGatewayAd...	0	2015-07-09 13:34 EDT

1. You will see a Review screen. Click *Create Group* to finally create the group.

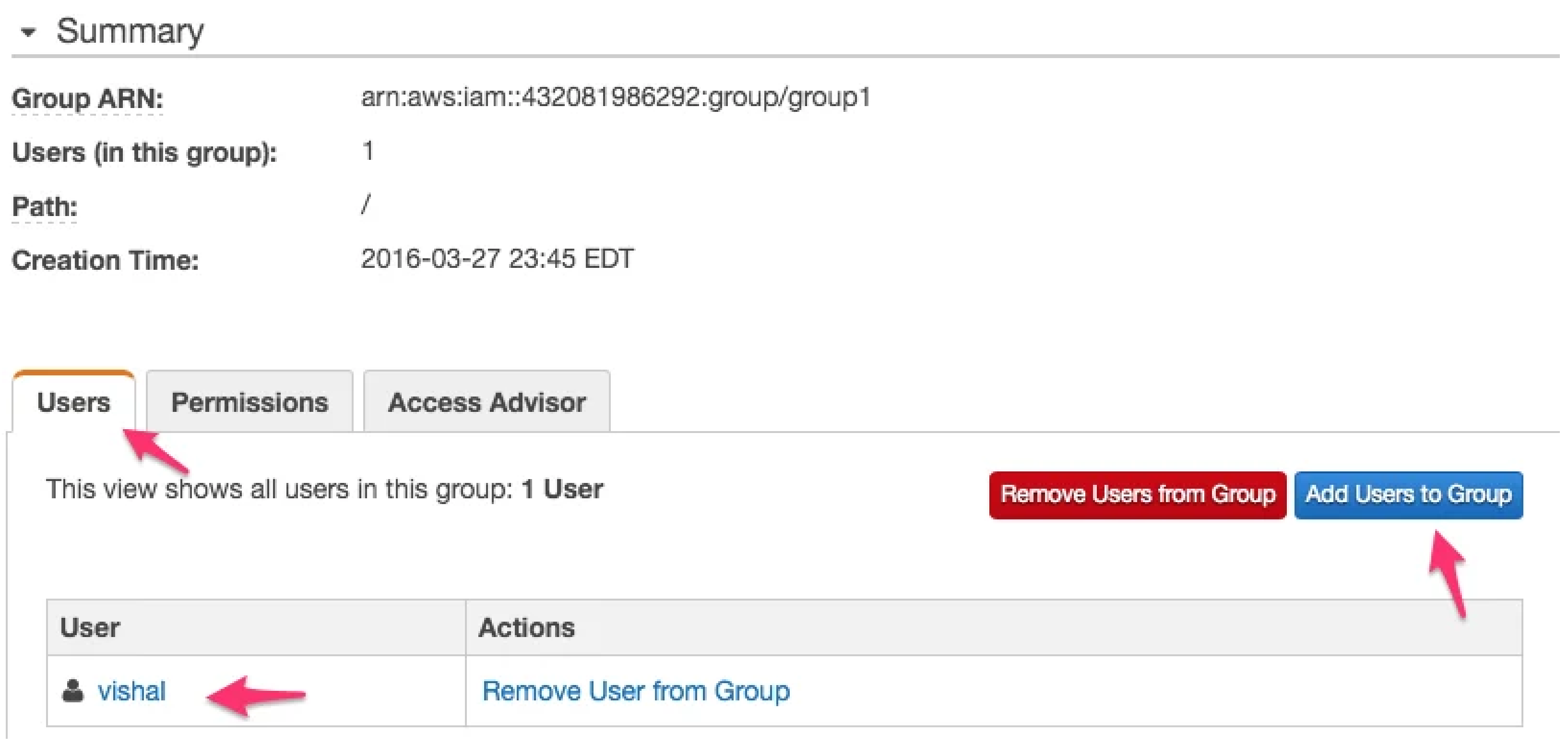


Review

Review the following information, then click **Create Group** to proceed.

Group Name	group	Edit Group Name
Policies	arn:aws:iam::aws:policy/AdministratorAccess	Edit Policies

1. Once the group is created, click on the name of the group to see a screen to manage users, permissions, etc. Add the user that you previously created to this group by clicking *Add Users to Group*.



Summary

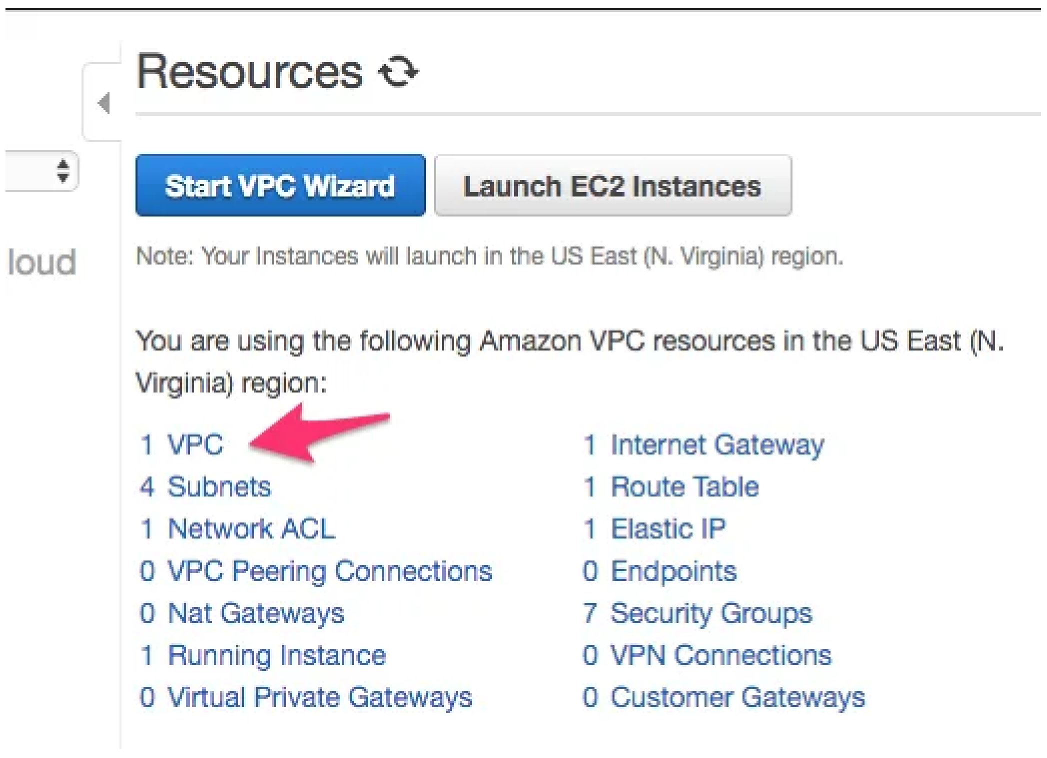
Group ARN:	arn:aws:iam::432081986292:group/group1
Users (in this group):	1
Path:	/
Creation Time:	2016-03-27 23:45 EDT

This view shows all users in this group: **1 User**

User	Actions
vishal	Remove User from Group

1. We will also need a vpc-id to launch our instance. While still logged on to AWS Console, click *Services->VPC*. Click on the link to see your VPC details. Copy the vpc-id from the grid view that appears. Now we are all set to

launch our instance.


Resources

Note: Your Instances will launch in the US East (N. Virginia) region.

You are using the following Amazon VPC resources in the US East (N. Virginia) region:

1 VPC	1 Internet Gateway
4 Subnets	1 Route Table
1 Network ACL	1 Elastic IP
0 VPC Peering Connections	0 Endpoints
0 Nat Gateways	7 Security Groups
1 Running Instance	0 VPN Connections
0 Virtual Private Gateways	0 Customer Gateways

To launch a new instance on AWS remotely from your computer, open a new shell window. Then use the Docker Machine `create` command to create a new EC2 instance. Use the `amazonec2-vpc-id` flag to specify the vpc-id. Use the `amazonec2-zone` flag to supply the zone in which your instance should exist. Finally, specify the name by which you would like to refer to the instance.

```
$ docker-machine create --driver amazonec2  --amazonec2-vpc-id [vpc-id-
copied-in-previous-step] --amazonec2-zone c aws07
Running pre-create checks...
Creating machine...
(aws01) Launching instance...
Waiting for machine to be running, this may take a few minutes...
Detecting operating system of created instance...
Waiting for SSH to be available...
Detecting the provisioner...
Provisioning with ubuntu(systemd)...
Installing Docker...
Copying certs to the local machine directory...
Copying certs to the remote machine...
Setting Docker configuration on the remote daemon...
Checking connection to Docker...
Docker is up and running!
To see how to connect your Docker Client to the Docker Engine running on
this virtual machine, run: docker-machine env aws07
```

Once this instance is launched and ready for use, we can tell docker execute commands against aws07 as follows:

```
$ eval $(docker-machine env aws07)
```

Now that we have Docker running on our Amazon instance, we can go ahead and run our containers.

As I mentioned before, we're going to run our MongoDB database and our web application on separate containers. I chose the official repo for Mongo on the docker repository. We can pull this image and run it as a detached container in one simple step:

```
$ docker run --name mymongodb -d mongo
```

The last argument mongo is the name of the image from which it should create the container. Docker will first search for this image locally. When it doesn't find it, it will go ahead and download it and all the base images that it is dependent on. Convenient!

Docker will then run this image as a container. -d flag ensures that it is run in detached mode (in the background) so that we can use this same shell to run our other commands. We can do a quick check after this to make sure that this container is up and running by using the docker ps command:

```
$ docker ps -a
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
2f93a31d4a3d mongo:latest "/entrypoint.sh mong About a minute ago Up
About a minute
27017/tcp mymongodb
```

The startup script script for this image already runs the mongo service listening on 27017 port by default. So there is literally nothing else we had to do here except for that one docker run command.

Running the MEAN Stack Container

The next phase of this project is to run our web application as a separate container.

The MEAN stack code base has a lot of dependencies like Node, Bower, Grunt, etc. But once again, we don't need to worry about installing them if we have an image that already has all these dependencies. Turns out there is an image on the Docker Hub that already has everything we need.

Once again, we will pull it in and run it with just one command:

```
$ docker run -i -t --name mymeanjs --link mymongodb:db_1 -p 80:3000
maccam912/meanjs:latest bash
...
..
Status: Downloaded newer image for maccam912/meanjs:latest
root@7f4e72af1cf0:/#
```

Now there is a lot going on with this single command. To be honest, it took me some time to get it exactly right.

1. The most important piece here is the `--link mymongodb:db_1` argument. It adds a link between this container and our mymongodb container. This way, our web application is able to connect to the database running on the mymongodb container. `db_1` is the alias name that we're choosing to reference this connected container. Our MEAN application is set to use `db_1`, so it's important to keep that name.
2. Another important argument is `-p 80:3000`, where we're mapping the 3000 port on the container to port 80 on the host machine. We know that web applications are accessed through the default port of 80 using the HTTP protocol. Our MEAN application is set to run on port 3000. This mapping enables us to access the same application from outside the container over the host port 80.
3. We of course have to specify the image from which the container should be built. As we discussed before, `maccam912/meanjs:latest` is the image we'll use for this container.
4. The `-i` flag is for interactive mode, and `-t` is to allocate a pseudo terminal. This will essentially allow us to connect our terminal with the stdin and stdout streams of the container. This stackoverflow question explains it in a little more detail.
5. The argument bash hooks us into the container where we will run the required commands to get our MEAN application running. We can bash into a previously running Docker container, but here we are doing all that with just one command.

CODESHIP
TAYLOR JONES
SOFTWARE CHEF AT IZEA
Test-Driven Development for JavaScript
CODESHIP.COM - BLOG.CODESHIP.COM - RESOURCES.CODESHIP.COM

https://js.hscta.net/cta/current.js

hbspt.cta.load(1169977, 'a48f90f2-519c-4f75-a875-d9dd7c4827d6', {});

Building and Running our MEAN Application

Now that we're inside our container, running the `ls` command shows us many folders including one called `Development`. We will use this folder for our source code.

cd into this folder and run `git clone` to get the source code for our MEAN.JS application from GitHub:

```
root@7f4e72af1cf0:/# cd Development/
root@7f4e72af1cf0:/Development# git clone
https://github.com/meanjs/mean.git meanjs
Cloning into 'meanjs'... remote:
....
..
Checking connectivity... done.
```

cd into our MEAN.JS folder. We can run `npm install` to download all the package dependencies:

```
root@7f4e72af1cf0:/Development# cd meanjs
root@7f4e72af1cf0:/Development/meanjs# ls
Dockerfile LICENSE.md Procfile README.md app bower.json config fig.yml
gruntfile.js karma.conf.js package.json public scripts server.js
root@7f4e72af1cf0:/Development/meanjs# npm install
```

A couple of hiccups to watch out for: For some reason, my npm install hung during a download. So I used Ctrl + C to terminate it, deleted all packages to start from scratch, and ran npm install again. Thankfully, this time it worked:

```
^C
root@7f4e72af1cf0:/Development/meanjs# rm -rf node_modules/
root@7f4e72af1cf0:/Development/meanjs# npm install
```

Install the front-end dependencies running by running bower. Since I'm logged in as the super user, bower doesn't like it. But it does give me an option to still run it by using the `--allow-root` option:

```
root@7f4e72af1cf0:/Development/meanjs# bower install
bower ESUDO Cannot be run with sudo
....
You can however run a command with sudo using --allow-root option
root@7f4e72af1cf0:/Development/meanjs# bower install --allow-root
```

Run our grunt task to run the linter and minimize the js and css files:

```
root@7f4e72af1cf0:/Development/meanjs# grunt build
...
Done, without errors.
```

Now, we are ready to run our application. Our MEAN stacks looks for a configuration flag called `NODE_ENV`, which we will set to production and use the default grunt task to run our application. If you did all the steps right, you should see this final output:

```
root@7f4e72af1cf0:/Development/meanjs# NODE_ENV=production grunt
...
..
MEAN.JS application started
Environment: production
Port: 3000
Database: mongodb://172.17.0.1/mean
```

Validating Our Application from the Browser

Our application would have given errors if there was some problem running it or if the database connection failed. Since everything looks good, it's time to finally access our web application through the browser.

But first, we'll need to expose our virtual machine's port 80 over HTTP.

1. Go back to the EC2 dashboard.
2. Click on the security group link for the given instance. You should see the settings page for the security group.

1. Click the "Inbound" tab at the bottom, and then click the "Edit" link. You should see that SSH is already added. Now we need to add HTTP to the list of inbound rules.

1. Click "Add Rule."

1. Select HTTP from the dropdown menu and leave the default setting of port 80 for the Port Range field. Click "Save."
2. Pick up the URL to our instance from the Public DNS column and hit that URL from the browser.

You should see the homepage of our fabulous application. You can validate it by creating some user accounts and signing in to the app.

So that's it. We've managed to run our application on AWS inside isolated Docker containers. There were a lot of steps involved, but at the crux of it all, we really needed to run only two smart Docker run commands to containerize our application.

PS: If you liked this article you can also download it as an eBook PDF here: Running a MEAN Web Application in Docker Containers on AWS

Related Content

Git Push: An In-Depth Tutorial With Examples

BLOG 8 min read

Meet Adrienne Tacke, the Developer's Best Friend

BLOG 4 min read

Why Do Enterprises Grow from Jenkins? Part 1: Fragile Instances

BLOG 5 min read

Git Squash: How to Condense Your Commit History

BLOG 8 min read

Git Stash: A Detailed Guide to Shelving Your Code

BLOG 8 min read

The Democratization of Development in the Enterprise

BLOG 4 min read

BLOG 4 min read

Capabilities

Resources

Why CloudBees