International Journal of Science and Research (IJSR)
ISSN: 2319-7064
SJIF (2022): 7.942

Running MEVN Stack on AWS App Runner

Jagadeesh Kancharana

Abstract: ***In this paper, we demonstrate the deployment of a MEVN (MongoDB, Express.js, Vue.js, Node.js) stack application using Docker with AWS App Runner. The process includes dockerizing the MEVN stack, pushing Docker images to Amazon Elastic Container Registry ECR, and creating an App Runner service to seamlessly manage and scale the application. The article details the deployment process, covering prerequisites, MongoDB Atlas setup, externalizing environment variables, and configuring the Docker runtime for production. This paper also provides insights into application logging, testing, and considerations for future enhancements such as automatic deployments and custom domain configurations. The practical steps outlined herein offer a comprehensive guide for developers looking to leverage AWS App Runner for deploying containerized web applications efficiently.***

Keywords: MEVN stack, AWS App Runner, Docker, MongoDB Atlas, Full stack deployment

To deploy your application on the managed platform by selecting the runtime, AWS App Runner is the right choice. You can run the entire API using Docker runtime without worrying about the configuration. AWS App Runner is an AWS service that provides a fast, simple, and cost-effective way to deploy straight from source code or a container image directly to a scalable and secure web application in the AWS Cloud.

You can dockerize the MEVN Stack and deploy that in the Docker runtime. The Docker images can be pulled from ECR, etc. In this post, we will see how to run MEVN Stack with Docker runtime on AWS App Runner.

- Prerequisites
- Example Project
- Set up a MongoDB Atlas
- Build For Production
- Externalize Environment Variables
- Dockerize the Project
- Running the WebApp on Docker
- Pushing Docker Image To ECR
- Creating an App Runner Service
- Testing The WebApp
- Application Logs
- Summary
- Conclusion

1. Prerequisites

If you are new to web development, go through the below link on how to develop and build MEVN Stack.

How To Develop and Build MEVN Stack

The other prerequisites to this post are Docker essentials. We are not going to discuss the basics such as what is a container or Docker. Below are the prerequisites you should know before going through this article

Docker Essentials

You need to understand Docker concepts such as creating images, container management, etc. Below are some of the links that you can understand about Docker if you are new.

Docker Docs

Docker — A Beginner's guide to Dockerfile with a sample project
Docker — Image creation and Management
Docker — Container Management with Examples
Understanding Docker Volumes with an example

AWS Prerequisites

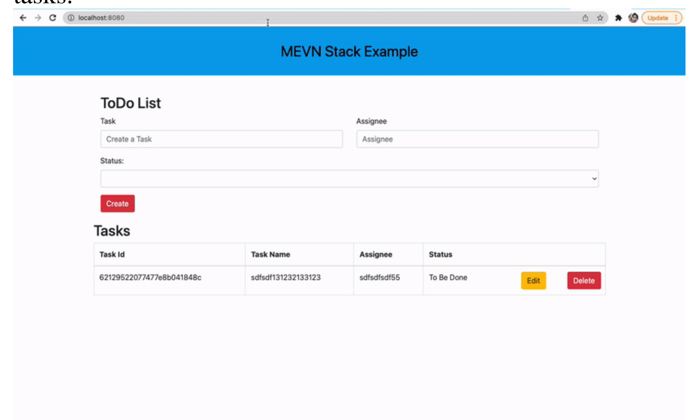
Amazon is a leading cloud provider and a pioneer in cloud computing. AWS provides more than 200 services, and it's very important to know which service you should select for your needs.

If you are new to AWS or just getting started you can see the following article.

How To Get Started With AWS

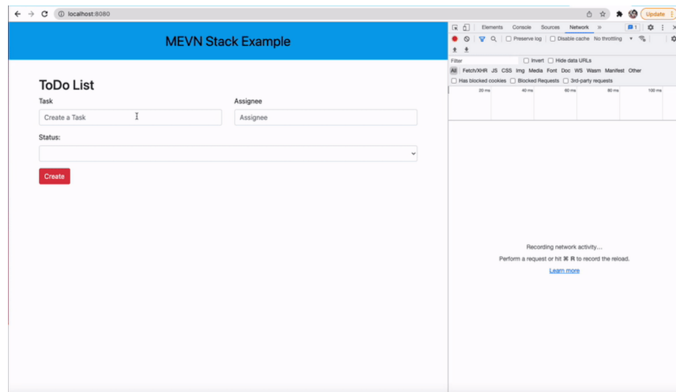
1) Example Project

Here is an example of a simple tasks application that creates, retrieves, edits, and deletes tasks. We actually run the API on the NodeJS server and you can use MongoDB to save all these tasks.



Example Project

As you add users we are making an API call to the NodeJS server to store them and get the same data from the server when we retrieve them. You can see network calls in the following video.



Network Calls

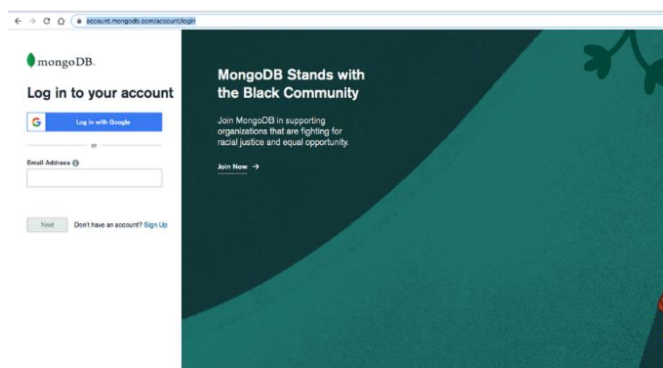
Here is a Github link to this project. You can clone it and run it on your machine.

```
// clone the project
git clone https://github.com/bbachi/mevn-stack-example.git

// Vue Code
cd ui
npm install
npm run serve

// API code
cd api
npm install
npm run dev
```

2) Set up a MongoDB Atlas

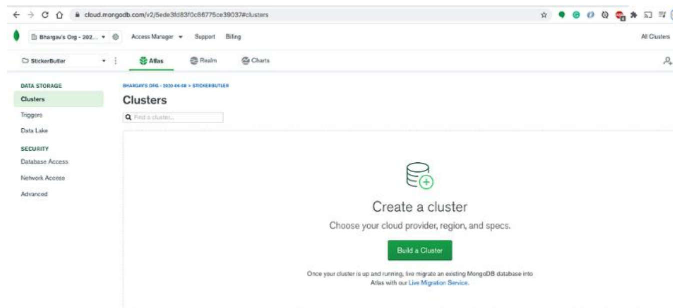
The core of MongoDB Cloud is MongoDB Atlas, a fully managed cloud database for modern applications. Atlas is the best way to run MongoDB, the leading modern database. There are two ways to deploy MongoDB on AWS and you can check them here on this page. We are using fully-managed MongoDB Cluster for this post.

Let's create your MongoDB Account here. You can either log in with any of your Gmail accounts or you can provide any other email address to create the account.



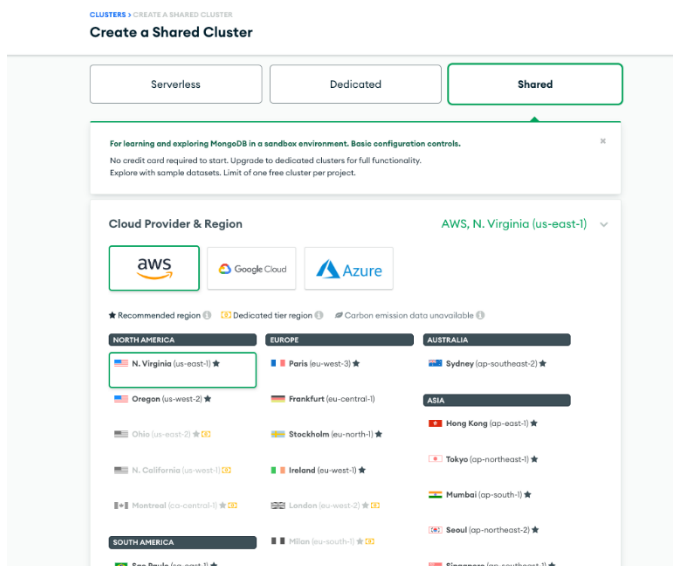
MongoDB Atlas login

Once you log in with your account you will see the dashboard below where you can create clusters.



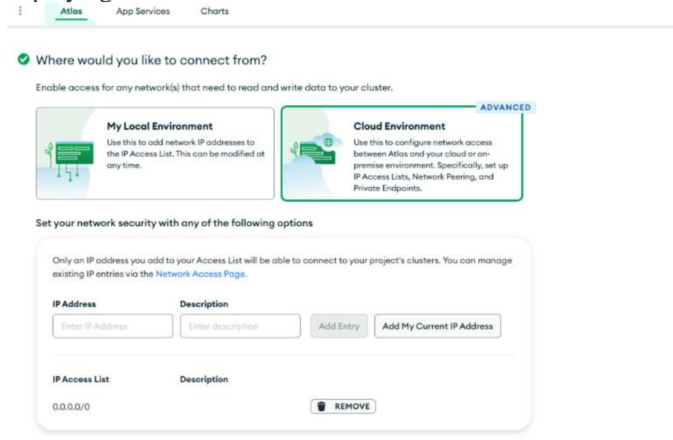
MongoDB Dashboard

Let's create a cluster called todo-cluster by clicking on the build a cluster and selecting all the details below. Make sure you select AWS Cloud.



Creating a Cluster

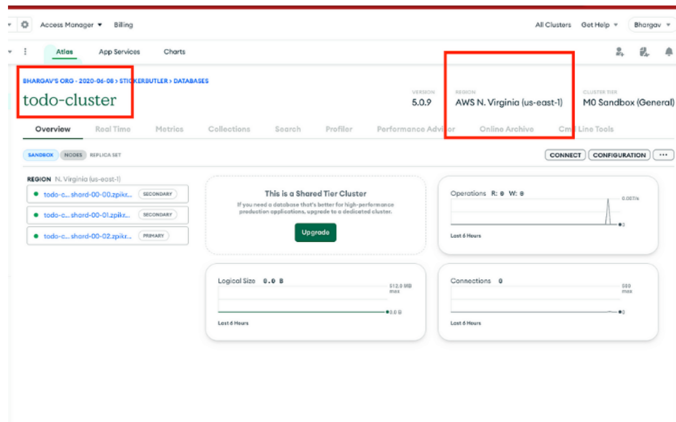
Make sure you select the Cloud Environment since we are deploying this on AWS Cloud.



Cloud Environment

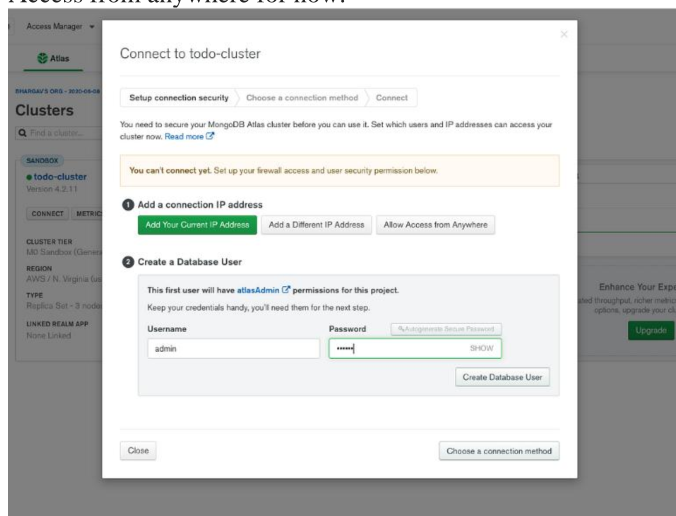
Here is the cluster we created below.

International Journal of Science and Research (IJSR)
ISSN: 2319-7064
SJIF (2022): 7.942



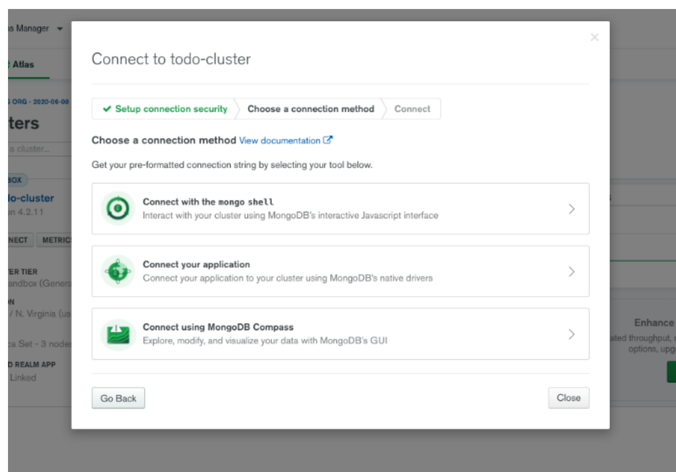
todo-cluster

You can click on the connect button to see the details about connecting to the cluster. You need to create a user and Allow Access from anywhere for now.



Connecting to cluster

You can see three ways of connecting to the cluster on the next screen.

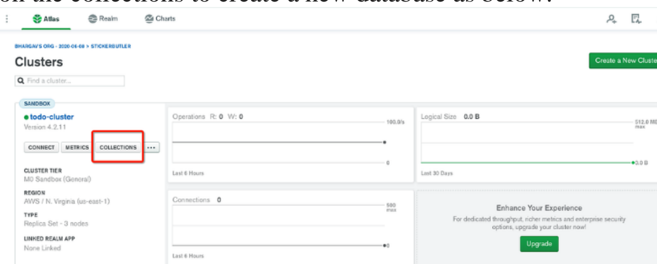


Ways of connecting

We will see all these three ways to connect to the cluster in the next sections.

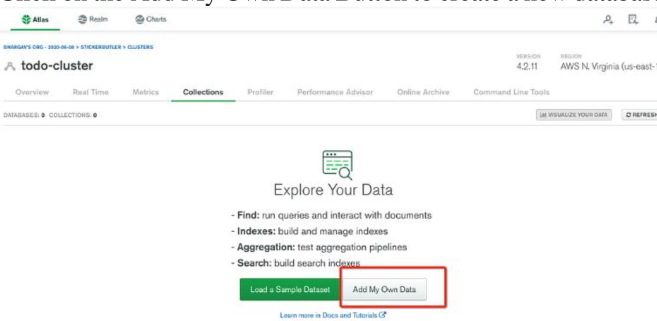
Create a Database

We have created a cluster and it's time to create a database. Click on the collections to create a new database as below.



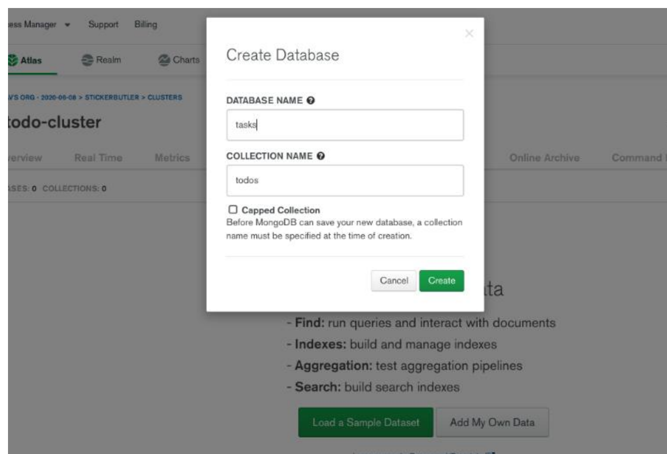
Collections

Click on the Add My Own Data Button to create a new database.



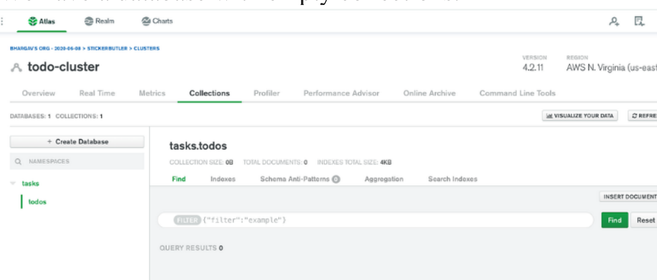
Add My Own Data

I have given a database name as tasks and the collection name is todos.



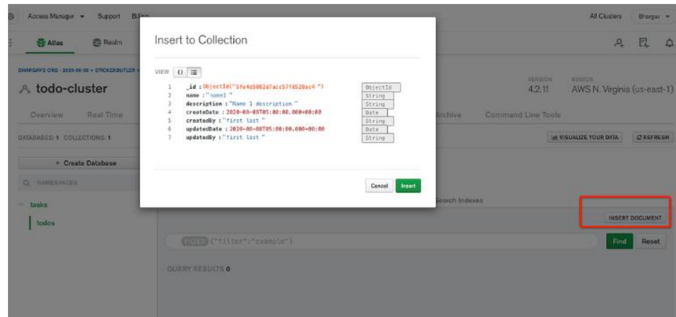
Creating a Database

You will see the below dashboard once the database is created. We have a database with empty collections.

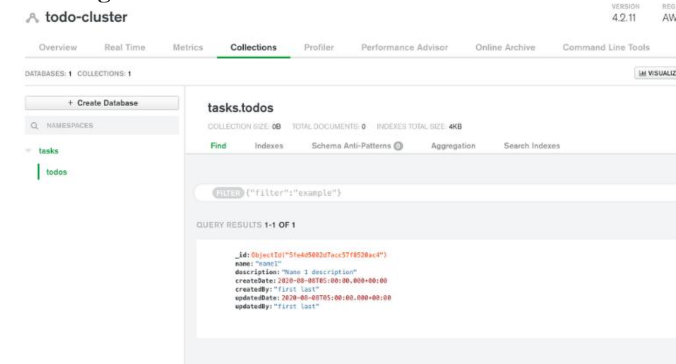


Empty Collection

Let's insert the first document into the collection by clicking the button insert document



Inserting the Document



Document Inserted

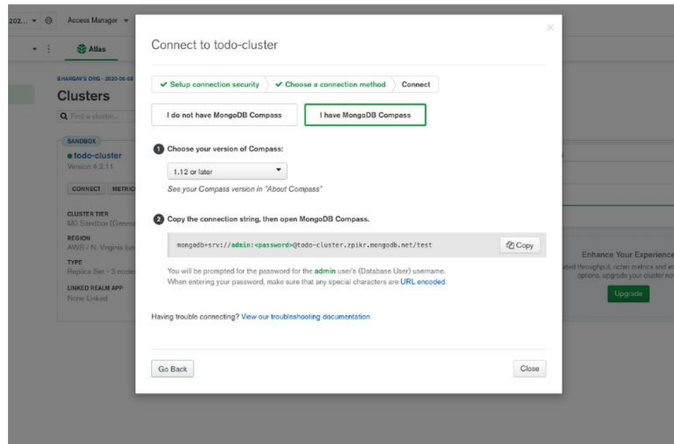
I. Connect With Mongo Compass

We have seen three ways we can connect to this cluster and read the collections. Let's connect to the database with Mongo Compass. The first thing we need to do is to download and install Mongo Compass from this link.

Let's get a connection string from the Atlas dashboard as below.

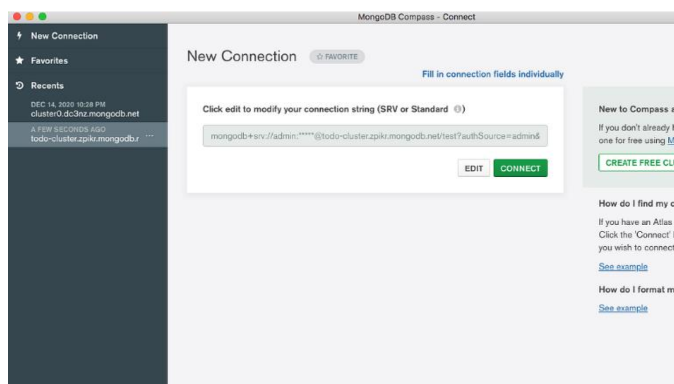
Connect with MongoDB Compass

Replace the password with the password that you created above.



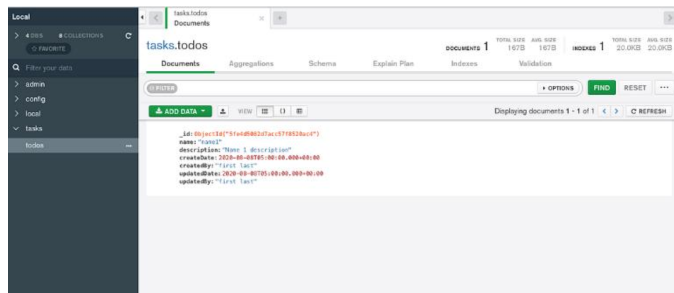
Connection String

Let's connect to the database with the connection string.



Connect with Connection String

You can actually see the same collection in the MongoDB Compass as well.



MongoDB Compass

Here is the connection string that you can connect to MongoDB.

```
mongodb+srv://admin123:admin123@todo-cluster.zpikr.mongodb.net/?
retryWrites=true&w=majority
```

3) Build For Production

There are so many ways you can build MEVN Stack for production and it depends on the use case or where we are deploying the MEVN Stack. This article explains different ways to build MEVN Stack for production.

How to Build MEVN Stack for Production

4) Externalize Environment Variables

Reading environment variables is one of the most common things that we do when we are building apps. It doesn't matter whether you are developing front end app or back-end API you have so many variables that should be outside of your application source code that makes your app or API more configurable. For example, if you want to hide logger statements in production or do something else based on the environment you can pass this as an environment variable. If you want to change later all you need to change is in one place.

Reading Environment Variables In NodeJS REST API

When it comes to this application, there are two environment variables one is the Mongo Connection string and another one is PORT.

```
PORT=80
MONGO_CONNECTION_STRING=mongodb+srv://admin123:admin123@todo-
cluster.zpikr.mongodb.net/?retryWrites=true&w=majority
```

You have to put these in the webpack.config.js file so that these values are used when we dockerize the app for **production**.

```
1   const path = require('path');
2   const webpack = require('webpack');
3
4   const environment = process.env.ENVIRONMENT;
5
6   console.log('environment::::', environment);
7
8   let ENVIRONMENT_VARIABLES = {
9     'process.env.ENVIRONMENT': JSON.stringify('development'),
10    'process.env.PORT': JSON.stringify('3080'),
11    'process.env.MONGO_CONNECTION_STRING': JSON.stringify('mongodb://mongo-db:27017')
12  };
13
14  if (environment === 'test') {
15    ENVIRONMENT_VARIABLES = {
16      'process.env.ENVIRONMENT': JSON.stringify('test'),
17      'process.env.PORT': JSON.stringify('3080'),
18      'process.env.MONGO_CONNECTION_STRING': JSON.stringify('mongodb://mongo-db:27017')
19    };
20  } else if (environment === 'production') {
21    ENVIRONMENT_VARIABLES = {
22      'process.env.ENVIRONMENT': JSON.stringify('production'),
23      'process.env.PORT': JSON.stringify('80'),
24      'process.env.MONGO_CONNECTION_STRING': JSON.stringify('mongodb+srv://admin123:admin123@todo-
25    };
26  }
27
28  module.exports = {
29    entry: './server.js',

32      filename: 'api.bundle.js',
33    },
34    target: 'node',
35    plugins: [
36      new webpack.DefinePlugin(ENVIRONMENT_VARIABLES),
37    ],
38  };
```

webpack.config.js hosted with ❤ by GitHub view raw

webpack.config.js

5) Dockerize the WebApp

Amazon EKS is a managed service that makes it easy for you to run Kubernetes on AWS. The first thing you need to do is to dockerize your project.

We use multi-stage builds for efficient docker images. Building efficient Docker images are very important for faster downloads and lesser surface attacks. In this multi-stage build, building a Vue app and putting those static assets in the build folder is the first step. The second step involves building the API. Finally, the third step involves taking those static build files and API build and serving the Vue static files through the API server.

We need to update the server.js file in the NodeJS API to let Express know about the Vue static assets and send the index.html as a default route. Here is the updated server.js file. Notice the line numbers **41** and **20**.

```
1   const path = require('path');
2   const express = require('express');
3   const bodyParser = require('body-parser');
4
5   console.log('environment    ', process.env.ENVIRONMENT)
6   console.log('PORT    ', process.env.PORT)
7   console.log('MONGO_CONNECTION_STRING    ', process.env.MONGO_CONNECTION_STRING)
8   if(process.env.ENVIRONMENT !== 'production') {
9       require('dotenv').config()
10  }
11
12
13  const taskController = require('./controller/task.controller')
14
15
16
17  const app = express();
18  const port = process.env.PORT || 80;
19
20  app.use(express.static(path.join(__dirname, "./ui/build")));
21  app.use(bodyParser.json());
22
23  app.get('/api/tasks', (req, res) => {
24      taskController.getTasks().then(data => res.json(data));
25  });
26
27  app.post('/api/task', (req, res) => {
28      console.log(req.body);
29      taskController.createTask(req.body.task).then(data => res.json(data));
30  });
31
32  app.put('/api/task', (req, res) => {
33      taskController.updateTask(req.body.task).then(data => res.json(data));
34  });
35
36  app.delete('/api/task/:id', (req, res) => {
37      taskController.deleteTask(req.params.id).then(data => res.json(data));
38  });
39
40  app.get('/', (req, res) => {

44
45
46  app.listen(port, () => {
47      console.log(`Server listening on the port  ${port}`);
48  })
```

server.js hosted with ❤ by GitHub view raw

server.js

Let's build an image with the Dockerfile. Here are the things we need for building an image.

Stage 1

- Start from the base image node:14-slim
- There are two package.json files: one for the Node.js server and another for the Vue.js UI. We need to copy these into the Docker file system and install all the dependencies.
- We need this step first to build images faster in case there is a change in the source later. We don't want to repeat installing dependencies every time we change any source files.
- Copy all the source files.
- Install all the dependencies.
- Run npm run build to build the Vue App and all the assets will be created under dist a folder within the ui folder.

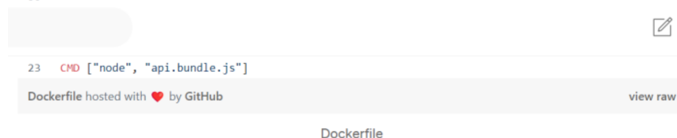
Paper ID: ES24822092645 DOI: https://dx.doi.org/10.21275/ES24822092645

Stage 2
- Start from the base image node:14-slim
- Copy the nodejs package.json into ./api folder
- Install all the dependencies
- Finally, copy the server.js into the same folder

Stage 3
- Start from the base image node:14-slim
- Copy all the built files from UI Build
- Copy all the built files from API Build
- Finally, run this command node api.bundle.js

Here is the complete Dockerfile for the entire build.

```
1   # Stage1: UI Build
2   FROM node:14-slim AS ui-build
3   WORKDIR /usr/src
4   COPY ui/ ./ui/
5   RUN cd ui && npm install && npm run build
6
7   # Stage2: API Build
8   FROM node:14-slim AS api-build
9   WORKDIR /usr/src
10  COPY api/ ./api/
11  RUN cd api && npm install && ENVIRONMENT=production npm run build
12  RUN ls
13
14  # Stage3: Packagign the app
15  FROM node:14-slim
16  WORKDIR /root/
17  COPY --from=ui-build /usr/src/ui/dist/ ./ui/build
18  COPY --from=api-build /usr/src/api/dist .
19  RUN ls
```



Dockerfile

Let's build the image with the following command.

```
// build the image
docker build -t mevn-image .

// check the images
docker images
```

6) Running the WebApp on Docker

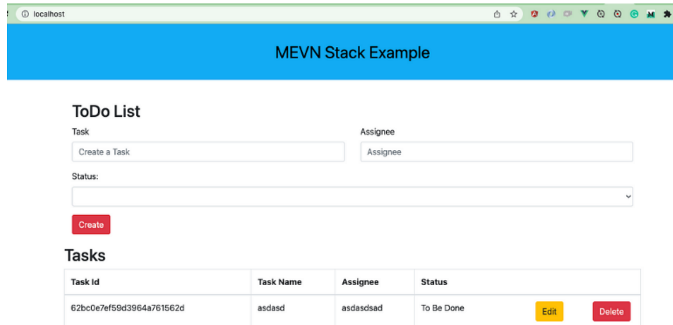
Once the Docker image is built. You can run the image with the following command.

```
// run the container
docker run -d -p 80:80 --name mevn-stack mevn-image

// list the container
docker ps

// logs
docker logs mevn-stack

// exec into running container
docker exec -it mevn-stack /bin/sh
```

```
bash-3.2$ docker run -d -p 80:80 --name mevn-stack mevn-image
61162dbba4fa59a5ee7078eaa190f93659bde9b2d1aae54a27d846bfeddd64cb
bash-3.2$
bash-3.2$ docker ps
CONTAINER ID   IMAGE        COMMAND                  CREATED         STATUS         PORTS                NAMES
61162dbba4fa   mevn-image   "docker-entrypoint.s…"   3 seconds ago   Up 2 seconds   0.0.0.0:80->80/tcp   mevn-stack
bash-3.2$
```

docker ps

You can access the application on the web at this address http://localhost



MEVN Stack Running on port 80

7) Pushing Docker Image To ECR

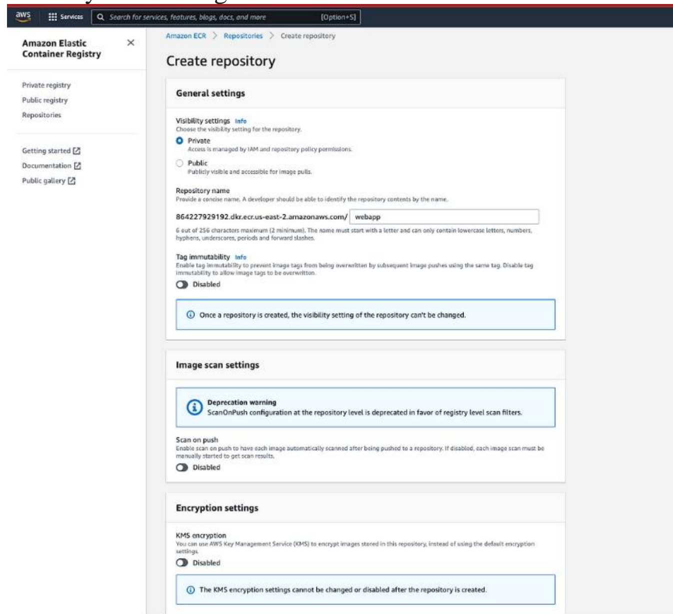
Amazon Elastic Container Registry (ECR) is a fully-managed Docker container registry that makes it easy for developers to store, manage, and deploy Docker container images. Amazon ECR is integrated with Amazon Elastic Container Service (ECS), simplifying your development to production workflow.

Amazon EKS works with Amazon ECR and ECR Public. However, in this article, we explore how to use Amazon ECR to store Docker images. Once you set up the Amazon account and create an IAM user with Administrator access the first thing you need to create a Docker repository.

You can create your first repository either by AWS console or AWS CLI

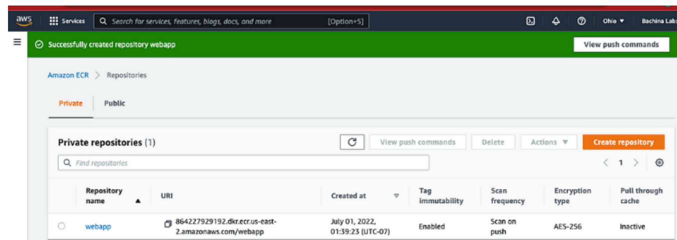
a) *AWS Console*

Creating a repository with an AWS console is straightforward and all you need to give a name.



Paper ID: ES24822092645 DOI: https://dx.doi.org/10.21275/ES24822092645

Creating a Repository



Repository

b) AWS CLI

The first thing you need to do is authenticate to your default registry. Here is the command to authenticate to your default registry. You need to make sure you are putting the correct regions and account id in the command.

```
aws ecr get-login-password --region us-east-2 | docker login --
username AWS --password-stdin aws_account_id.dkr.ecr.us-east-
2.amazonaws.com
```

```
bash-3.2$ aws ecr get-login-password --region us-east-2 | docker login --username AWS --password-stdin 864227929192.dkr.ecr.us-east-2.amazonaws.com
Login Succeeded
bash-3.2$
```

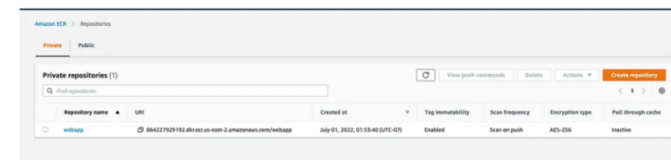
Authenticating to ECR

```
aws ecr create-repository --repository-name rest-api --image-
scanning-configuration scanOnPush=true --image-tag-mutability
IMMUTABLE --region us-east-2
```

```
bash-3.2$ aws ecr create-repository --repository-name webapp --image-scanning-configuration scanOnPush=true --image-tag-mutability IMMUTABLE --region us-east-2
{
    "repository": {
        "repositoryArn": "arn:aws:ecr:us-east-2:864227929192:repository/webapp",
        "registryId": "864227929192",
        "repositoryName": "webapp",
        "repositoryUri": "864227929192.dkr.ecr.us-east-2.amazonaws.com/webapp",
        "createdAt": "2022-07-01T01:53:40-07:00",
        "imageTagMutability": "IMMUTABLE",
        "imageScanningConfiguration": {
            "scanOnPush": true
        },
        "encryptionConfiguration": {
            "encryptionType": "AES256"
        }
    }
}
bash-3.2$
```

Creating a Repository

You will have the same result as well.



Repository

a) Tagging your local Docker image and Pushing

You created a Docker image on your local machine earlier. It's time to tag that image with this repository URI in the above image.

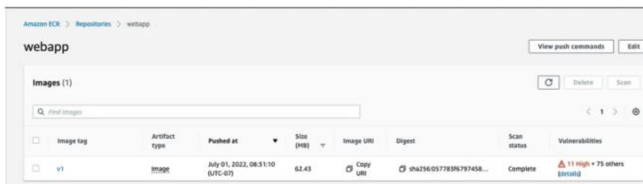
```
docker tag webapp:latest 864227929192.dkr.ecr.us-east-
2.amazonaws.com/webapp:v1
```

Once you tag the image and it's time to push the Docker image into your repository.

```
// list the images
docker images

// push the image
docker push 864227929192.dkr.ecr.us-east-2.amazonaws.com/webapp:v1
```

```
bash-3.2$
bash-3.2$ docker push 864227929192.dkr.ecr.us-east-2.amazonaws.com/webapp:v1
The push refers to repository [864227929192.dkr.ecr.us-east-2.amazonaws.com/webapp]
743c0db8307e: Pushed
09f85063052b: Pushed
1a61156d77c2: Pushed
5f70bf18a086: Pushed
2cbf69b076c5: Pushed
6d1172fd8548: Pushed
2635fead7a6c: Pushed
00a7d47e2fb5: Pushed
992f38e1a81c: Pushed
v1: digest: sha256:057783f679745885bae27fc37743f3f53d8b792183c90d0503ab4f6685b756c8 size: 2199
```

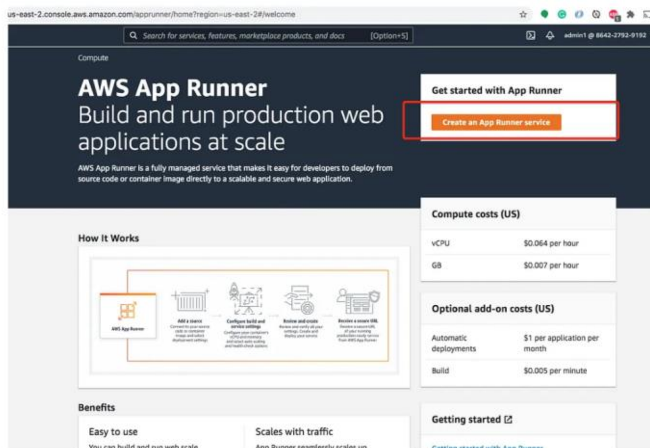
Pushing Docker Image



Docker Image Pushed to Repository

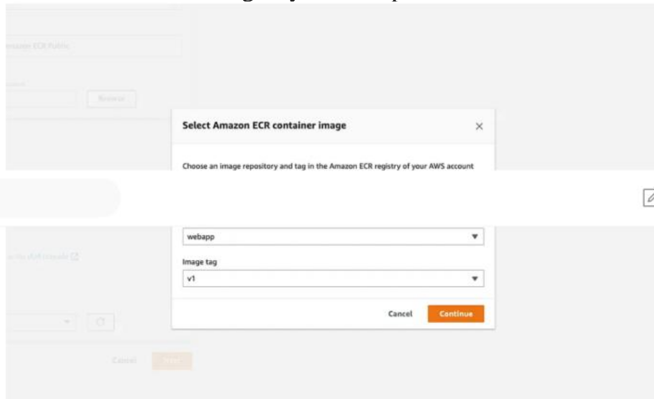
8) Creating an App Runner Service

We have seen how to dockerize the Python REST API, pushing the image to ECR in the above sections. Let's create App Runner from the services section.



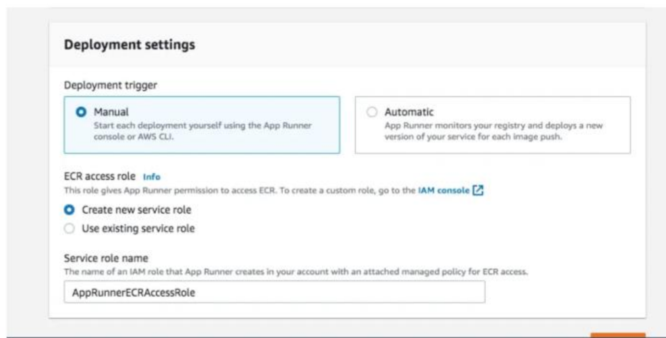
Getting Started

Since we are using the Docker image, we need to select the source as Container Registry and the provider as Amazon ECR.



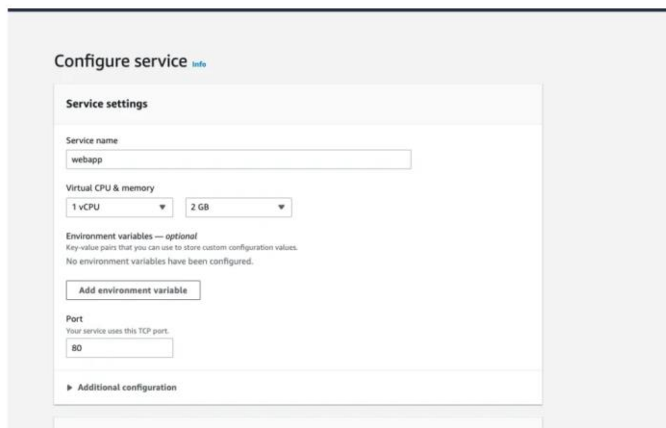
Choosing the Image

You can choose the manual under the Deployment settings and choose the new service role as below. You can click next now.



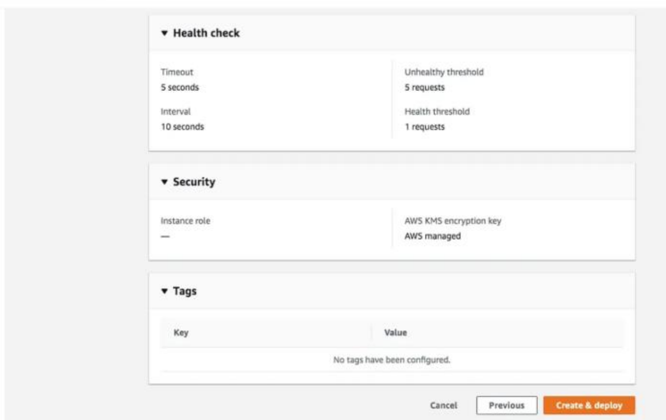
Deployment Settings

The next important things are the service name and the port in which API is listening. Let's give the service name and the port as below.



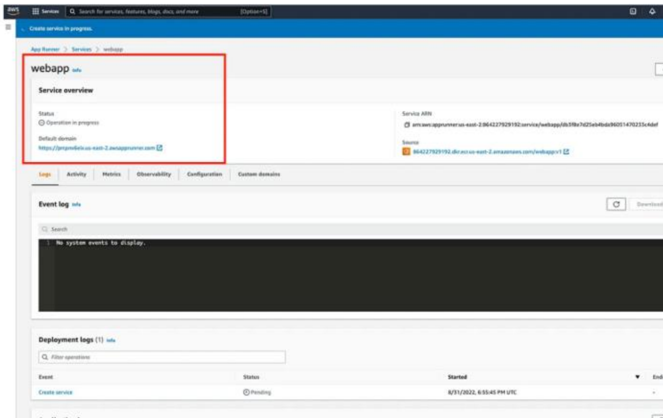
Configure Service

Finally, click the Create Deploy button on the final review page.



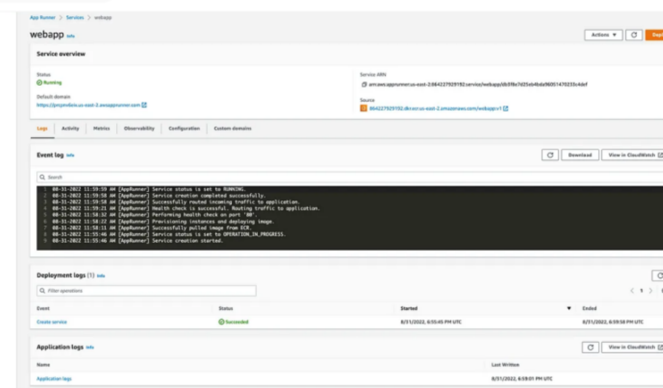
Creating Deployment

Once you click on that button, the service is created as below. It takes some time to complete the deployment.



Service is being created

If everything goes well, you can see the following message and status changes to running as below.

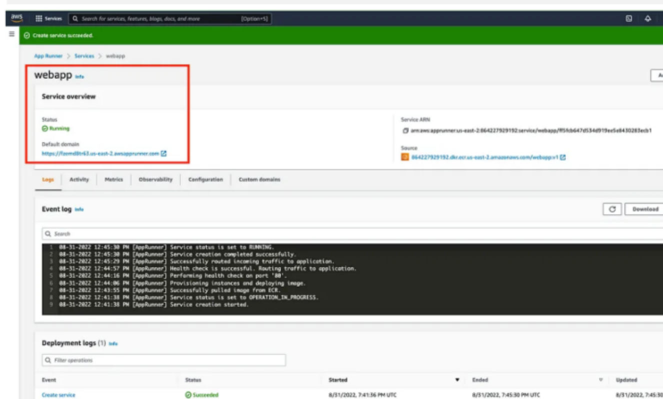


Service Deployed

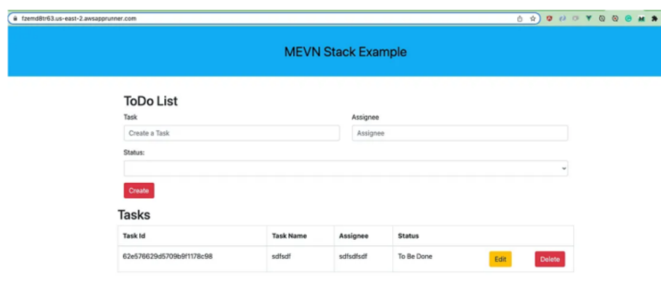
9) Testing The WebApp

You can take the default domain URL from the below location and hit it in the browser with the actual path.

```
https://fzemd8tr63.us-east-2.awsapprunner.com/
```



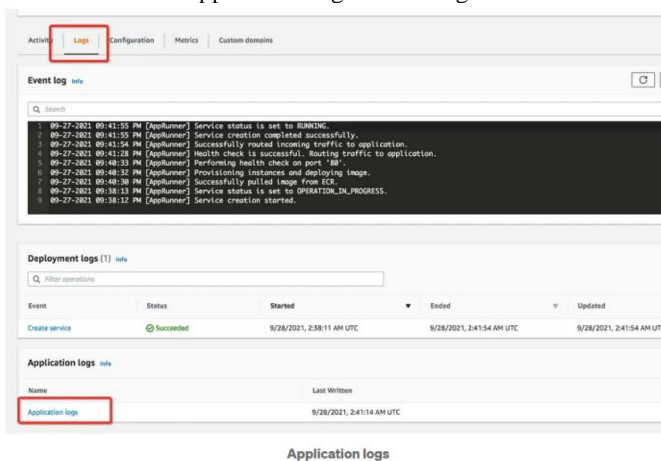
Service Running



WebApp Running

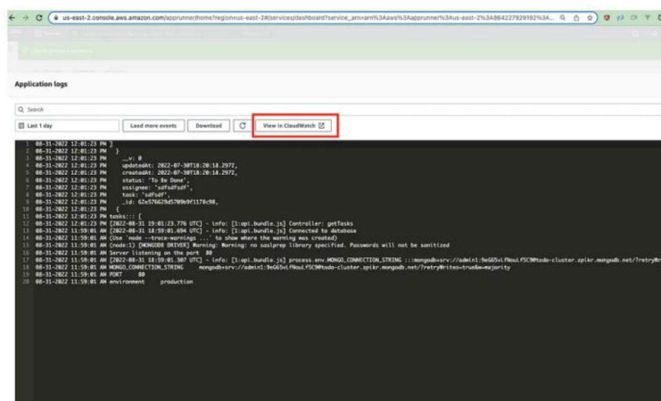
Application Logs

You can view the application logs in the Logs section below.



Application logs

You can click on the link to see the logs in detail. You can even see these logs in Cloudwatch as well.



Logs in detail

2. Summary

- If you want to deploy your application on the managed platform by selecting the runtime, AWS App Runner is the right choice.
- You can run the whole API with Docker runtime without any worry about the configuration from your side.
- You can dockerize the WebApp and deploy that in the Docker runtime. The Docker images can be pulled from ECR, etc.
- Amazon Elastic Container Registry (ECR) is a fully-managed Docker container registry that makes it easy for developers to store, manage, and deploy Docker container images.
- AWS App Runner is an AWS service that provides a fast, simple, and cost-effective way to deploy straight from source code or a container image directly to a scalable and secure web application in the AWS Cloud.

3. Conclusion

We have seen how to run the MEVN Stack with Docker Runtime on AWS App Runner. In future posts, we can see how to do automatic deployments, configure environment variables, link custom domains, etc, and other configuration stuff.

References

1. JavaScript Documentation https://developer.mozilla.org/en-US/docs/Learn/Getting_started_with_the_web/JavaScript_basics
2. VueJS Documentation https://vuejs.org/guide/introduction.html
3. Bachina, Bachina. (2024). *Ultimate Full Stack Web Development with MEVN*. OrangeAVA. Available at https://orangeava.com/collections/all-ebooks/products/ultimate-full-stack-web-development-with-mevn?variant=45552578560218
4. Bhargav Bachina. (2022). Architecting and Deploying MERN Stack Applications on AWS ECS. Journal of Scientific and Engineering Research, 9(2), 123–142. https://doi.org/10.5281/zenodo.10903268